IEEE COMMUNICATIONS SURVEYS
The Electronic Magazine of Original Peer-Reviewed Survey Articles

FORMAL METHODS FOR SPECIFICATION AND ANALYSIS OF COMMUNICATION PROTOCOLS

FULVIO BABICH AND LIA DEOTTO

ABSTRACT

Formal methods (FM) are mathematically-based techniques that provide a rigorous basis for software development: the application of FMs makes it possible to achieve provable correctness and reliability in the various steps of system design and implementation. This article is a tutorial presentation of formal methods and description techniques that address modeling and analysis of distributed systems and concurrent processes in telecommunications and protocol engineering. The aim of the article is to introduce to non-practitioners the main formal methods for communication protocols. For each method, a synthetic description of its textual or graphical syntax is provided. Also, the modeling capabilities and the basic communication features are pointed out by the application of the method to a common case study, a simple retransmission protocol. The tutorial description is completed by examples of recent applications of FMs to the specification and analysis of transmission and signaling protocols in industrial and research projects, which describe the methods' application domain and offer selected references for further readings.

The development of a complex software system, be it a safety-critical control program or the entire protocol stack of a complex information processing system, is an engineering discipline, consisting of a sequence of tasks, leading from the prototype to the code. It comprises the capturing of system requirements, service description, SW/HW partitioning, performance prediction, design and implementation of the software components.

Traditionally the development process is carried out in an informal way, relying on informal textual documentation, graphical description techniques, and structural analysis and design. Informal methodologies, lacking any scientific foundation, lead to ambiguous definitions of the desired features and offer no means to prove the completeness and consistency of the system design and implementation. Error checking is carried out with conformance testing, based on heuristic test-suites whose coverage decreases with system complexity. If faults are encountered, the code or even the entire development process shall be reviewed, thus delaying commercial release. If testing fails and functional errors appear after commercialization, the financial cost can be immense. The final check is often covered by human expertise (by means of peer-reviews, walk-throughs, and informal inspection), which cannot eliminate the chance that errors will appear during system operation.

Conversely, a rigorous approach to system design and implementation can be accomplished with the adoption of mathematically-based techniques, which enable the introduction of rigorousness and reliability into the various steps of the development process. Such techniques are called formal methods (FM) and represent the theoretical foundation of software engineering methods in the same way classical mechanics supports civil engineering. As in any other engineering discipline, in addition to a few technical issues that can be addressed with mere practical experience, there are critical aspects that require scientific, hence "formal," quantification and resolution, whose added value is the mathematically supported correctness of the results.

The need for such tools in software engineering has long been recognized (see [1] for an early introduction). In the last three decades research in computer science strongly contributed to the definition and formalization of formal description techniques (FDT). Several formalisms and related automatic tools have been developed and are currently available (for a comprehensive list of notations, methods, and tools see [2]). Within the wide spectrum of formal techniques, this tutorial focuses on a selected subset of FDTs, which represents, either for historical reasons or recently achieved popularity, the essential state of the art in the field of communication systems and protocols.

Formal Methods in Protocol Engineering

Though it can speed up the development process and give more confidence in the final implementation, the formal approach to communication protocols is neither widespread nor well established. One of the main objections to the adoption of formal methods in the industrial world is that their indisputable merits are counterbalanced by the high costs in terms of time and resources incurred by users, costs that are required to become familiar with the technique. Actually, learning a specification language is as difficult as learning a programming language, because the number of syntactic operators is similar and no particular mathematical background is required. By means of a theoretical introduction and practical examples, this tutorial aims to introduce the main formal methods and illustrate how protocol development can be conveniently managed with the help of formal techniques and, by refering to up-to-date applications in the field of communication protocols, to expand the interest and trust in such an approach.

Application Fields of Formal Methods

When adopted in the critical steps of the code development process (validation, verification, testing), FMs can help in systematically producing highly reliable software with costs and time consumption that are comparable to traditional methods, but which can be capitalized in software maintenance and reuse [3]. However, as the history of successful industrial projects has taught [4, 5], FMs wll not replace traditional development techniques, but rather integrate them with additional insight into the system's behavior, early error detection, unambiguous documentation and, for high-integrity systems, mathematical correctness proofs. More precisely, the main applications of FM in the steps of SW development are:

- **System specification**: A formal approach to system requirements enables unambiguous detection of the main features that embody the service definition and the system behavior.
- **Validation and verification**: High-level formal prototypes result in executable specifications,[1] which can be simulated and checked for proper behavior by means of model-checking [6].
- **Functional testing**: Conformance or interoperability testing can be carried out with optimal test suites that are produced from a formal model of the system behavior [7].
- **Rapid prototyping**: Reliable source code can be produced in an automatic (although not yet efficient) manner thanks to the provably correct mapping between the syntaxes of specification and programming languages.
- **Performance testing**: Performance measures are usually obtained with simulation or theoretical investigation. A quantitative analysis of the implementation efficiency in the presence of stochastic phenomena can be obtained if formal prototypes are automatically linked to performance models.

[1] In most formal approaches for communicating processes, the system specification can be translated into a dynamic model (e.g., finite state machine), which can be "animated" with a set of stimuli. Some formal methods for sequential and distributed algorithms (which are outside the scope of the present tutorial) result in a static representation of the process of implementation, through progressive refinements of the system features. This kind of specifications are not executable; the correctness can be verified theoretically by proving (with computer-assisted tools) that each refinement step from high-level formal specification to the final implementation preserves the required properties.

Model-checking routines perform validation of the system behavior starting from a mathematical definition of the properties to be checked, by verifying that the required properties are valid in all the reachable states and the execution sequences of the equivalent dynamic model. The state space size of such a model depends on the number of concurrent processes, the number and range of the internal variables, the type of the exchanged messages, and the nature of the communication (i.e., the possibility of message queuing and the maximum number of stored messages). If the state space size is too large to allow exhaustive exploration (state space explosion problem), partial validation can be performed on a subset of execution sequences, selected either randomly or according to user-defined criteria (by user-defined constraints on the range of local variables or by neglecting parameters that do not affect system behavior).

Validation concerns originally the logical consistency of communication rules and internal data processing (the absence of deadlocks, non-progressive loops, invariance violations). Full analysis of reactive systems also requires validation of temporal properties on timed models, which is usually carried out with model-checking of temporal logic formulas (TLF). TLFs specify a requirement on the relative order of events in the system behavior and are expressed in a notation called Linear Temporal Logic.

Besides interaction with their environment, real-time systems are affected by time elapsing: in order to deal with the density of time, a finite number of temporal intervals (called temporal regions) can be envisaged where the system behavior is invariant; within this assumption, the equivalent state space is finite and complete model-checking of logical and temporal properties can be executed on it.

Not all FDTs support all the above mentioned concepts and application domains: FMs are based on different theoretical models (state machine, process algebra, Petri nets, etc.) and philosophies (i.e., they can be more abstraction-oriented or implementation-oriented), which affect the extensibility of the methods and the scope of their application.

Tool support is another relevant factor that affects the practical applicability of FMs. In this regard, it shall be observed that the majority of FM-based tools have been developed by universities, because academia has long been the depository of FM theory and practice; for a few better-established FDTs, commercial packages are also distributed. Obviously there is a gap between the performance of the two families of tools: commercial packages often offer a general-purpose environment, whose high costs discourage wide acceptance by the medium to small business and the academic target. Freeware packages usually address a particular design problem and have the advantage of offering optimized and efficient solutions, often on a graphical user interface, a feature that is necessary to promote wider acceptance in the industrial world.

The FDTs reviewed in this article have been selected based on the availability of adequate freeware tool support, in order to provide the non-practitioner who intends to approach FMs with introductory information on appropriate and practicable solutions to address the various steps of software development with a formal approach.

Organization of the Tutorial

This article is a tutorial description of FMs for communication protocols, where a common framework is used to introduce the reader to the most popular notations. After an introduction to the formalisms and the mathematical principles exploited, each technique is illustrated by the specifica-

tion of the go-back-N ARQ protocol. An overview of available automatic tools is then presented and critical conclusions on both methods and tools are finally drawn, based on application examples taken from the literature and from the authors' experiences.

The FDTs are ordered according to the operational model on which they rely (finite state machine, process algebra, Petri Nets, timed automata). Although timed automata are the timed-version of finite state machine, they are introduced separately as the only method supporting model-checking of real-time systems. The majority of the selected FTDs are based on state machines, because of their familiarity in many branches of electrical engineering and their natural correspondence with communication protocol. The other operational models that are illustrated in the article are less intuitive but have equal capabilities in terms of description and analysis.

Fore the sake of completeness, a short paragraph is devoted to two more notations, MSC and UML, which are quite popular in telecommunications. The latter, in particular, has gained widespread diffusion thanks to the object-oriented modeling features it provides. Although the FMs that are illustrated in this tutorial do no support the object-oriented approach (with some exceptions that are noted), they can be integrated within an object-oriented modeling framework, for example, to model and validate the dynamic behavior of specific communication entities.

A final paragraph summarizes the main advantages of the formal approach and formulates critical conclusions on the most appropriate application domains for each of the discussed methods.

block B	block declaration	process P	process declaration
channel C [sig_1, ...] [sig_2, ...]	declaration of a channel connecting blocks	route R [sig_1, ...] [sig_2 ...]	declaration of a route connecting processes
	starting state	(x==0) (x>0) x (x<0)	decision block
state_name	control state	x:=expr(arg); set(T(Pld));	task with data update and timer handling
SIG(var)	input signal with parameter	*	notation for "all states"
SIG(var)	output signal with parameter	–	notation for the terminating state: "same state as originating one"
$SIG(var_N)$ $expr(var_{N-1})$	enabling condition on signal consumption	expr(var) priority 1	continuous signal, which has lower priority than input signal consumption
Process_X	process instantiation	Procedure_X	procedure call
SIG	high priority input signal	SIG	signal saving, to postpone signal consumption
newtype ... signal Req(Int); signallist I=Req,...	declaration block in system diagram, for the definition of new types, signals, signal lists.	dcl x type; timer T:=1.0; synonym N Int:=2;	declaration block in process diagram, for the declaration of variables, timers, constants (synonym).
none	spontaneous transition triggered by a null signal	any	"any" construct used to trigger non-deterministic transitions

TABLE 1. *Basic SDL graphical syntax.*

SDL

SDL (Specification and Description Language) is a formal notation evolved and standardized between 1976 and 1992 by ITU-T [8]. Several updated versions have been issued since then (SDL-2000 is the latest), following the object-oriented approach that has been tailoring the development of software engineering and programming languages in the last few years. SDL is a high-level general-purpose description language for event-driven, real-time and communicating systems; telecommunication systems and protocols are one of its main application fields [9].

The effectiveness and the intuitive graphical format of SDL have won it a widespread popularity in both the academic and industrial sectors and have led the standardization institutes, that is, ETSI (European Telecommunications Standards Institute) and 3GPP (Third Generation Partnership Project), which is the international body responsible for UMTS (Universal Mobile Telecommunications System) standardization, to include SDL diagrams in their official specification documents.

SYNTAX

SDL is based on Finite State Machines (FSM), a standard technique for studying reactive systems that dates back to the 1950s with Turing, Moore and Mealy machines. The system described by an SDL specification is actually an Extending Communicating Finite State Machine (ECFSM), because it consists of a set of concurrent processes, extended with variables and data space, which communicate by exchanging control signals (abstract stimuli) or structured messages (signals associated with parameters) on finite-length asynchronous channels.

In system specification, structural and functional aspects are distinguished. The system architecture is sketched by deploying the building blocks (possibly in a hierarchical/modular way) and the channels connecting them and by declaring the signals exchanged on each communication path. The behavior of each basic block is detailed by editing the internal processes in accordance with their required operations. The association among blocks and processes is accomplished in the block diagram, which contains the declaration of the processes (one or more for each block) and the routes connecting each process to other internal processes (if any) or to the block's interface.

Similar to a traditional FSM, SDL processes are characterized by a finite number of "macro" control states (which can be compositionally refined in SDL-2000) connected by a finite number of transitions. The transitions can be triggered by the reception of an input signal (possibly subject to so called *enabling conditions*, i.e., guard conditions on the current data values of the process), by the expiring of a previously set timer, or by the validity of a specified condition on the current value of the state variables (*continuous signal*). Within the execution of an enabled transition, a set of tasks (signal sending, procedure or macro calls, process instantiation, variable updating, evaluation of choices with multiple output branches, etc.) is performed. The execution of the transitions leads to a final state that, when equal to the originated one, is denoted as "same state."

SDL supports non-determinism (i.e. ANY concept, representing an alternative of equally possible actions), which can be useful to describe uncertainty in the environment and to validate accordingly all possible reactions of the system.

SDL graphical syntactical constructs are illustrated in Table 1.

Data Types

Processes can handle local variables and parameters of different data types. The data type concept in SDL was originally based on axiomatic definitions of Abstract Data Types (ADT), defined by the set of possible values and the operations that can be executed with them. With ADT it is possible to characterize types through inheritance, thus avoiding redundant redefinition of common operators, and to define parametric data types, i.e., type generators, used for example to define arrays of elements of any type, whose index can be of any type supporting internal ordering. Due to the complexity of ADT, a set of predefined data types and type generators are provided. In addition, SDL tools provide compatibility with ASN.1 (Abstract Syntax Notation 1, a formal notation for abstract data types widely used in the specification of PDU formats), and support object data and algorithmic definition of operators (SDL-2000).

Communication in SDL

In SDL, communication on the bidirectional routes that connect processes is always asynchronous: that is, each process has its own input queue, where signals coming from the environment or from another process are buffered and sequentially processed ("consumed"). Synchronous communication is provided by the remote-procedure call instruction. SDL also supports non-ordered signal reception by means of the SAVE operator, which postpones the consumption of a specified signal to the following transition.

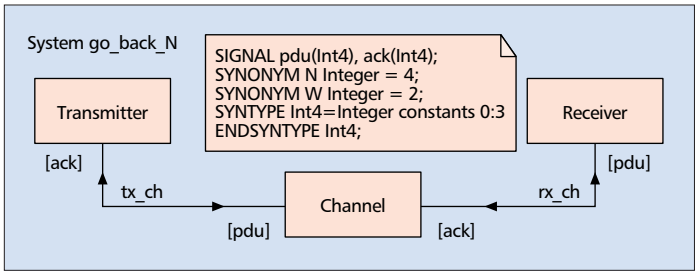

Figure 1. *Constitutive blocks of the go-back-N protocol specification. Signal, data types and constants are declared at system level.*

Example of Specification

Figures 1 and 2 illustrate the specification of the Go-Back-N Automatic Repeat Request (GBN ARQ) protocol governing data transfer between sender and receiver on a noisy channel. In the GBN ARQ protocol, the sender transmits data packets (PDU Protocol Data Unit) carrying the sequence number *ns*, and receives the Acknowledgments from the peer entity. For each correctly and orderly received PDU, the receiver issues an acknowledgement indicating the expected sequence number of the following packet; if duplicated PDUs are received, acknowledgments are repeated. Denoting the sequence number of the next expected PDU with *nr*, the reception of an Acknowledgment message carrying *nr* by the sender implies the acknowledgment of all pending PDUs with sequence numbers lower than nr. In order to achieve unambiguous numeration of data packets, the maximum number of pending PDUs (which is called the transmission window) shall be at most half of the available sequence numbers. When the transmission credit is exhausted, the sender starts retransmitting the pending PDUs. When an Acknowledgment is received, the transmission window can be updated and new PDUs can be issued and sent. In this example and in those following, sequence numbers range from 0 to 3 (4 sequence numbers) and the maximum window size W is fixed to 2.

In Fig. 1 the system diagram illustrates the system structural decomposition in three basic blocks: transmitter, channel and receiver. This kind of diagram usually comprises the global declaration of signals, signals lists, and definitions of new types and constants. In a separate diagram (not shown), each block is associated with the related process, represented by an octagonal shape, which carries for simplicity the same identifier of the block.

Figure 2 shows the diagrams of the three processes, which declare local variables to perform internal data handling. In this example, all processes perform a starting transition to their single control state, where they remain indefinitely. The initial transition in the process channel has been omitted due to lack of space.

The specification of the transmitter's behavior can be summarized as follows. As long as the transmission window (denoted with *win* and incremented for each sent PDU) is below the maximum value W, the process can transmit *pdu* messages following the spontaneous transition guarded by continuous signal $win<W$; if the transmission credit is exhausted within this transition (decision block $win=W$), a timer T is set to a default duration. The reception of an ack message resets the transmission window, stops the timer T and restarts the transmission of PDUs from sequence number *nr*. When the transmitter is in the stalling condition $win=W$, if no acknowledgement is received due to loss of *pdu* or *ack* messages on the channel, a timeout occurs, which triggers the retransmission of all pending PDUs.

The process receiver behavior is triggered by reception of *pdu* messages from the channel: it checks whether the sequence number of the received PDU matches the expected

sequence number, updates *nr* if possible, and issues an *ack* signal.

According to the system diagram, *pdu* and *ack* signals are conveyed to the channel process, whose handling of the signal *pdu* is sketched in Fig. 2: the unreliability of the channel is modeled with the ANY transition, which represents both the possibilities of correct forwarding and unsuccessful delivery (through silent consumption) of the signal. The handling of the *ack* signal, which has been omitted due to lack of space, can be obtained simply by changing the signal's identifier.

Tool Support

SDL is the formal specification language most exploited by telecommunication manufacturing companies worldwide, and a great deal of software has been produced on SDL-based platforms [10]. Several software packages have been developed to handle formal specifications of systems in SDL. Some freeware tools are available online. The most promising are JADE, a public domain tool for specification written in Java [11], for which code generation and optimization have been recently announced [12], and SITE, an open development environment supporting compilation of SDL and ASN.1 to target languages Java and C++ [13]. Unfortunately, these tools are not yet mature enough to adequately support system development with SDL and offer no significant alternative to commercial software.

The most successful commercial products are Telelogic Tau SDL Suite (SDT), by Swedish Telelogic, and ObjectGeode (recently acquired by Telelogic). In particular, SDT offers full support for SDL-2000, inclusion and automatic generation of C and C++ code (although not yet efficient enough for real-time applications), and linking to test generation and execution tools. Validation facilities incorporate model-checking routines and conformance testing of SDL prototypes by means of testing sequences provided in TTCN (Tree and Tabular Combined Notation, an ISO standard for the specification of tests for communication systems). The appealing support of the entire life cycle of software is counterbalanced by the high price of the packages, which usually induces users to eliminate what is deemed less critical, that is, validation, which can be performed with freeware tools based on other FMs (e.g, SPIN, see following section).

Although SDL provides timers, performance evaluation and analysis of temporal properties are poorly supported. Tools for performance analysis of queuing systems specified in SDL have been proposed, but are either frozen projects or not yet sufficiently mature for professional use.

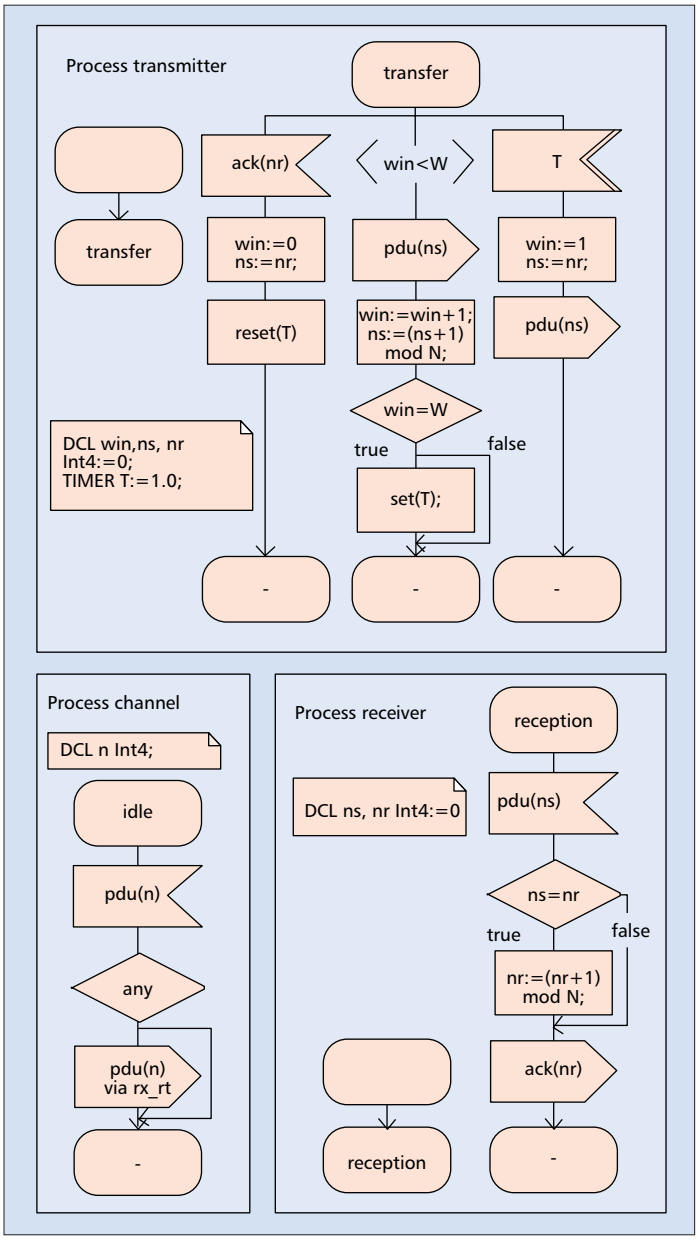

FIGURE 2. *SDL diagrams of the protocol entities: the channel's nondeterministic behavior is sketched only with respect to data handling.*

Examples of Applications

SDL has been successfully applied to system analysis and design in many application domains for many years. For an overview, see the report of the SDL and SAM (SDL and Message Sequence Chart) forum [14].

We have used Telelogic Tau SDL Suite for the specification and simulation of UMTS Radio Access Network (UTRAN) protocols involved in UMTS-GSM intersystem handover procedure, a key feature of the next-generation dual-mode terminals [15]. With the aid of the SDL modular approach, the software architecture has been designed according to a hierarchical structure, following the standard reference models for UMTS and GSM. For the definition of service primitives and PDU, thanks to the integration of techniques offered by the tool, the ASN.1 standard modules provided by 3GPP as technical documents have been imported and used to generate the encoding rules, which are reusable C functions.

Critical Evaluation

As far as the modeling phase is concerned, the modular approach and the clear distinction between structure and behavior are very useful features in describing OSI-like protocol architectures. In addition to this, the translation of the informal textual specifications issued by ETSI and 3GPP, for example, to SDL diagrams is a very straightforward task because the style adopted complies with SDL concepts and operators. Two other features that are worth mentioning are the inclusion of ASN.1 and C-type packages for a better message description, and the ability to employ the underlying C environment for efficient

data manipulation. Verification of static and dynamic properties of the system in SDT is performed by exhaustive or partial model-checking on the equivalent FSM produced by the combination of control states and data values. For complex SDL specifications, compositional verification and reliance on other formalisms (i.e., LOTOS [16], Spin [17, 18], and Petri Nets [19], which are introduced in later sections) have also been suggested. The lack of adequate freeware tool support for validation prompts the research for alternative solutions, that is, translators of SDL to other formal notations. The simulation phase, when not provided with sufficient quantitative information (i.e., timer values, error generation functions), shrinks to mere MSC tracing. In the hypothetical alternative use of SDT for simulation of broader systems, an intensive reliance on C functions and variables seems to be necessary and users will hardly take advantage of SDL friendly graphical notation.

SPIN

Spin (Simple ProMeLa Interpreter) is a widely used tool for specification, simulation, and validation of communication protocols, freely available online [20]. Being developed with the aim of rigorously supporting protocol engineering [21], Spin adopts a strong formal basis, established, like SDL, on ECFSM theory, and supports efficient model-checking, i.e., validation of consistency requirements, invariant assertions, and temporal properties expressed in an ad hoc Linear Temporal Logic.

Spin uses a C-like specification notation (ProMeLa — Process Meta Language) which increases its applicability in the first stages of the design and makes the following phase of code implementation a rather mechanical task of quantitative detailing.

SYNTAX

ProMeLa is a textual notation for ECFSM that comprises the constructs for data manipulation and communication between processes. The system's basic components are the processes, whose internal behavior is described as a set of possible transitions (gathered within the *if… fi* construct), which can be alternative or simultaneously enabled (non-determinism). The firing of a transition leads to termination (execution states labeled with *end*:) or non-terminating sequences of actions (when the transitions are part of a loop *do… od* or a *goto* operator points to a new control state). The triggering conditions guarding each branch are: message receiving and sending actions, which are true if executable; boolean expressions on local or global variables or on the status of visible channels; and timeout events, executed only when no other transition is enabled and the system is stalled (Table 2). All starting conditions inside *do… od* and *if… fi* constructs are introduced by operator "::" and separated by operator " " from the set of actions they trigger. All other actions in process behavior are delimited by operator ";."

The *timeout* construct mimics the possible occurrence of a timer's expiring, without quantitative or real-time concerns. This approach is close to that of SDL, in which a timer, once set, is treated as a signal and is processed when no other immediate, ordinary signal can be put into the input message queue. Modeling of multiple timer instances is described in [22]. To keep track of the temporal increment consequent to timeout events, a temporal counter should be explicitly updated for each time-consuming action. To cope with time and temporal constraints an enhanced version of Spin, called real-time Spin, was developed [23]. This version introduces clocks and relies on a proper semantic to convert the system in a finite state machine in spite of its continuous-time characteristics. The project seems to be frozen (it adopts an old version of Spin as the core program), but it is nevertheless worth mentioning as an effort to add further usefulness to a well established validator tool.

Processes can handle variables of predefined basic data types, as well as arrays and struct types (*typedef pdu* {$type_1$ $type\_id_1$; $type_2$ $type\_id_2$; …}). Besides, messages can be declared as symbolic constants (*mtype* = {*list_of_identifiers*}) and used as literals. The specification of channel types (*chan chan_id* = [*L*] of {$type_1$, $type_2$, …}) consists of the characterization of the length of the equivalent queue (*L*) and of the list of information elements constituting the PDU.

Processes can be immediately activated (by placing the attribute active before the process declaration) or defined as parametric types (*proctype proc_id* (*par_id par_type*; …)). In the last case they are instantiated by a meta-process *init* {*run proc_id*(*actual_par*);}, which is executed for the first and carries out the deployments of the others. They can also be activated by other processes, as independent entities with their own interfaces and channels or as procedures that deliver a result to the parent process, either on a channel or by overwriting a global variable (Fig. 3). Processes called procedures are declared in the same way and on the same level as other processes; hence, they are visible and available to them all.

General syntax and semantic interpretation of the if…fi construct

$state_P$: if :: $expression_1$ :: … :: $expression_N$ fi $state_Q$: …	The process remains in $state_P$ until one of the alternative expressions becomes executable. If the executable expression does not contain a statement *goto* $state_M$, the next process control state is $state_Q$.

General syntax and semantic interpretation of the do…od construct

$state_P$: do :: $expression_1$ :: … :: $expression_N$ od $state_Q$: …	The process remains in $state_P$ until one of the alternative expressions becomes executable. If the executable expression does not contain a statement *goto* $state_Q$, the next process control state is again $state_P$.

Examples of specifications in ProMeLa

```
if/do
:: bool_expr(vars)      /* boolean expression on variables */
:: else                 /* true if other conditions are false */
:: asyn_ch?mesg         /* executable if mesg present in channel async_ch */
:: asyn_ch!mesg         /* executable if channel async_ch is not full */
:: syn_ch?mesg          /* ex. if there is a coupling action syn_ch!mesg */
:: syn_ch!mesg          /* ex. if there is a coupling action syn_ch?mesg */
:: full(asyn_ch)        /* test on asynchronous channel: ex. if true */
:: timeout              /* true when whole system is blocked */
:: skip                 /* fictitious action: always executable */
fi/od
```

Table 2. *Syntax and semantics of the if…fi e do…od constructs and examples of transitions in ProMeLa.*

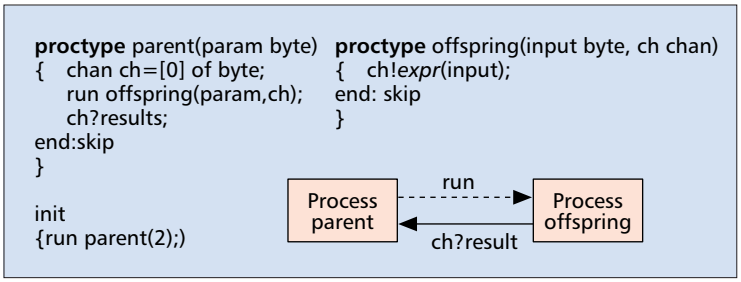

FIGURE 3. *Examples of specification of a procedure call in ProMeLa.*

Processes can be declared as *deterministic* if their behavior is not affected by external actions (*dproctype proc_id*). Deterministic processes and execution steps isolate the algorithmic part of the specification from the communicating and reactive part. They are treated as lumpable states and cause a smaller increase of the state space size, thus simplifying the following validation. Grouping of logically linked actions in *atomic* sequences has a similar effect.

COMMUNICATION IN PROMELA

Processes can communicate on channels, which are modeled as finite-length queues. The default communication is therefore asynchronous, but can also be synchronous when relying on channels of zero length. On these so called *rendezvous* channels, sending and receiving actions must match to be executed (they are interpreted as an atomic action), otherwise a deadlock is detected.

Channels deliver structured messages: sending actions are specified with the expression *ch*!*expr1,expr2,…* or equivalently with *ch*!*expr1*(*expr2,…*), where the first field is usually the message identifier. The reception of a message can be interpreted as data transfer (value-passing) — *ch*?*var1*,*var2*,…, when the values received on channel *ch* overwrite the local variables used by the receiving process — or, in contrast, the received values with constant expressions (value-matching) *ch*?*msg_id*($const_1$,$const_2$,…), when the message *msg_id* is extracted from the input queue *ch* provided that the received information elements equal the specified constants.

Information exchange can also be performed by writing and reading global variables, here interpreted as shared channels. This approach can help in modeling multicast transmission and MAC protocols.

EXAMPLE OF SPECIFICATION

Figure 4 illustrates the specification of the GBN ARQ protocol. In this example, only the two protocol entities are modeled. The unreliability of the connection between process *transmitter* and *receiver* is rendered by means of nondeterministic choice (*if* ::skip …; ::skip …; *fi*) between reception or silent consumption of data packets and acknowledgments.

The two processes communicate through unidirectional channels *tx* and *rx*; messages are constituted by a message type identifier (*pdu* or *ack*) and an integer value, which can encode the sequence numbers of sent and acknowledged PDU *ns* and *nr* (in this example, *nr* indicates the number of the last acknowledged *pdu*).

Two local arrays (*busy* and *rec*) store the state of sent and received data packets, respectively. In the process transmitter, *busy*[*ns*] is set to 1 for each PDU sent within the transmission window, and it is set to 0 for each acknowledgment. Based on the contents of the buffer *busy*, the integer local variable *h* is used to update, during the stalling condition ($w = W$), the transmission window *w*. Similarly, in the process *receiver*, *rec*[*ns*] is set to 1 for each correctly received PDU. The local variable *p* is used to store the bound of the reception window (the range of valid sequence numbers that can be received according to the concerted value of the maximum window size *W*). For sequence numbers outside the reception window, *rec*[*ns*] is set to 0. Thus, for each correctly received PDU (*rec*[*nr*] = 1), an *ack* message is issued and *nr* is set to the next expected sequence number.

When the system behavior stalls due to the corruption of *pdu* or *ack* messages, the *timeout* expression in the process *transmitter* becomes executable and triggers the retransmission of all pending PDUs in the transmission window.

It can be observed that there is no need for labels to identify the internal states, because the behavior of both processes (as in the previous example) is described by a single loop with multiple alternative transitions.

TOOL SUPPORT

Given a ProMeLa specification file, Spin performs simulation and validation on it. In the graphical version of the tool (xspin), these basic functions are integrated with MSC tracing facilities, generation of graphical FSMs starting from process models and analysis of process activity factor.

Preliminary analysis is carried out with random or interactive simulations. For a more detailed inspection, the validator checks for invalid end-states (deadlocks), non-progress cycles, violation of temporal claims, and so on, tracing back the shortest execution sequences to the incriminated states. When the system is too huge to be handled due to the bounds in available memory, the validation is carried out on a randomly selected subset of the whole state-space (sub-optimal solution). Recently a slicing routine has been added to extract, from complex protocol descriptions, partial specifications related to the properties of interest [24]. These concepts are powerful enough to cover almost all the correctness and consistence requirements that can be imposed on prototypes.

EXAMPLES OF APPLICATION

Spin is the most popular and exploited among freeware FM-based tools for the specification and verification of communication protocols. Extensive work has been carried out with its help, both in industrial and academic areas. Theoretical advancements and practical experiences are continuously reported in the proceedings of the International Spin Workshops. Interesting applications of Spin model-checking to C and Java code are presented in [25, 26].

Our expertise with Spin stems from the analysis of GPRS radio interface protocols and UMTS radio connection establishment procedures, which resulted in a very concise and provably correct specification of the essential signaling exchange [27].

The power of Spin as a verifier, together with its free availability for commercial purposes, justifies its frequent usage in the verification of large SDL models [17, 18]. Translating technical documents in SDL diagrams is trivial, thanks to the SDL-based style they adopt. Due to their common operational model, converting SDL into ProMeLa is a mechanical task because all SDL operators and concepts find a proper correspondence in ProMeLa [28]. Automatic translators have also been developed.

Critical Evaluation

Spin has proved to be very useful in the simulation and validation of system specifications. Modeling protocols from technical specification documents is rather straightforward and results in compact specifications, but specifying a complete protocol stack is not yet possible. Protocols ought to be isolated as orthogonal functions and verified layer after layer in the services they are suppose to render. To overcome the overhead of protocol processing often implied by layering and modularity in protocol design, a ProMeLa-based development environment is proposed in [29]. In that framework, protocols are specified and validated separately with Spin, but are treated as a whole in the final C-code generation. However, even without relying on rapid-prototyping, the software implementation of protocols is easily obtained from the formal model thanks to the similarities between C and ProMeLa.

Unlike other high-level FDTs, ProMeLa has no clear-cut separation between architectural and functional aspects. Due to the lack of hierarchical structuring facilities, it is not possible to point out the interdependencies between processes without looking at their behavior. A graphical interface to ProMeLa with hierarchical and object-oriented extensions has recently been presented [30], which is likely to grant more popularity to the tool.

Spin could be used as a mere simulator. As far as this application is concerned, an underlying computational environment, such as SDT inclusion of C code fragments, is indispensable. It should not be difficult to add these features in the simulator source code.

```
#define N 4       /*max sequence number*/
#define W 2       /*max window size*/
#define L 2       /*channel length*/
mtype={pdu,ack}  /*message type*/
chan tx=[L] of {mtype,int};       /*from transmitter to receiver*/
chan rx=[L] of {mtype,int};       /*from receiver to transmitter*/

active proctype transmitter( )
{   int busy[N];  /*pdu buffer: 1 if sent, 0 if ack'd*/
    int ns=0, h=0, nr=0, w=0;   /*w=window*/
do
::(w<W)->       w++; busy[ns]=1; tx!pdu,ns; ns++;
                ns=ns%N; /*ns mod N*/
::rx?ack,nr->   if
                :: skip->       atomic{busy[(nr-w)%N]=0;
                                busy[nr]=0} /*correctly received ack*/
                ::skip          /* corrunpted ack*/
                fi
::(w=W)->       if
                ::(busy[h]==0)-> w--;h++;h=h%N;
                ::(busy[h]==1)->   if
                                   ::timeout->tx!pdu,h /*re-transmission*;
                                   ::rx?ack,nr -> busy[nr]=0;
                                   fi
                fi
od
}

active proctype receiver( )
{   int rec[N];         /*pdu buffer: 1 if received, 0 if out of range*/
    int ns=0, nr=0, p=0;

do
::tx?pdu, ns->  if
                ::(rec[ns]==1)->    if /*already received*/
                                    ::((p-ns+N)%N>0&&(p-ns+N)%N<=W)->
                                          rx!ack,nr,
                                    ::else ->skip;              /*out of range*/
                                    fi
                ::(rec[ns]==0->     rec[ns]=1;     /*marked as received*/
                                          if          /*update ack message*/
                                          ::(N+p-W<N)->p=N+ns-W;
                                                       rec[p]=0;
                                          ::else ->p=ns-W;  rec[p]=0;
                                          fi;
                :: skip /*corrupted pdu*/
                fi
::(rec[nr]==1) ->rx!ack,nr;
                atomic{nr++; nr%N;} /*update counter of ack'd pdus*/
od
}
```

FIGURE 4. *Examples of specification of the GBN ARQ protocol.*

Estelle

Estelle is a formal description technique for the specification of distributed and concurrent systems approved as an international ISO standard in 1989 and particularly devoted to communication protocols. Estelle is based on ECFSM theory in modeling event-driven behaviors (similarly to SDL and Spin) and employs Pascal in data manipulation. With these two notations, both well established, a concise and consistent prototype is developed. The formal specification can be automatically compiled into an executable model or a target application, for simulation or implementation purposes, respectively.

Because of the inclusion of a programming language for the algorithmic segments of the specification, Estelle is more implementation-oriented than LOTOS (see the following section), which is conceived mainly for design and validation. Their formal nature enables automatic transformation from the latter to the former, for optimal support of the entire development process [31]. For an introduction to Estelle related to the other standard specification languages SDL and LOTOS, see [32].

Syntax

A standard Estelle specification is a unique document that collects several fields. The first section is comprised of the declaration of data types, channel types, procedures, and functions. Channels are bi-directional routes between two entities conveying structured values. The specification then describes a hierarchy of (possibly nested) module instances, each characterized by its own interface on which messages are exchanged by means of communication primitives. The basic element of the specification is the "module," comprising a *header*, which lists the module's interface by instantiating the appropriate channel, and a *body*, which characterizes the internal structure and/or functional behavior of the module. The body includes the formal specification of the modules (if any) used to refine its functional description, their declaration (multiple instantiations) and displacement (mapping of the channels) and the "textual" FSM depicting the system behavior as observable from other protocol layers. This modeling rule is recursively

applied to every substructure, preserving the coherence between the specification of each module and those that refine it.

In Estelle the system specification follows a strict order. All modules are defined as abstract entities and then actualized by giving them an identity and naming and connecting their channels, as can be observed in the following example.

```
specification go_back_n systemactivity;
default individual queue;      {queuing policy}
const N=4; const W=2;
type
    pdu_type=...;

function decode(pdu:pdu_type):integer; primitive;
function MOD_N(x:Integer;y:integer):integer;primitive;

channel Ch_type(arq_entity,medium);
    by arq_entity: Data_Req(n:integer);
    by medium: Data_Ind;    {contents are extracted from the receiving buffer}
channel T_type(arq_entity,t_entity);
    by arq_entity: Activate; Reset;
    by t_entity: Timeout;

module S_entity activity;
    ip  interface:Ch_type(arq_entity);      {towards lossy channel}
        control_ch:T_type(arq_entity);      {towards internal timer}
    end;
body sending for S_entity;
    state ACTIVE_STATE;      {list of control states}
    var ns:integer; var nr: integer; var 1_nr:integer; {local variables}
    var win:integer; var pdu:pdu_type;
initialize                                  {starting state}
    to ACTIVE_STATE
    begin
        ns:=0; nr:=0; win:=0;               {variable initialization}
    end;
trans                                       {list of transitions}
    from ACTIVE_STATE to same               {starting and ending states}
    provided win<W                          {guarding condition on variables}
        begin
            output interface.Data_Req(ns);          {sending action}
            win:=win+1; ns:=MOD_N(ns+1,N);          {data manipulation}
        end;
    from ACTIVE_STATE to same
    when interface.Data_Ind                 {signal reception}
        begin
            1_nr:=decode(pdu); output control_ch.Reset;
            win:=MOD_N(1_nr-nr+N,N); ns:=1_nr; nr:=1_nr;
        end;
    from ACTIVE_STATE to same
    when control_ch.Timeout                 {signal reception}
    provided win=2
    begin
        ns:=nr; win:=0;
    end;
end;

module R_entity activity;
    ip  interface:Ch_type(arq_entity);
    end;
body Receiving for R-entity; external       {specified in a separate document}

...(declaration of remaining modules)

modvar
    Sender: S_entity; Receiver: R_entity; Timer:T_entity; Medium: M_entity;
    initialize                  {association between bodies and modules}
    begin
        init Sender with Sending; init Receiver with Receiving;
        init Medium with Loss; init Timer with Count;
        connect Sender.interface to Medium.SAP1;
        connect Receiver.interface to Medium.SAP2;
        connect Sender.control_ch to Timer.control_ch;
    end;
end.
```

FIGURE 5. *Abstract from Estelle specification of the GBN protocol.*

EXAMPLE OF SPECIFICATION

Figure 5 illustrates part of the formal specification of the GBN ARQ protocol. The system *go_back_n* is described as the association of entities *Sender*, *Medium*, and *Receiver* (whose body is defined externally), which are instances of *S_module*, *M_module*, and *R_module*, respectively. The process *Sender* is connected to *Medium* through a channel (of type *Ch_type*) that exchanges the service primitives *Data_Req* and *Data_Ind*. The process *Receiver* comprises a similar interface.

The *Sender* is also connected through a control channel to the process *Timer*, whose specification is not included in the figure. The *timeout* signals produced by process *Timer* are handled when the transmission credit is exhausted (i.e., *provided w* = 2). The timer is reset by process *Sender* (instruction: *output control_ch.Reset*) at the reception of acknowledgment messages from process *Medium* (*when interface.Data_ind*).

In the handling of the *Data_ind* carrying the acknowledgment, the sequence number is extracted (*l_nr=Data_Ind.n*) and used to update the transmission window (*win:=MOD_N*(*l_nr-nr+N,N*)) and the sequence number of the next PDU to transmit (*ns:=l_nr*).

Figure 5 illustrates all state transitions of process *Sender*, whose simple interpretation is left to the reader, who can profit from the explanations given in the previous examples. In the final section *mod-var…end*, the four entities (*Sender*, *Receiver*, *Medium*, and *Timer*) are instantiated with the related modules and associated to the related bodies within the construct *initializer… end*. Finally, the common interfaces are connected with the construct *connect… to*.

TOOL SUPPORT

Several tools for design, debugging, simulation, and testing of Estelle specifications have been developed at universities and are distributed free of charge. Among Estelle commercial tools, the most complete one is the Estelle Development Toolset (EDT), which supports model animation with MSC tracing and implementation through C-code generation.

EXAMPLE OF APPLICATIONS

Estelle has been used in several real-life examples for the specification, simulation, and testing of communication protocols. Recent reports of successful case studies are found in [33–35]. Large system verification, usually not achievable with exhaustive state-space exploration, has been addressed with probabilistic partial-state occurrence analysis [36], or relying on different operational models for which adequate model-checking packages exist [37].

```
specification System:noexit
behavior (* system declaration*):
    Process_A[gates_A](actual_parameters_A)
    |[common_gates_AB]|
    Process_B[gates_B](actual_parameters_B)
where
process Process_A [formal_gates_A](formal_parameters_A):noexit:=(*definition*)
(*list of actions ending with a recursive instantiation of the same process*)
    list_of_actions; Process_A[gates_A](expr(formal_parameters_A)]
(*or, alternatively, declaration of the process as a composition of sub-processes*)
    subprocess_A1[gates_A1](actual_pars_A1 or expr(formal_pars_A))
    |[gates_A1-2]|
    subprocess_A2[gates_A2](actual_pars_A2 or expr(formal_pars_A))
endproc
endspec
```

FIGURE 6. *Structure of a formal specification in LOTOS*

CRITICAL EVALUATION

In Estelle, system models are specified as ECFSM, recalling the successful SDL approach toward asynchronously communicating concurrent processes. Support of Pascal for data definitions and calculations makes Estelle more suitable for implementation and testing issues than model-checking.

Although the theoretical foundations of the language have been thoroughly investigated in more than 10 years of research, Estelle's complex textual notation has not been sufficiently updated. More friendly development environments as well as graphical editors are necessary to meet industrial needs.

LOTOS

LOTOS (Language Of Temporal Ordering Specifications) is a formal description technique standardized by ISO in 1989 for specifying concurrent and communicating systems [38]. Together with Estelle (standardized within the same ISO committee responsible for the definition of the OSI architecture) and SDL, LOTOS is an official FDT for use in standardization. In contrast with the other standard notations, which are both based on the finite state machine paradigm, LOTOS adopts a peculiar modeling approach, namely, process algebra.

Although initially conceived for information processing systems modeled after the OSI reference model, LOTOS has been widely applied to specification and validation of sequential, concurrent, and distributed systems in different scientific domains. Thanks to the sound theoretical basis of the language, LOTOS syntax has been integrated with temporal and stochastic features, which have widened its application field to the performance analysis of dynamic systems.

Feedback coming from the application of LOTOS to system design in industrial environments has suggested several improvements: more friendly definition of data types and operators, support of temporal features, modularity, abstraction, testing, and so on. These additional capabilities have been merged into an enhanced version of the standard (E-LOTOS [39]), which has recently been proposed as a draft by ISO/IEC's (International Electro-technical Commission) competent working group.

SYNTAX

LOTOS consists of a language for describing the behavior of processes (initially based on CCS [40]) and an algebraic data type specification language called ACT ONE, which is conceptually equivalent to SDL's specification language. As anticipated, LOTOS is primarily based on the theory of process algebra (PA). PAs are abstract languages for formal specification which, thanks to powerful logical operators, model distributed systems and concurrent communicating processes in an effective and synthetic way [41].

In LOTOS, system specification consists of two parts (Fig. 6):

- Declaration, where the system behavior is characterized as an interleaved or synchronized composition of subsystems.
- Definition, where the behavior of each component of the system is described by listing the possible transitions and the ordered sequence of actions (behaviors) each transition gives rise to.

The specification of components can also be done by describing their structure as a composition of simpler constitutive processes, similar to system declaration. This approach is analogous to SDL distinction between structural and functional issues: the declarative sections illustrate the deployment of the basic components and their interconnections, whereas the definition of the behavior is equivalent to tracing an ECFSM in a textual way.

A process is characterized by its formal interface (*gates*, which can be explicitly selected at process instantiation) and parameters, which account for the data it manipulates, whose types and operations are usually defined in an external library included in the specification file. The process behavior P is a deterministic sequence of actions or a random/guided choice between competitive behaviors ([$bool\_expr_1$] P_1 [ ] [$bool\_expr_2$] P_2), which are often described with the instantiation of a different or the same process (thus implementing recursion). An action is a communication activity; assignments on parameters are executed by recursive instantiation of the same process with a different numerical value ($P(x) := a$; $P(x+1)$).

Processes can be combined in different ways: they can be interleaved ($P|||Q$) and evolve in a completely independent manner; they can be partially synchronized on specified gates ($P|[gate_1, gate_2,\ldots]|Q$) on which input/output action must be matched; or they can be completely synchronized ($P||Q$), sharing the whole of their gates. A process can also terminate producing a value that enables another process to start. This relation is called "enabling" ($P \gg Q$) and can be exploited in modeling procedure-call actions. A "disabling" construct instead ($P[>Q$) renders interruption and can account for competitive behaviors or high priority message handling.

A graphical notation has also been proposed [42], although no significant tool support has been available so far. The interest in this matter is high [43], because a friendlier graphical interface could widen the acceptance of LOTOS in the industrial world.

COMMUNICATION

LOTOS communication is basically synchronous. Two or more processes that are connected by some gates must perform coupled actions, otherwise a deadlock occurs: send/receive actions are synchronized on value passing (a!x; P $|[a]|$ $a?y{:}Int;$ Q, where y takes the value of x); value-matching ($a!x;$ P $|[a]|$ $a!y;$ Q, which is executable only if x and y have the same value); or value-negotiation ($a?x;$ P $|[a]|$ $a?y;$ Q, where the two parallel behaviors synchronize on the receiving actions and a random value z is generated and copied in both parameters x and y). LOTOS synchronous communication is similar to the hand-shaking of two processes connected by a

P[a,b] || Q[b,c]	Parallel composition of processes
P[a,b,c] |[a,b,c]| Q[a,b,c]	Synchronized composition
P[a,b] |[a]| Q[b,c]	Partially synchronized composition
P [ ] Q [ ] R	Competitive choice among behaviors
[expr] -> P [ ]	Enabling condition (usually combined with [ ] with other alternative behaviors)
P[a,b](formal_pars):= …; P[b,a](expr{pars})	Process declaration with re-labeling and data manipulation
a; P	Immediate action
tau; P	Immediate internal action
a!x; P	Immediate action with sending
a?x:int; P	Immediate action with receiving
a!x; P |[a]|a?y:int; Q	Synchronization through value passing (par y assumes value x)
a!x; P |[a]| a!y; Q	Synchronization through value matching (executable if x=y)
a?x; P |[a]| a?y; Q	Synchronization through value negotiation (x and y take the same randomly generated value)
hide a, … in P	Abstraction

Table 3. *Basic syntax of LOTOS.*

rendezvous channel in Spin. In practice, in LOTOS more than two processes can be synchronized within a single communication action, given that the synchronizing actions are unambiguously characterized by a compatible format, that is, same channel name, same values or type of exchanged values.

The asynchronous method of communication implies the insertion between communicating instances of message queues that implement a FIFO algorithm. The complexity of the specification is the main drawback of this formal method, balanced by its generality and implementation independence.

Another useful concept is abstraction of internal behavior: LOTOS provides an operator (*hide*) to conceal the details of the specification. In this way, internal actions lose their identity but maintain their operative functions, thus enabling more efficient state-space generation, simplification of the model, and verification of the correctness of the observable behavior, that is, through the equivalence with the service specification. Table 3 summarizes the basic syntax of LOTOS.

Example of Specification

Figure 7 shows the specification of the ARQ GBN protocol. The service rendered by the protocol to the higher layers is the ordered delivering of data packets and is formally specified in Fig. 8.

The protocol specification consists of the parallel composition of the processes *Transmitter* and *Receiver*, which are synchronized on gates *Pdu* and *Ack*, representing successful signal reception, and *Pdu_l*, *Ack_l*, representing corrupted messages. When the sender is not in a stalling condition (i.e., when the transmission window *ns-nr* equals 2), it is stimulated by the (spontaneous) handling of *Data_req* service primitives. Conversely, the receiver issues a *Data_ind* primitive for each correctly and orderly received PDU.

The peer entities communicate according to a LOTOS-specific synchronous model. Instead of specifying an asynchronous and unreliable channel between the processes *Transmitter* and *Receiver*, the unreliability of the transmission is described as a non-deterministic choice between the two executable actions.

Every message can be received either correctly (actions *Pdu* and *Ack*) or can be lost (actions *Pdu_l* and *Ack_l*). With respect to PDU transmission, *Pdu!ns; ReceiveAck [ ] Pdu_l;…* and *Pdu!x:Int;…[ ] Pdu_l;…* are the choices for the processes *Transmitter* and *Receiver*, respectively.

Depending on the synchronizing actions and the current value of internal parameters (*ns*, *nr,* and *ns*2), the behavior of the process *Transmitter* is described by the process *ReceiveAck*, which specifies the successful or failed reception of the acknowledgment message; the process *Update*, which decreases the window size by updating the counter *nr*; and the process *Repeat*, which exploits the integer variable *ns*2 to keep count of retransmitted PDUs. The process *Receiver* has a simpler behavior, as it enters the *Send_Ack* process when *Pdu*s are received, and it loops in the case of synchronization with *Pdu_l* action.

Since standard LOTOS does not support the definition of constant values, the maximum sequence number *N* and the maximum window size *W* are replaced by their numerical values (4 and 2, respectively).

Finally, it shall be observed that all signals are hidden (with the *hide* operator), with the exception of the service primitive *Data_Ind*. This modeling choice makes it possible to isolate the observable behavior of the protocol layer, which shrinks to the activation, within the receiver entity, of gate *Data_Ind* with its associated numerical values. With the aid of LOTOS-based tools, it is possible to verify the conformance of the protocol specification with the service specification, which is illustrated in Fig. 8 as the ordered delivery of PDUs with progressive sequence numbers (*Data_Ind!n*, with *n* cycling from 0 to 3).

Tool Support

Due to its textual format, its strong abstraction, and the original synchronous evolution of concurrent behaviors without time support, LOTOS requires greater effort than SDL to be profitably employed in industrial environments. Nevertheless, some software packages for system design have been developed. The most complete is the Caesar/Aldebaran Development Package (CADP) [44]. Its most interesting features are parsing and simulation of LOTOS specification, generation of graphical FSM from textual specifications for deeper analysis, model minimization (for the analysis of complex systems), model comparison based on the equivalence of observable behaviors (bisimulation), model-checking with efficient algorithms, verification of logical temporal formulas, test case generation, and compilation of abstract data type libraries. The package includes a large set of examples, which encompass a decade of case studies developed within LOTOS research projects.

Some effort has been devoted to support specification and animation in graphical LOTOS, but so far no mature product has been developed.

For simulation and performance analysis purposes, some promising tools are available (PEPA [45] and TIPPtool [46]). Both of these tools adopt a stochastic extension of LOTOS notation, namely, the timed action (a,), which occurs after an exponentially distributed temporal interval of mean value 1/ . TIPPtool supports also the probabilistic branching, expressed as the choice between alternative behaviors weighed with a

discrete probability distribution (*process P*: = $[p_1]\ P_1[\ ]\ [p_2]\ P_2 \ldots$). In addition, TIPPtool provides a default integer data type for parameters and communication through single-value passing, which makes the specification task easier, although the capability of structured data and message handling provided in standard LOTOS is not yet available. The operational model of such stochastic PA is easily translated into a discrete or continuous-time Markov chain (MC), the reference theoretical model for system performance analysis.

In the formal framework provided by process algebras, systems can be modeled in their temporal and dynamical behavior, verified for absence of deadlocks and loops, reduced with the minimization algorithms available for process algebras (thus extending the level of complexity that can be handled), and finally evaluated in their performance. Performance measures, which are automatically evaluated, are expressed as probability of a state or a set of states, throughput of timed actions, and mean values of process parameters.

Examples of Applications

LOTOS has been used in the scope of some projects intended to prove the validity of the formal approach in development (specification, verification, and testing) and implementation (automatic code generation) of real communication systems. The *Proceedings* of the PSVT (Protocol Specification, Validation and Testing) symposia, starting from [47], and of FORTE (FORmal Description TEchniques for Distributed Systems and Communication Protocols) are a good source of references. Interesting recent applications are validation and testing of GPRS systems [48], specification [49] and performance analysis [50] of multimedia networks, and validation of security protocols [51], a key issue in computer networks and an application domain where FMs are gaining increasing relevance.

```
specification go_back_N [Data_Ind]: noexit
  library NATURAL,BOOLEAN endlib
behaviour
hide Data_Req,Pdu,Pdu_1,Ack,Ack_]in
    ( Transmitter [Data_Req,Pdu,Pdu_1,Ack,Ack_1](0,0,0)
                  |[Pdu,Pdu_1,Ack,Ack_1]|
      Receiver    [Data_Ind,Pdu,Pdu_1,Ack,Ack_1](0)   )
where
  process Transmitter [Req,Pdu,Pdu_1,Ack,Ack_1] (ns,nr,ns2:Nat) : noexit :=
  [((ns-nr)mod 4) lt 1]-> Req!ns;(Pdu!ns; Receive_Ack[...]((ns+1)mod 4,...)[]
                                  Pdu_1; Transmitter[...]((ns+1)mod 4,...) )[]
  [((ns-nr)mod 4) eq 1]->Req!ns;(Pdu!ns; Receive_Ack[...]((ns+1)mod 4,...) []
                                  Pdu_1; Repeat [...]((ns+1)mod 4,...) ) []
  [((ns-nr)mod 4) eq 2]->Repeat [...](ns,nr,ns2)
  where
    process Receive_Ack [Req, Pdu,Pdu_1,Ack,Ack_1](ns,nr,ns2:Nat) : noexit:=
    Ack?y:Nat; Update[...](ns,nr,ns2,y) []
    Ack_1; Transmitter [...](ns,nr,ns)
    where
      process Update[Req,Pdu,Pdu_1,Ack,Ack_1](ns,nr,ns2,y:Nat) : noexit:=
        [y eq nr]-> Transmitter [...](ns,nr,nr) []
        [y ne nr]-> Update [...](ns,(nr+1)mod 4,ns2y)
      endproc
    endproc
    process Repeat [Req,Pdu,Pdu_1,Ack,Ack_1](ns,nr,x: Nat) : noexit:=
      [((x+1)mod 4)eq ns]->    (Pdu!x; Receive,Ack[...](ns,nr,nr) []
                                Pdu_1; Repeat[...](ns,nr,nr)  ) []
      [((x+1)mod 4)ne ns]->    (Pdu!x; Receive,Ack[...](ns,nr,(x+1)mod4) []
                                Pdu_1; Repeat[...](ns,nr,(x+1)mod 4) )
    endproc
  endproc

  process Receiver [Ind, Pdu,Pdu_1,Ack,Ack_1](nr:Nat) : noexit :=
    Pdu?ns:Nat; (   [ns eq nr]-> Ind!ns; Send_Ack[...] ((nr+1) mod 4) []
                    [ns  ne nr]-> Send_Ack[...] (nr) ) []
    Pdu_1; Receiver [...] (nr)
  where
    process Send_Ack[...] (nr:Nat) : no exit :=
      Ack!nr; Receiver [...] (nr) []
      Ack_1; Receiver [...] (nr)
    endproc
  endproc
endspec
```

FIGURE 7. *LOTOS specification of the go-back-N protocol.*

Critical Evaluation

LOTOS's compositional approach to system specification, high abstraction, and support of model reduction with respect to observational equivalence are extremely useful when coping with large communication systems. As proof, in [15] SDL specification (meant for implementation) and, with some more effort, SDL-specific asynchronous communication have been translated into LOTOS to make thorough model-checking possible. This experience hints at the possibility of going through LOTOS specifications for complete validation of complex prototypes, at least according to the validation capabilities currently offered by FDT.

As for performance analysis, the formal approach based on stochastic process algebras effectively supports theoretical investigations, because it provides an effective notation to model queuing systems with a high level of abstraction and powerful tools to automatically solve the correspondent performance models.

The main drawback of this formalism is the distance between the specification style of the technical documents, which adopt a FSM-like language in describing the system behavior, and LOTOS's high-level approach, based on PA formalism, an operational model that is far less implementation-oriented and which requires extreme thoroughness in dealing with synchronization aspects. Another rather disagreeable feature is the cumbersome declaration of data types within the system specification. Type definitions can be collected in libraries that shall be included in the specification body in order to be comprised of model-checking and validation, but this shortcut has not been sufficient to grant LOTOS a predominant position among industrial users of formal methods

```
specification go_back_N [Data_Ind] : no exit
    library
        NATURAL, BOOLEAN
    endlib

behaviour
            ARQ[Data_Ind](0, 4)
where
    process ARQ [Ind] (n, N : Nat) : noexit :=
        Ind!n; ARQ[Ind]((n+1)mod N,N)
    endproc
endspec
```

FIGURE 8. *Specification of the service offered by the GBN protocol.*

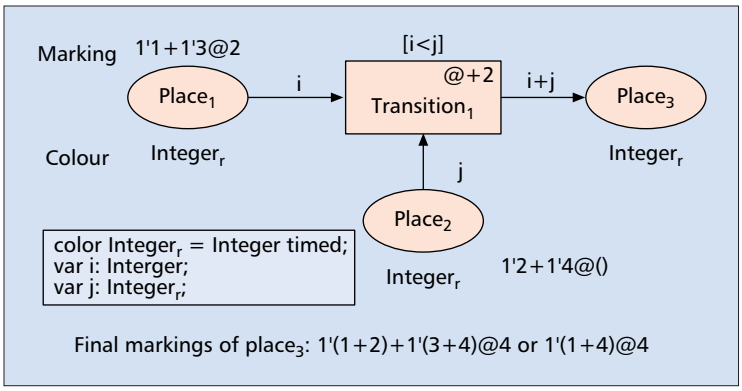

FIGURE 9. *CPN graph with a timed transition and a guard condition.*

for communication protocols, as shown by the scarcity of LOTOS commercial tools for system design.

PETRI NETS

Petri Nets (PN) are a formal modeling technique for the description of concurrent and distributed system behavior. Since their introduction in the 1960s they have impressively evolved. Currently, several versions of this graphical modeling language exist, which find widespread application in specification, verification, and performance analysis of distributed parallel systems, communication protocols included.

SYNTAX

A traditional PN is a collection of *places* (represented by ellipses or circles), *transitions* (straight bars), *edges* (oriented arcs), and *tokens*, which are the marker of a place. Each edge connects a place to a transition or vice versa. Places can be considered as conditions on the system control states and transitions as actions. When a condition is verified, that is, when one or more tokens fill all input places, a transition can fire and carry the enabling token to the places of destination, which now become active. When more than one transition are simultaneously enabled (non-deterministic behavior), in the validation phase all the possible execution sequences are explored, whereas during simulation the execution step can be chosen randomly or interactively.

The *initial marking* of a PN determines the execution sequence: if for any initial assignment of tokens the behavior of the net does not stall, the net is called "life." Otherwise, through model-checking, it is possible to find the "safe" markings that do not lead to blocking conditions.

There are many variants of the generic PN mentioned above, from the simplest one (boolean/integer token PN) to the high-level token PN, where tokens are structured values that, by traversing the net, enable an effective communication between parts of the system. For each class there is a set of *ad hoc* software tools. The relation between high-level and ordinary PN is analogous to the relation between high-level programming languages and assembler code: the modeling capabilities should be the same, but the superior abstraction of the high-level specification simplifies the design phase. From a model-checking perspective, instead, high-level nets are equivalent to huge transition systems, difficult to cope with without the aid of partial analysis or reduction techniques.

A powerful dialect of high-level PN are the Colored Petri Nets (CPN) [52], a specification formalism that employs PN features to model parallel behavior and high-level programming languages to define data types (namely, the "colors" of the tokens), functions, and computation on data. Beside structured data values, tokens carry, as a timestamp, the deterministic or randomly distributed temporal length of the transition they enable. A time consuming transition, then, increments the delay associated to the *timed* token it processes, which can be consumed by the following transitions only when such delay has elapsed.

COMMUNICATION

Communication is represented by token exchange between different parts of the net, which embody separate communication entities. Token "reception" can be conditioned by explicit guard conditions appended on the arcs entering the transition and on the transition itself. Based on the current value of the token, which is copied in a locally bound variable, a transition is blocked or enabled and different values for the output tokens can be delivered to the final states.

The graph depicted in Fig. 9 contains all basic concepts of CPN syntax and semantic. The simple transition $Transition_1$ fires when the values i and j (extracted from the tokens that mark starting states $Place_1$ and $Place_2$) satisfy the guard condition $i < j$. Depending on which tokens are consumed first from input places, one or two firings of the transition can take place. The final marking of $Place_3$ can be, respectively, one token with value 5 or two tokens with values 3 and 7. With respect to temporal behavior, the timed tokens in $Place_1$ are active after two temporal units have elapsed. The one or two transitions they give rise to increment the token delays, so that the timestamp of the output tokens is 4.

EXAMPLE OF SPECIFICATION

Figure 10 shows an example of a hierarchically structured CPN specification, representing a sender and a receiver communication on an unreliable channel through the ARQ GBN protocol.

The two communicating processes are described at the same hierarchical level in the same graph. They are identified as the collection of the states, respectively, above and below the transitions *CH1* and *CH2*. More precisely, the sender entity comprises the states *Sender*, *seq_num*, *window,* and *Acks*, whereas the receiver is embodied by the state *seq_num*. It can be observed that there are a couple of states with the same identifiers (*buffer*, *Packets*). They are distinguished states with similar function, that act as interfaces among protocol entities and channels.

The unidirectional channels *CH1* and *CH2* are two instances of the same substitution transition *Channel*, whose behavior is specified at a lower hierarchical level, in a separate page. The unreliability of the generic channel behavior is described as a substitution transition in a separate diagram. Within the execution of the transition *loss*, the global variable *loss* extracts from (and gives back to) state *Loss* one of the two tokens of the marking. If the value that has been copied is 0, a token (of color *Packet*) is forward from place *Data_in* to place *Data_out*. Otherwise, the token is consumed. The matching between the states (*Packets* and *buffer*) connected to the channels and the input and output ports appearing in the sub-page is determined by the flow and the colors of the tokens. The deterministic delay related to packet delivery is associated with the transition *loss*.

Declarations of colors (types) and variables are collected in a declaration area, which is local to the main graph representing the GBN ARQ protocol.

Tool Support

Many freeware and commercial tools to handle modeling, animation, and validation of low and high-level PNs have been developed. In telecommunication topics, Design/CPN [53] is one of the most elaborate and successful. It supports hierarchical description by means of transition refinements, model-checking for static and dynamic properties, formal analysis methods (places' and transitions' invariance verification) and simulation (automatic as well as interactive with feedback information and several debugging options). Validation of timed graphs is not yet supported; this compels developers to turn to the untimed version to apply model-checking routines. For the analysis of complex models, a highly efficient simulation engine has recently been developed, which dramatically speeds up automatic simulation runs. An announced library supporting MSC tracing will further increase the pertinence of the tool to the field of communication protocols and the equivalence with other design approaches.

The software package and reference guides are distributed free of charge to all types of users, including commercial companies; in fact, the tool has been exploited in several industrial projects.

Examples of Applications

Typical applications of CPN are in the field of communication protocols, telecommunication networks, and software engineering. The proceedings of the annual CPN Workshops comprise a selection of real-life case studies.

We have exploited this approach to the modeling and performance analysis of the GPRS radio interface, giving a detailed representation of the physical layer and thus proving the applicability of this description method to any layer of the OSI architectural model [27]. A recent example of the application of CPNs to software engineering is given in [54], where the software architecture of a mobile phone family has been modeled in CPN and analyzed in both time and space (buffer size requirements) performance. The complexity of the prototype has hindered full validation and required abstraction of implementation details to be carried out in an automatic way.

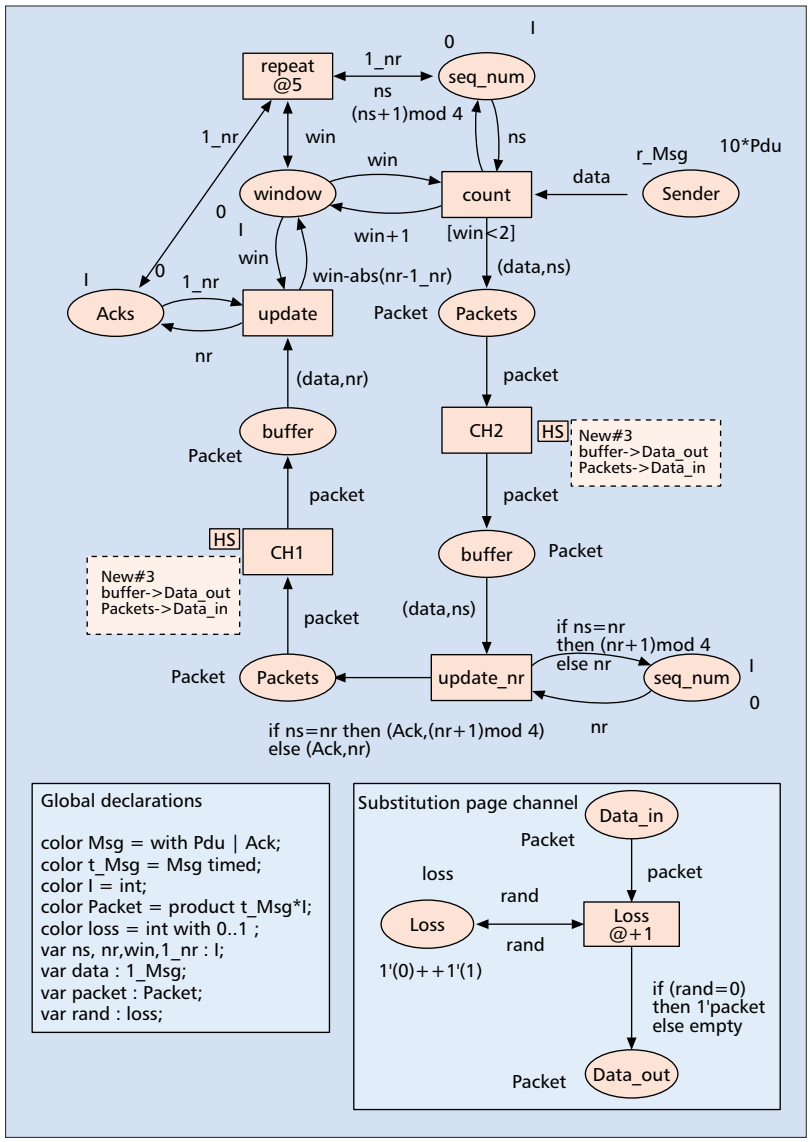

Figure 10. *Example of a hierarchically structured Colored Petri Net representing the GBN ARQ protocol.*

Critical Evaluation

Despite a modeling philosophy that is not easy to be acquired, Petri Nets, thanks to their appealing graphical notation that is similar to the more familiar FSM, are a powerful modeling language to describe and investigate communicating and resource-sharing processes. High-level PN, and colored PNs in particular, by combining the rigorousness of the basic formalism and the effectiveness of the programming language for computational details and communication aspects, enable the creation of compact but accurate system models taking into account temporal and stochastic issues.

Despite the richness of the graphical notation (which is usually balanced by lower analysis capabilities), the pertinence of this formal method to validation and performance analysis is high thanks to the availability of powerful tools.

Similar to other graphical FDTs, embedding user-defined functions into the specification is an interesting possibility to avoid heavy graphical descriptions of deterministic calculations. In Design/CPN this requires some effort to gain an adequate knowledge of the functional language (Standard ML) supported by the tool, which is not a typical implementation language. Thus, user-defined routines cannot be immediately reused in code production.

UPPAAL

UPPAAL is a toolbox for modeling, simulation, and verification of Timed Automata (TA), which has been developed jointly by the University of Uppsala and Aalborg [55]. Being devoted to concurrent non-deterministic systems with limited internal control structures, it is well suited to the research area of communication protocols, where aspects related to temporal constraints are often crucial. Thanks to the commitment devoted to simplifying as much as possible the tool usage, UPPAAL has been provided with a graphical, platform-independent interface, being written in Java.

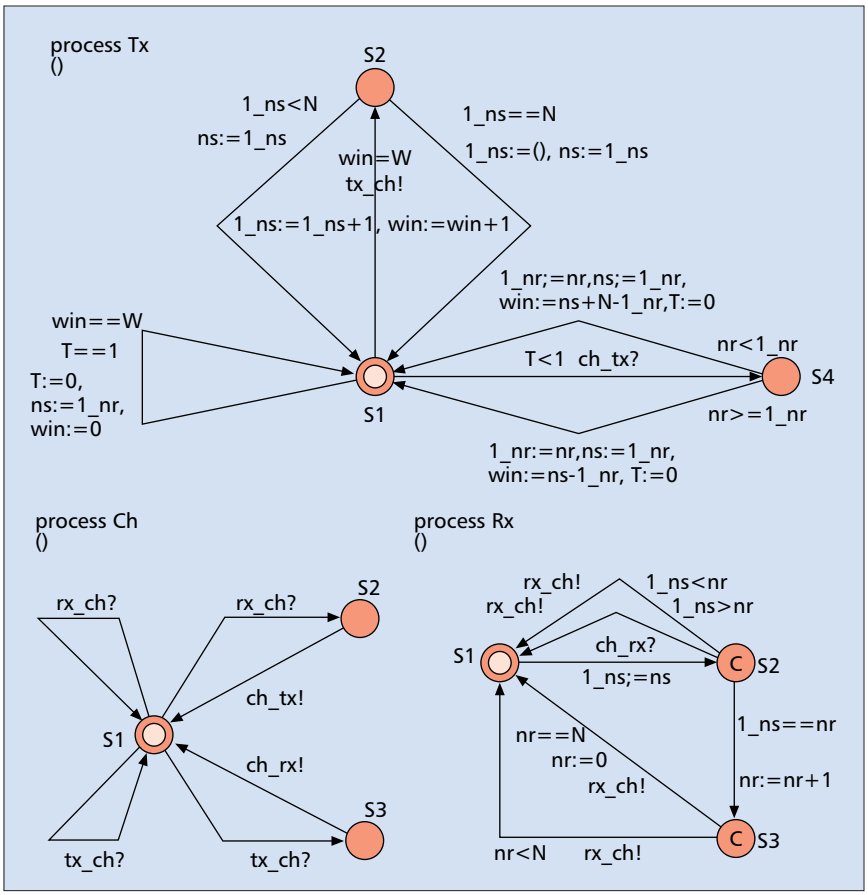

FIGURE 11. *Time Automata specification of the GBN ARQ protocol.*

SYNTAX

UPPAAL specification language is based on the theory of Timed Automata [56], which can be interpreted as an extension of FSMs by means of timers. Continuous-timed systems possess infinite execution sequences due to the dense nature of time. The functional equivalent of so called temporal regions makes it possible to convert the unbounded behavior into a finite state space, thus enabling a quantitative approach to temporal properties. The system model is formed by a finite number of processes (instances of automata templates, which can depend on formal parameters) in parallel execution, each characterized as a communicating FSM that is able to manage global and local variables and temporal counters (timers).

Each state of the process is marked by properties on variables and/or timers, which must hold as long as the state is active, otherwise a dynamic error (deadlock or, more precisely, *timelock*) is detected. Transitions are guarded by conditions on timers or data values which, once satisfied, enable the execution of the actions that are appended to each transition: signal exchange, assignment on data variables, or set/reset of timers. The instant of execution is selected in a non-deterministic manner within the period of validity of the guard condition. The numerical analysis addresses all the execution sequences this random start can give rise to, hence the results represent only hard constraints on the execution timing. When tasks performed in consecutive transitions cannot be interleaved with transitions in other automata, the intermediate states are declared *committed* and treated as an atomic action.

COMMUNICATION

Communication between processes is implemented as an exchange of unstructured control signals on bidirectional synchronous channels. A sending action within an enabled transition is executable only if there is in the system a hanging receiving action to mach it, similar to Spin synchronous communication on zero-length channels. The transmission of structured messages can rely on global variables, once assured that no undesirable collision occurs between writing and reading tasks.

Message exchange can be time-consuming, if it is associated with the reset of a timer or interleaved with the contemporary execution of other time-consuming actions within the system. On the contrary, when the communication must be executed instantaneously and with higher priority, the correspondent channel is declared *urgent*.

EXAMPLE OF SPECIFICATION

Many of the modeling concepts previously described are exploited in the specification of the GBN protocol (Fig. 11). The system specification is given by three instances of process templates, *Tx*, *Rx,* and *Ch*, which use no formal parameters. Process *Tx* (*Rx*) communicates with the unreliable channel through signal *tx_ch* (*rx_ch*), which is directed to the channel, and signal *ch_tx* (*ch_rx*), which is received from the channel. Since no numerical value can be transferred through communication actions, the global variables *ns* and *nr* are used to convey the sequence numbers associated with the exchanged messages.

To avoid collision between reading and writing of such global data, *Tx* and *Rx* exploit respectively the local variables *l_nr* and *l_ns* to store temporarily the sequence numbers associated with signal reception.

In state *s1*, when the transmission window is full (*win==W*) and the clock *T* has reached its default value 1 (*T==1*), the sender entity resets the timer and the transmission window ((*T:=0*; *win:=0*) and starts the retransmission of pending packets from sequence number *l_nr*. Before the timeout condition (*T<1*), *Tx* can receive a signal from the peer entity (*ch_tx?*), which triggers the updating of variables *l_nr* and *ns*.

Process *Rx* behaves without temporal references: when a signal from the peer entity is received (*ch_rx?*), a signal is sent to the channel (*rx_ch!*). The updating of the expected sequence number *nr* is based on the comparison between *nr* and the received sequence number (*l_ns:=ns*). To avoid possible collision due to interleaved executions, this critical task shall be performed as an atomic action, labeling the intermediate states as committed.

The unreliability of the channel is rendered as the alternative non-deterministic choice between correct signal forwarding (with respect to the transmission from *Tx* to *Rx*, transition to state *s3 tx_ch?* and back to *s1 ch_rx!*) and silent signal consumption (transition *tx_rx?*, remaining in the same state *s1*).

Tool Support

UPPAAL supports graphical system specification, simulation through animation of the prototypes, and validation of user-defined temporal properties, formulated as temporal logic formulas. Alternatively, validation can be executed by introducing in the system specification observer or watchdog processes, which are synchronized on undesired behaviors. When the latter occur, a *timelock* is detected and a guided simulation can be performed to resolve inconsistencies in the specification.

Simulation helps in the initial design phase and in random verification when no thorough analysis is possible due to memory constraints.

Examples of Application

As some examples in the literature suggest [57, 58], UPPAAL can be used to determine under what rigid bounds a temporal property is satisfied, that is, the threshold value of the timer controlling a retransmission protocol or the maximum sustainable jitter in the input signal of a decision block. Performance or probabilistic behaviors are banished from this context. Quantitative results assume the form of a minimum guaranteed time interval during which the requirements on the system still hold.

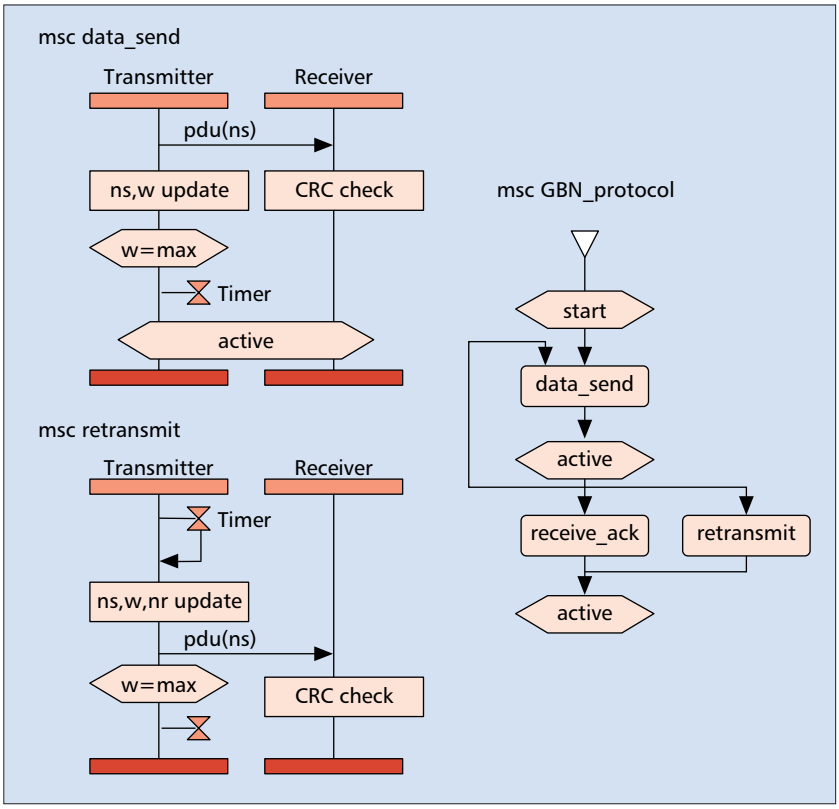

Figure 12. *MSC of process interaction in the GBN ARQ protocol.*

Critical Evaluation

UPPAAL can be useful in investigating the impact of timer values on real-time communication protocols. In some cases, the high modularity of protocol behavior makes it possible to isolate the temporal aspects and evaluate the correct values almost by hand. On the contrary, when dealing with several independent timer instances ruling data transfer within the same communication session, automatic validation with UPPAAL can be extremely useful in fixing the range of timer values that satisfies temporal requirements and guarantees functional correctness.

As for performance testing, research on stochastic automata and specification and analysis of soft-time constrained systems is fervent and tools are expected to follow.

Other Notations in Short

MSC

Message Sequence Chart is a graphical language for describing the interactions between components of real-time systems, in particular telecommunication switching systems. MSCs are a standard ITU-T notation for the specification of requirements, conventional behavior, simulation output, and test cases [59].

MSC syntax provides notations for message exchange, process activation, timer handling, generic tasks, and conditions. Conditions can be used to associate partial execution sequences in a unique MSC representing the entire system behavior.

Object-oriented features are MSC types, which are instantiated by declaring the actual message names, and virtual MSC, which can be redefined to particularly suit operating conditions. Figure 12 collects simplified MSCs of data transmission procedures and the MSC roadmap of the GBN ARQ protocol.

Some MSC editors exist; for textual to graphical format conversion, LaTeX macro packages are also available.

UML

UML (Unified Modeling Language) is the most important graphical notation for object-oriented software systems, whose standardization is coordinated by the Object Management Group (OMG) [60]. UML specifications are collections of diagrams. Each diagram supports a specific phase of the development process by capturing a particular abstraction of the system behavior (analysis of the requirements, architectural and functional design, HW/SW partition, etc.). Some features of UML have been included in SDL2000 to enrich SDL software architecture description capabilities.

The following diagrams are useful for protocol specification:

- Sequence diagram, similar to standard MSC.
- Class diagram, a graph of abstract elements (classes) connected by static relationships: hereditary, aggregation, association, and so on.
- Statechart diagram, a graph that illustrates the dynamic behavior of classes representing event-driven processes. UML notation is derived from statecharts, a powerful FDT that extends the finite state machine models with parallel decomposition of states and transversal communication between different hierarchical levels of the specification [61].
- Object diagram, which collects the class instances (objects) implementing the actual system.

In UML, processes are modeled with classes, local variables with private or public attributes, internal procedures with private methods, input signals with public methods, messages, and information elements with methods' parameters.

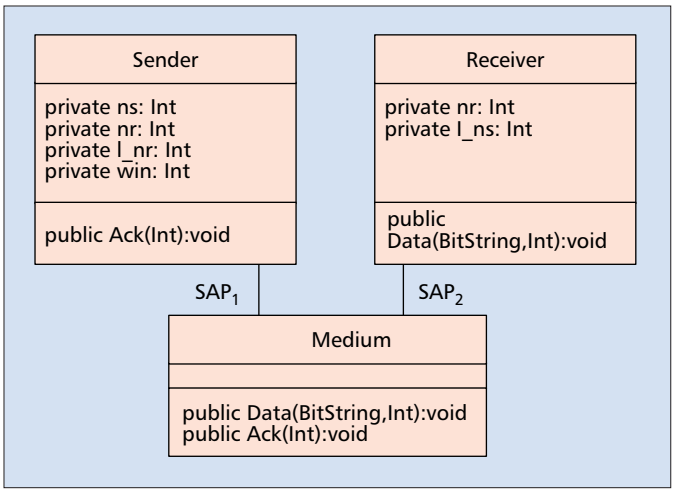

FIGURE 13. *Class diagram of the GBN ARQ protocol.*

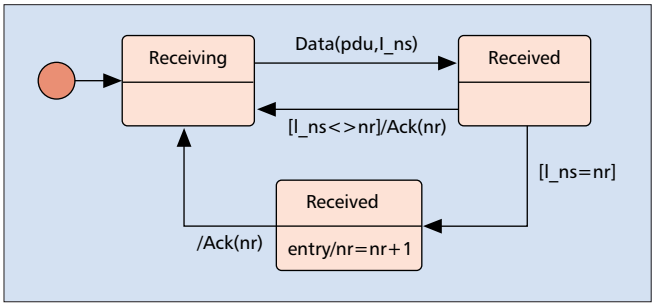

FIGURE 14. *Simplified Statechart diagram of the Receiver class.*

Classes communicate through the invocation of other classes' methods, which triggers an internal reaction and, possibly, a response. Figure 13 presents a class diagram and Fig. 14 presents a statechart diagram of a simplified version of the GBN ARQ protocol. In order to support syntactic and semantic verification, UML includes OCL (Object Constraint Language), a formal language to specify invariant conditions and well-formedness rules on the system model. As for the extension of the modeling language to realtime systems (Real-Time UML), a sound mathematical foundation is still under investigation.

There are dozens of commercial and freeware tools for UML, ranging from simple graphical editors to complete CASE packages.

CONCLUSIONS

With the steady increase in the complexity and diffusion of information processing systems, software vulnerability to errors becomes more and more crucial. Traditionally, error checking is carried out with simulation, testing, and code inspection; if a fault is detected, the entire design process may have to be reviewed, thus increasing costs and lengthening time to market.

Formal methods offer a complementary approach to the reliability problem by enforcing the system requirements on a mathematical model, whose correctness can be automatically proved. Formal methods can support all relevant phases of software development, from specification through validation, test generation, and performance testing.

This article provides a tutorial description of formal methods and tools for design and implementation of communication protocols. It introduces the main formalisms by explaining the basic syntax and exemplifying it with the help of a common case study, the specification of the go-back-N ARQ protocol. Based on the available tool support and on the applications to real-life case-studies, some critical conclusions are formulated on the utility of the various FDTs. A synthesis of the evaluations on the various techniques is presented in Table 4 and is here briefly summarized.

Among the notations devoted to system specification, SDL has achieved widespread success for its friendly graphical notation, its conformance to the idiom adopted by standardization institutes, and its support for other popular notations such as ASN.1, MSC, and TTCN. Spin outperforms SDL-based commercial tools because it can check, besides static and dynamic properties, the consistency of the system's temporal behavior in all execution sequences. This makes Spin the most successful freeware tool for large-system validation by both academic and industrial users. Although formally equivalent to the other FMs and similarly based on FSM, Estelle is worth mentioning basically for historical reasons, as it has evolved poorly in usability and tool support.

Another interesting FM is the process algebra LOTOS, which can handle very complex systems by exploiting the notions of abstraction from internal details (i.e., the hide operator) and reduction of state space (observational equivalence). In addition, because LOTOS-timed systems are semantically close to Markovian processes, a quantitative analysis of performance can be easily obtained starting from LOTOS synthetic specifications of communication systems. However, its complex and strict syntax limits widespread diffusion in real-life industrial projects and confines most of the research activity to academia. Petri nets, on the contrary, adopt a friendly (although unconventional) graphical notation to address several aspects of the development process: formal modeling, model-checking, and efficient simulation of high-level prototypes. Timed automata analyze temporal properties of non-deterministic timed systems, verifying the correctness of the specification and calculating temporal hard bounds on the system performance. This kind of check is indispensable for safety-critical real-time systems, in whose development this FM finds a most appropriate application.

Finally, it is observed that the evaluation of each method shall take into account, besides theoretical concerns, the time and resources the user requires to become familiar with the method; the learning curve is strongly connected with the maturity of languages and tools. However, the "cost" of the learning phase can be moderated when the acquired know-how can be reused in all future projects. Therefore, once the critical steps of protocol development have been selected to introduce a formal approach, the choice of the most suitable technique implies a tradeoff between abstraction, large system handling capabilities, friendliness of the notation, possibility of knowledge reuse, and cost of the tool. There are several different approaches to modeling system behavior (synchronization, communication, data manipulation, real-time issues) which often do not offer a general-purpose solution. However, in the several steps of the development of a telecommunication system, it is possible to identify the formal techniques that best suit each particular phase. Some commercial packages gather many notations to cope with the entire development process. This integrated solution, not yet based on a common semantic of the different formalisms, offers a doubtfully optimal, yet surely expensive solution. Integration of traditional structural development and automatic FM-based tools is seen as a promising alternative, and research is working toward a unified mathematical framework for system specification and enhanced CASE tool support [62–64].

	SDL	Spin	Estelle	LOTOS	Petri nets	Timed automata
Pertinence to protocols	Very high	Very high	Very high	Very high	Very high	Very high
Modeling capabilities	Very good, relying on distinction between structure and functions	Good (very good for single protocol layer)	Good, requiring coherence at different heirarchical levels	Very good, relying on abstraction	Low for 1/2-level PN; very good for high-level PN	Limited in PDU handling
Validation capabilities	Good, limited by model size	Very good, limited by model size but extended to LTL	Hindered by the usage of Pascal	Very good	Good for low-level PN; limited by system size for high-level PNs	Very good for hard time-constrained systems
Simulation capabilities	Good	Good, producing MSC	Good	Discrete	Very good	Good, with random or guided execution of the automata
Performance analysis	Discrete, by means of analysis of queuing SDL systems	Not supported	By means of simulation of the implementations	Good, e.g., linking PA and Markov chains	Very good	Quantitative measures are upper bounds
Rapid-prototyping	Possible	Compilers are under development	Possible (main feature)	Possible	Supported by some tools	Not supported
Tools availability	Commercial	Freeware (Spin)	Commercial and freeware	Freeware	Several ad hoc tools (commercial and freeware)	Freeware (Uppaal)
Friendliness	High	Medium	Medium	Low	High	High
Industrial applications	Widespread usage for system design	Several, in system verification	Several applications to testing and FM-assisted implementation	Limited	Widespread usage in SW engineering, TLC systems and communications protocols	Several, on simple but tricky real-time problems
Main application domain	General-purpose	Validation of logical properties	Testing and implementation	Validation and performance analysis	General-purpose (particularly simulation)	Validation of hard temporal constraints

Table 4. *Summary of the critical evaluations on formal methods for communication protocols.*

Acknowledgments

The authors would like to acknowledge the anonymous reviewers for their insightful comments and valuable suggestions to improve the article.

References

[1] J. M. Wing, "A Specifier's Introduction to Formal Methods," *IEEE Computer*, vol. 23, no. 9, Sept. 1990, pp. 8–24.
[2] archive.comlab.ox.ac.uk/formal-methods.html
[3] A. Hall, "Seven Myths of Formal Methods," *IEEE Software*, vol. 7, no. 5, Sept. 1990, pp. 11–9.
[4] J. P. Bowen and M. G. Hinchey, "Seven More Myths of Formal Methods," *IEEE Software*, vol. 12, no. 4, July 1995, pp. 34–41.
[5] J. P. Bowen and M. G. Hinchey, "Ten Commandments of Formal Methods," *IEEE Computer*, vol. 28, no. 4, April 1995, pp. 56–63.
[6] E. M. Clarke and R. P. Kurshan, "Computer-Aided Verification," *IEEE Spectrum*, vol. 33, no. 6, 1996, pp. 61–7.
[7] D. Lee and M. Yannakakis, "Principles and Methods for Testing Finite State Machines: A survey," *IEEE Proc.*, vol. 84, no. 8, Aug. 1996, pp. 1090–123.
[8] ITU-T, Recommendation Z.100 (11/99) Specification and Description Language (SDL).
[9] J. Ellsberger, D. Hogrefe, and A. Sarma, *SDL Formal Object Oriented Language for Communication Systems*, Prentice Hall, 1997.
[10] www.telelogic.com/industries/telecoms/casestudies.cfm
[11] www.dcc.ufmg.br/~coelho/jade.html
[12] C. L. Pereira *et al.*, "JADE: An embedded Systems Specification, Code Generation and Optimization Tool," *Proc. 13th Symp. Integrated Circuits and Sys. Design*, 2000, pp. 263–68.
[13] www.informatik.hu-berlin.de/Themen/SITE/5
[14] www.sdl-forum.org/Publications/Proceedings.htm
[15] F. Del Frè, "Analysis and Implementation of Intersystem UMTS-GSM Handover Procedure with Application of Basic Encoding Rules to ASN.1 Messages," master thesis (in Italian), University of Trieste, 2001.
[16] S. Leppänen and M. Luukkainen, "Compositional Verification of a Third-Generation Mobile Communication Protocol," Distributed System Validation and Verification DSVV '2000, Taipei, Taiwan, April 2000.
[17] N. Sidorova and M. Steffen, "Verification of a Wireless ATM Medium-Access Protocol," *Proc. 7th Asia-Pacific Software Engineering Conf.*, 2000, pp. 84–91.
[18] B. Blakovic, S. Dembitz, and P. Knezevic, "Model Checking of Concurrent System with SDL Specification," *10th Mediterranean Electrotechnical Conf.*, vol. 1, 2000, pp. 77–80.
[19] N. Husberg, "Verifying SDL Programs Using Petri Nets," *IEEE Int'l. Conf. Systems, Man, and Cybernetics*, vol. 1, 1998, pp. 208–13.
[20] netlib.bell-labs.com/netlib/spin/index.html
[21] G. Holzmann, *Description and Validation of Computer Protocols*, Prentice Hall, 1992
[22] T. C. Ruys, "Low-Fat Recipes for SPIN," *Lecture Notes in Computer Science*, Springer-Verlag, vol. 1885, Sep. 2000, pp. 287–321.
[23] S. Tripakis *et al.*, "Extending Promela and Spin for Real Time," *Proc. TACAS '96, Lecture Notes in Computer Science,* vol. 1055.

[24] L. Millett and T. Teitelbaum, "Slicing Promela and Its Applications to Protocol Understanding and Analysis," *Proc. 4th SPIN Wksp.*, Paris, France, 1998.
[25] G. J. Holzmann, "Logic Verification of ANSI C Code with SPIN," *Lecture Notes in Computer Science*, Springer Verlag, vol. 1885, Sep. 2000, pp. 131–47.
[26] K. Havelund, "Using Runtime Analysis to Guide Model Checking of Java Programs," *Lecture Notes in Computer Science*, Springer Verlag, vol. 1885, Sept. 2000, pp. 245–64.
[27] L. Deotto, "Formal Methods and Tools for Specification, Validation and Performance Analysis of Communication Protocols of 2nd- and 3rd-Generation Mobile Systems," Ph.D. thesis (in Italian), University of Trieste, 2002.
[28] H. Tuominen, "Embedding a Dialect of SDL in ProMeLa," *Proc. Theoretical and Practical Aspects of SPIN Model Checking 5th and 6th Int'l. SPIN Wksps.*, 1999.
[29] A. Basu *et al.*, " Promela++: A Language for Constructing Correct and Efficient Protocols," *Proc. INFOCOM*, San Francisco, California, Mar.–Apr. 1998.
[30] S. Leue and G. Holzmann, "v-Promela: A Visual, Object-Oriented Language for Spin," *Proc. 2nd IEEE Int'l. Symp. Object-Oriented Real-Time Distributed Computing*, May 1999, Saint Malo, France.
[31] H. El-Gendy and H. Baraka, "Transformation of LOTOS Specifications to Estelle Specifications," *Proc. IEEE Symp. Computers and Communications*, 1997, pp. 215–20.
[32] K. J. Turner ed., *Using Formal Description Techniques: An Introduction to Estelle, LOTOS and SDL*, John Wiley, 1993, online at www.cs.stir.ac.uk/~kjt/research/using-fdts/using-fdts.html
[33] J. Templemore-Finlayson and E. Borcoci, "Simulating Multicast Protocols in Estelle," *Proc. Int'l. Conf. FORTE/PSTV'2000*, Kluwer 2000.
[34] M. A. Fecko *et al.*, "A Success Story of Formal Description Techniques: Estelle Specification and Test Generation for MIL-STD 188-220," *Computer Communications*, vol. 23, no. 12, July 2000, pp. 1196–213.
[35] J. W. Atwood, M. Ghodrat, and D. Tasak, "Using Formal Specification and Observers to Specify and Validate the ATM Signaling Protocols," *Conf. Local Computer Networks*, 1999, pp. 117–20.
[36] C-M. Huang and J-M. Hsu, "An Estelle-Based Probabilistic Partial Timed Protocol Verification System," *Proc. 7th Int'l. Conf. Parallel and Distributed Systems*, 2000, pp. 83–90.
[37] T. Tsang and R. Lai, "Verifying Time-Estelle Specification Using Communicating Time Petri Nets," *Proc.12th Int'l. Conf. Info. Net.*, 1998, pp. 678–83.
[38] T. Bolognesi and E. Brinskma, "Introduction to the ISO Specification Language LOTOS," *Computer Networks and ISDN Systems*, vol. 14, Apr. 1987, pp. 92–100.
[39] ISO/IEC JTC1/SC7, Final Draft International Standard 15437: Information technology Enhancements to LOTOS (E-LOTOS), July 2000.
[40] A. J. R. G. Milner, "A Calculus of Communication Systems," *Lecture Notes in Computer Science '92*, Springer-Verlag, 1980.
[41] R. Milner, *Communication and Concurrency*, Prentice Hall, 1989.
[42] T. Bolognesi and D. Latella, "Techniques for the Formal Definition of the G-LOTOS Syntax," *IEEE Wksp. Visual Languages*, 1989, pp. 43–9.
[43] Z. Yulan, Ye Xinming, and J. Bin, "An Automatic Translation from Textual E-LOTOS into Graphic E-LOTOS," *Proc. WCC-ICCT 2000*, vol. 2, 2000, pp. 1232–5.
[44] H. Garavel *et al.*, "CADP'97 Status, Applications, and Perspectives," *Proc. 2nd COST 247 Int'l. Wksp. Applied Formal Methods in System Design*, Zagreb, Croatia, June 1997.
[45] J. Hillston, *A Compositional Approach to Performance Modelling*, Cambridge Univ. Press, 1996.
[46] H. Hermanns, "Compositional Performance Modelling with the TIPPtool," *Performance Evaluation*, vol. 39, 2000, p. 535.
[47] H. Rudin and C. West, eds., *Protocol Specification, Testing and Verification VII*, Elsevier Science/North-Holland Publishers, Amsterdam, 1987.
[48] L. Andriantsiferana, B. Ghribi, and L. Logrippo, "Prototyping and Formal Requirement Validation of GPRS: A Mobile Data Packet Radio Service for GSM," *Dependable Computing for Critical Applications 7*, 1999, pp. 109–28.
[49] R. O. Sinnott, "Specifying Aspects of Multimedia in LOTOS," *Proc. ICCIMA '99 (Int'l. Conf. Computational Intelligence and Multimedia Applications)*, 1999, pp. 326–30.
[50] F. Babich and L. Deotto, "Modeling and Performance Analysis of Resource Allocation Strategies for Real-Time Services in UMTS using TIPPtool," accepted for publication, *Performance Evaluation*, Elsevier.
[51] G. Leduc and F. Germeau, "Verification of Security Protocols using LOTOS Method and Application," *Computer Communications*, vol. 23, no. 12, July 2000, pp. 1089–103.
[52] L. M. Kristensen, S. Christensen, and K. Jensen, "The Practitioner's Guide to Coloured Petri Nets," *Int'l. J. Software Tools for Technology Transfer*, vol. 2, Springer Verlag, 1998, pp. 98–132.
[53] www.daimi.aau.dk/designCPN/
[54] J. Xu and J. Kuusela, "Modeling the Execution Architecture of a Mobile Phone Software System by Colored Petri Nets," *Int'l. J. Software Tools for Technology Transfer*, vol. 2, Springer-Verlag, 1998, pp. 133–43.
[55] K. G. Larsen, P. Pettersson, and W. Li, "UPPAAL in a Nutshell," *Springer Int'l. J. Software Tools for Technology Transfer*, vol. 1+2, 1997.
[56] R. Alur and D. L. Dill, "A Theory of Timed Automata," *Theoretical Computer Science*, vol. 126, no. 2, April 1994, pp. 183–236.
[57] H. Bowman *et al.*, "Automatic Verification of a Lip Synchronization Algorithm using UPPAAL," *Formal Aspects of Computing*, vol. 10, no. 5/6, 1998, pp. 550–75.
[58] P. R. D'Argenio *et al.*, "The Bounded Retransmission Protocol Must Be On Time!," *Proc. TACAS '97*, *Lecture Notes in Computer Science*, vol. 1217, Springer-Verlag, 1997, pp. 416–31.
[59] ITU-T, Recommendation Z.120 Message Sequence Chart (MSC), Nov. 1999.
[60] OMG Unified Modeling Language Specification, Version 1.3, March 2000.
[61] M. von der Beeck, "A Comparison of Statecharts Variants," *Lecture Notes in Computer Science*, vol. 863, Springer Verlag, 1994, pp.128–48.
[62] M. Jmaiel, "A Unified Algebraic Framework for Specifying Communication Protocols," *3rd IEEE Int'l. Conf. Formal Engineering Methods*, Sept. 2000, pp. 57–65.
[63] M. Broy, "Toward a Mathematical Foundation of Software Engineering Methods," *IEEE Trans. Software Eng.*, vol. 27, no. 1, Jan. 2001, pp. 42–57.
[64] D. G. Priddin, "Method Integration for Real-Time Software Design," *IEE Colloquium on Applicable Modelling, Verification, and Analysis Techniques for Real-Time Systems*, 1999.

Additional Reading

[1] M. A. Ardis *et al.*, "A Framework for Evaluating Specification Methods for Reactive Systems Experience Report," *IEEE Trans. Software Eng.*, vol. 22, no. 6, June 1996, pp. 378–389.

Biography

FULVIO BABICH [M'95, SM'02] (babich@univ.trieste.it) received the doctoral degree in electrical engineering from the University of Trieste in 1984. In 1992 he joined the Department of Electrical Engineering (DEEI) of the University of Trieste, where he is Associate Professor of Digital Communications. His current research interests are in the field of wireless networks, telecommunication protocols, and personal communications.

LIA DEOTTO was born in Udine, Italy. She received the doctoral degree in electrical engineering in 1998 and Ph.D. degree in telecommunication engineering in 2002 from the University of Trieste. Her main research fields are formal methods for telecommunication protocols and wireless systems. Her current activity is devoted to specification and development of radio and mobility protocols in GPRS mobile networks. She is with Telit Mobile Mobile Terminals, Italy.