INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Tailored Protocol Development Using ESTEREL

C. Castelluccia, I. Chrisment, W. Dabbous, C. Diot,
C. Huitema, E. Siegel, R. de Simone

N° 2374

Octobre 1994

PROGRAMME 1

Architectures parallèles, bases de
données, réseaux et systèmes distribués

ISSN 0249-6399

Tailored Protocol Development Using ESTEREL

C. Castelluccia, I. Chrisment, W. Dabbous, C. Diot, C. Huitema, E. Siegel, R. de Simone*

Programme 1 : Architectures parallèles, bases de données, réseaux et systèmes distribués

Projet RODEO

Rapport de recherche n°2374- Octobre 1994

51 pages

Abstract: The rapid evolution of networking and the multiplication of new applications re-emphasizes the importance of the efficient communication supports. Implementations must be able to take maximal advantage of the details of application-specific semantics and of specific networking environments. In other words, the application needs to have more control over data transmission. Such control can be obtained by tailoring the communication facilities (or protocols) to the application characteritics, and by integrating the communication control to the application. Because such a task is too complex to be realized manually, we propose to automate the protocol development process using a formal approach. This report presents our approach to the automated design and implementation of application-specific communication protocols based on information provided by the application. Starting from the formal description of an application, our approach is based on a tool called "Protocol Compiler" that will automatically produce the implementation of a communication protocol tailored to the application. The formalism we use is ESTEREL, a synchronous reactive language dedicated to the description of real-time systems. Protocol description and verification using ESTEREL are described, as well as protocol optimization and implementation principles.

Key-words: Communication Protocols, Specification, Automated Design, Verification, Customization, Formal Technics

(Résumé : tsvp)

* Email: first-name.second-name@sophia.inria.fr

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
Unité de recherche INRIA Sophia Antipolis : 2004, route de Lucioles - B.P. 93 - 06902 Sophia Antipolis cedex **(France)**
Téléphone : +33 93 65 77 77 - Télécopie : (33) 93 65 77 65 - Télex : 970 050 F

Établissement public national à caractère scientifique et technologique - Décret N° 85.831 du 2 août 1985

Développement de Protocoles de Communication en ESTEREL

Résumé : L'évolution rapide des moyens de communication et la multiplication des applications distribuées nécessite l'utilisation de moyens de communications efficaces et performants. L'implantation des protocoles de communication doit prendre en compte les caractéristiques de l'application et les spécificités du réseau sous-jacent avec un maximum de précision. En d'autres termes, une application à besoin de plus de contrôle sur les données qu'elle souhaite transmettre sur un réseau. Le contrôle de transmission de donnée peut être réalisé en construisant un support de communication (ou plus simplement un protocole) dédié à l'application, et en intégrant ce support de communication à l'application. Cette tâche étant trop complexe pour être réalisée manuellement, nous proposons dans ce rapport une approche automatique utilisant des techniques formelles de description de protocoles. Ce rapport présente notre approche de conception et d'implantation automatisée de protocoles de communications dédiés à une application. Cette approche s'appuie sur une spécification de l'application écrite en ESTEREL (formalisme appartenant à la famille des langages réactifs synchrones) ; et sur un outil appelé "compilateur de protocoles" qui va générer automatiquement le protocole et son implantation à partir de l'analyse de la spécification de l'application. Le rapport montre comment on peut spécifier et vérifier un protocole de communication décrit en ESTEREL, ainsi que les techniques d'optimisation et d'implantation automatique intégrée.

Mots-clé : Protocoles de communication, Spécification, Conception Automatique, Vérification, Optimisation, Techniques Formelles

1 The HIPPARCH Project

This report is the deliverable for task B.1 within the scope of the HIPPARCH project. The following subsections will present a brief overview of both the project and this specific task.

1.1 Project overview

The HIPPARCH project proposes to study a novel architecture for communication protocols based on the *Application Level Framing* (ALF) and *Integrated Layer Processing* (ILP) concepts [Clark90]. Its global objective is to build a development environment which automatically generates the communication module required by a distributed application, using application-specific knowledge to tailor this communication module for improved performance. The architecture will be validated by prototype protocol implementations and test application demonstrations.

More specifically, this project addresses the design of efficient and application-oriented communication systems over high speed networks. The main objectives can be specified as follows:

- The design of an architecture for integrated design of communication protocols: based on the knowledge available from integration techniques for communication protocols, techniques for grouping protocol functions in order to meet specific application requirements will be proposed and tested.
- The definition/selection of a specification language to describe the application requirements.
- The design and implementation of a compiler to automatically generate the customized communication system based on integrated protocol modules. This will be the main deliverable of the project.
- The design and implementation of an efficient execution environment capable of exploiting multiprocessor systems, for the code generated by the compiler.
- The building of prototype integrated implementations of communication protocols for a set of test applications.

An experimental methodology will be used, starting from feasibility experiments and progressing to case studies. Implementations generated by the compiler will be validated by comparison with manually generated implementations. The following phases in our methodology can be identified:

- Experimentation with new and existing integration mechanisms, with implementations and their execution environments.

- The expression of application requirements. Following initial experimentation, the following parallel activities will be pursued:

  Formulation of the specification language.

  Design and implementation of the execution environment

  Conceptual work on the architecture, including dynamic configuration

- Design of the compiler and its runtime system.

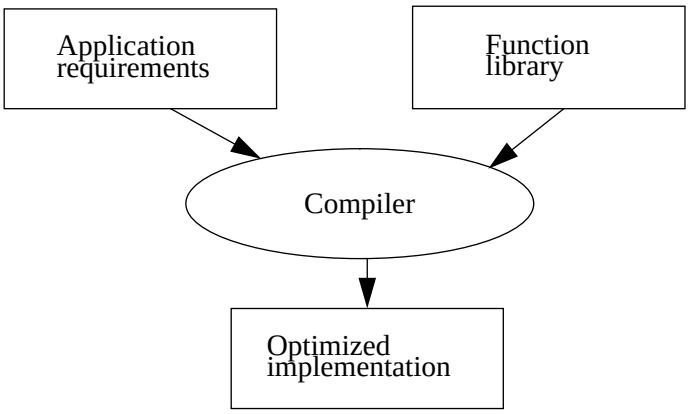


Figure 1 : Architecture of the HIPPARCH Protocol Compiler

The project involves three European and one Australian research groups (INRIA (F), University College London (UK), the Swedish Institute for Computer Science (S), and the University of Technology, Sydney (Au)) , all working on various aspects of the communication architecture and protocol design and specifically on the ALF/ILP concepts.

1.2 Selection of a specification language

This task aims at performing a careful survey of the existing languages which may be good candidates for protocol specification and for expressing high level application requirements.

Taking into account earlier experience in the field, it is expected that the selected language will exhibit the following features:

- it is a formal language, with formal syntax and semantics,
- it consists of a control part and a data part which are as independent as possible,
- the data and the control part should be consistent in order to enable validation of the specification as a whole, rather than exclusively of the subset.

- the language should include features making it possible to perform stepwise refinement of the application profile specification.

This report presents a survey of several languages and formal description techniques. However, special emphasis will be given to the ESTEREL language. This a priori choice will be justified later in this report. The structure of the report is as follows. In the following chapter we discuss the process of communication protocol development, highlighting the aspects relevant to the choice of a specification language. In Chapter 3, we give the rationale for our choice of the ESTEREL language rather than another more "classical" or "dedicated" language, as our specification language. In the process we discuss properties of a representative set of specification languages, and compare ESTEREL to some of these other languages. Chapter 4 presents a brief overview of the ESTEREL language, focusing on the properties relevant to protocol description. Chapter 5 then describes a set of experiments designed to analyze the behavior of ESTEREL and of its development environment within the context of protocol development. Chapter 6 first discusses the requirements of protocol development and, in light of our experiments, identifies the strengths and the limitations of ESTEREL for this purpose. We conclude the report by discussing how a protocol compiler based on ESTEREL could be designed based on our accumulated exeperience.

2 Development of Communication Protocols

The traditional development of communication protocols can be viewed as a sequence of tasks. The protocol is usually specified in a natural language; occasionally, limited efforts are made to apply aspects of formal languages or specifications as early as the design step[Danthine 92]. A natural language specification describes, with questionable accuracy, the protocol functionalities and its expected behavior.

The validation of protocol specifications is based on formal languages like LOTOS [ISO 87a], SDL [CCITT 87,Belina 89] or ESTELLE [ISO 86]. This step consists of proving that the protocol definition does not include any situation which would be unexpected or prone to produce deadlocks. It gives no indication of the protocol efficiency. Simulation is based on dedicated tools associated with specific languages like QNAP [Veran 84]. It makes use of a different description (called a "model") of the protocol to be simulated.

Protocols are generally implemented directly from the natural language specification, taking into account the characteristics of the target system and environment. A few tools exist [Bochman 87] [Abbott 92], but they are still in experimental stages. Conformance testing fills the continuity gap between the original specification and an implementation. These tests provide guarantees that the implementation is consistent with the protocol specification. But the guarantee is not at 100% since the test patterns are generated semi-automatically and in a non-deterministic way.

The rapid evolution of the global networking environment and the multiplication of new applications re-emphasizes the importance of the efficiency of communication support. In particular, in many cases an implementation must be able to guarantee a defined level of performance (or service). In order to honor such a guarantee, an implementation must be able to take maximal advantage of the details of specific application semantics and of its specific networking environment. As traditional implementation methodology is mostly "intuition based on experience", implementations produced by the traditional approach are often unable to effectively access or process the necessary information, and are thus often not sufficiently efficient.

We claim that it is essential that communication support be customized to application requirements (ApRe) and to the physical environment, and that in order to be reliable and effective such customization must be largely automated. This would guarantee to the user that the functions and mechanisms used to transmit data correspond to application needs in terms of, for example, reliability and security, and that they are as efficient as possible within a given set of constraints.

Thus, what we need is an integrated environment for automated implementation based on behavioral specification facilitating the incorporation of application semantics into the communication subsystem. The environment would support a development process which starts from a protocol specification and yields an automatically (or semi-automatically) generated protocol implementation customized to application requirements. Such an environment is necessarily based on a formalism that has to be introduced as early as the protocol definition step; the introduction of such a formalism increases the reliability of the protocol development process. The use of such a development environment allows real customization to the application needs, reduces the cost of communication support development, guarantees the required level of service and performance, and generates efficient implementations.

The first step toward such an integrated environment is a protocol specification language which is sufficiently flexible to incorporate application requirements and sufficiently formal to to support automatic generation. This report analyzes the feasibility of using a synchronous language (in particular the ESTEREL language) for this specification within the scope of the HIPPARCH project. HIPPARCH does not focus on verification, simulation, and testing, but rather on efficiency ; however, these aspects of the protocol development process are also of significant interest and will be touched on in later chapters.

3 Rationale for the ESTEREL choice

The choice of ESTEREL is not the result of a formal review of all available languages for protocol description, but the result of an informal comparison with other languages. These "a priori" criteria of choice are developed in this section in two parts. In the first part, we present the main choice criteria. In the second, we briefly

describe other possible candidate languages and show their pros and cons for protocol description and automated implementation.

3.1 Choice criteria

As described in the introduction, our goal is not only limited to protocol specification. We believe that the interest of a formal description is very limited if this description cannot be used for protocol implementation. The description language we need will be the skeleton of a protocol development chain which starts at the specification level and finishes at the implementation level. Consequently, the language and its environment must facilitate:

- protocol description at various levels of abstraction;
- modular and flexible specification design; and
- protocol development (modification and optimization).

As there is no perfect language, we prefer to compromise and start with an existing language that suits the majority of our needs. We then use a series of experiments to understand its associated strengths and limitations well enough to consider potential improvements as they become necessary. In fact, many of the languages described in the following section could have been chosen, as well as some we have neglected to mention. Our choice is unsurprisingly influenced by our background and environment, since the opportunity to interact with the language designers and possibly to influence the implementation is a distinct advantage.

3.2 Protocol description languages

In this section we categorize some of the relevant language choices available for our task, focusing on the characteristics that relate specifically to issues of protocol specification.

3.2.1. Formal Description Techniques (FDTs)

Formal description techniques are in fact languages that have been designed to support the specification of communication protocols (and systems in general) for standardization purposes. Among these languages, the most "popular" are ESTELLE, LOTOS, and SDL. The main reason we did not choose one of these languages is that since they are languages designed for formal verification they are not particularly implementation-oriented; formal description is only a part of the problem we want to solve. The distance between an initial description in these languages and the final implemented code often becomes too large for the formal verification of the former to realistically apply to the latter. Let us give more details for each of these languages:

- ESTELLE : Estelle is the most "implementation oriented" of these languages. It could be used for our purpose, but it seems in fact too "implementation oriented"

to allow sufficiently flexible automation. Estelle is based on asynchronous communication by unbounded queues, which is by itself an implementation choice we cannot generalize in order to produce efficient and highly optimized implementations. Another limit in Estelle is the lack of synchronization mechanisms.

- LOTOS : Lotos certainly has the highest abstraction level one could wish for, but from our point of view this is its main problem. First, a protocol description has to be "readable" : specifications written in Lotos are complex, and difficult to digest (generally longer than the description of the protocol in a simple natural language (like English!). The second limit of LOTOS : the abstract data types. There are no tools today that efficiently manage the LOTOS abstract data types. That causes most of the power of the language to be lost.
- SDL : SDL is an older formalism largely based on graphical presentation. Its semantic is not fully formally defined. It serves mostly as documentation support, even through extensive ad-hoc tool support is provided.

3.2.2. General purpose languages

By this, we mean languages with notions of communication and parallelism, but not particularly dedicated to specification and to communication protocol description; most programming languages are included in this category. General purpose languages could also be divided among different domains, such as object and non object languages, or high level and low level languages. Despite a general lack of specific support for protocol specification, there are a few of them that could nevertheless be considered for this purpose, in particular the CSP [Hoare 79] based languages (e.g. OCCAM and ADA) and VHDL [Airiau 90][IEEE 87].

3.2.3. Dedicated languages

These languages have been designed specifically for protocol description or implementation: MORPHEUS [Abbott 92], TNP [Merlin 76], FAPL [Smith 83]. Unfortunately, these languages tend not to be widely used outside their development environments, and either as cause or as effect the associated tool and development environments tend not to be very developed ; thus, the stability of the language is not guaranteed. Moreover, one can question the advisability of defining new dedicated languages when there are so many well known existing examples which could be used with minor changes[Bochman 90].

3.2.4. Semi-formalisms

We call data description languages like ASN1 [ISO 87b] or XDR [Corbin 91] *semi-formalisms*. These languages obviously cannot meet our requirements; they are dedicated to the description of data, but they cannot describe how these data are processed. However, in combination with another language used for algorithm description, they can :

- complement the other language where it has a lack of power.

- offer a very flexible way to tackle implementation problems, by clearly separating control description from data description.

3.2.5. Synchronous languages

Synchronous languages have been designed for the description and the implementation of the control part of (soft) real-time and reactive systems (systems that "react" to their environment). The best known are LUSTRE [IEEE 91], SIGNAL [IEEE 91], and ESTEREL [Berry 92,Boussinot 91]. STATE CHART [Harel 87], which we will not discuss in this report, is a graphical formalism related to this group. Synchronous languages allow only for control description ; data description is not possible. They should consequently be associated with another language (a general purpose language or a semi-formalism) which handles the data descriptions necessary for e.g. protocol implementation.

4 Highlights of ESTEREL

4.1 Relevant properties of ESTEREL

The reasons we decided to use ESTEREL for protocol description are numerous. From a language point of view, we have seen in the previous subsection that there are several reasons to be very careful in saying that one language is better than the others. The reasons for our choice are thus not restricted to issues of syntax or semantics.

First of all, ESTEREL provides only for the description of the control part of a protocol. This feature creates a strong separation between specification issues and implementation issues. Moreover, the combination of ESTEREL with a data description language offers a very complete environment for protocol implementation.

ESTEREL is a synchronous language which reacts to external events. That means the treatment of an external event is processed instantaneously at the time it occurs. The effect of this hypothesis on protocol implementation is that the treatment of external events is atomic. Once processing is started, no other event can interrupt this treatment. The definition of an "event" can vary according to the level of abstraction of the description. Furthermore, ESTEREL does not know about data types, but about signals. An ESTEREL description is composed of signals, and relations on signals allow the ESTEREL description to be optimized. We will show how this facility can be used in the following section.

Another criterion of choice is the environment of the ESTEREL language. We can identify two different aspect in this environment :

- The development environment [Berry92] : ESTEREL is provided with a compiler, a debugger, and a simulator. Different versions of the compiler are available according to the data description language used in complement (C, ADA, LISP).

- The validation environment [Madeleine89, Boudol89, Touati 93, Roy 90] : Tools for description verification and for description optimization are also provided with the ESTEREL language. Verification is limited to the control part of the protocol. Verification in the ESTEREL environment consists of proving whether or not designated paths in the description exist. This verification is not exhaustive like the one proposed for the LOTOS environment [Garavel90], but it works on real cases.

A general characteristic of this environment is it offers very friendly interfaces. Automata can be visualized, and the protocol can be followed, instruction by instruction, using the debugger. All the tools enumerated are grouped in an integrated environment : TKMEIJE [http94].

The final reason for our choice is that ESTEREL and its environment have been designed jointly by the CMA and INRIA. The group that updates the ESTEREL language and designs new releases of the compiler and of the environment is co-located in Sophia Antipolis. This is a strong argument if we decide, for example, to make some modification to the language syntax, or if we need to modify the compiler for any reason. The ESTEREL group, which has worked very closely with us since the beginning of HIPPARCH, will participate in protocol verification and in the modification of existing tools.

4.2 An overview of ESTEREL

ESTEREL is an imperative language belonging to the family of the ***synchronous reactive formalisms*** (SRFs)[Berry92]. Some other members of the family are the data-flow languages Lustre [IEEE91] and Signal [IEEE91], and the graphical formalism of Statecharts [Harel87], to mention only the closest ones.

Synchronous reactive formalisms in their original form are meant to represent Finite State Machines (FSMs) with multiple-input/multiple-output behaviors. More precisely the word "reactive" tags the fact that such systems have successive configurations, reached by successive transitions (or reactions) ; this is as opposed to functional formalisms where an initial datum is processed to a unique final result. On the other hand each single reaction consist of the "synchronous" reception of several events -as inputs-, together with an instantaneous emission of output events in response. Events are called signals, in a way evocative of actual communications. The signal is the basic ESTEREL data structure. A signal can be compared to a CSP channel [Hoare79]. It consists of a state (present or not present) and in a value; a null-valued signal is called *pure*. A synchronous reactive system is thus engaged in a perpetual dialog with its environment, exchanging signals at discrete instants in time.

A communication protocol description in ESTEREL can be seen as a black box, with input and output signals, where input signals are the information carried by the incoming event (numbers, type, length, etc.) and where outputs are the events to be delivered to the application or to be transmitted on the network. Because of the

synchrony hypothesis, the treatment of incoming events is instantaneous ; in other words, input and output signals are simultaneous, but causally related. This synchrony hypothesis fits well with the description of communication protocols. The instantaneous treatment hypothesis will be implemented by an "atomic treatment" of the incoming events, which means that the implementation environment has to provide queuing facilities to buffer events that may be received while processing the current event.

ESTEREL is mostly a control description language. Data can be described in ESTEREL, but their type and structure remain opaque. Type/constant/function/procedure names exist, but they are not related to a real data type. Data variables of various types can then freely be declared, assigned and used as arguments of functions and procedures, tested and so on. The implementations of data types and data operations will only be linked with an external module (written in a data description language like C) in the last phase of the compiler, when executable code is produced. This proved flexible by allowing us to defer data handling to full-scale existing compilers, but it precludes reasoning on actual values prior to this stage, since the meaning of data operations is left uninterpreted. Common types such as integers, booleans and strings are predefined.

The particular place of ESTEREL in the SRF family goes with its inheritance from early Process Calculi theory. Its simple structural syntax is amenable to modular treatment, be it for semantics definition, compositional reasoning or program construction. This means that, while such topics may still sometimes remain technically subtle, they can be split by the nature of the operator applied. ESTEREL contains syntactic constructions for sequential compositions, signal in/outputting, explicit atomicity operators to divide reactions, and preemption operators to set signal priorities. Perhaps most important, ESTEREL contains an explicit parallel construction. It may involve private signals, invisible to the outside environment, and induces a specific programming style, with modularity now being mainly embodied by the synthesis of large systems by the cooperation of smaller ones. Parallel modules share signals in a broadcast fashion. All these constructions are naturally defined in terms of appropriate compositions and transformations on the underlying finite state machines.

Formally the syntax of ESTEREL contains the following elements (apart from the declaration parts and many derived constructions to ease programming):

P = stop | nothing | emit S | P || P | P ; P | present S then P else P end | (P) |
do P watching S | signal S in P end | loop P end | exit T | trap T in P end |
var X : Type in P end | if B, then P else P end | call Proc | X := Funct

where S, T, X, Type, Proc, Funct belong respectively to the syntactic classes of signals, exceptions, variables, types, procedures and functions.

By far the most important semantic issue for SRFs lies in their dealing with instantaneous causality. The problem is related with electronic race conditions in the case of hardware circuits. For instance local signals may respectively result from one another in parallel signal exchanges, as in the simple example "present S then emit S". Then several stable values (or none) can sometimes be found for these internal events, resulting of ill-caused situations. Characterizing sound programs is thus imperative, but exact characterization can be computationally costly. The ESTEREL compiler implements fast algorithms with good approximations, in the sense that all ill-caused programs are rejected, as well as (a few) programs which could be given a sound semantics.

As a consequence of soundness, programs should result in deterministic complete FSMs, where in each state for any input event there is one and only one reaction, in terms of outputs and state transitions. Preserving this determinacy while allowing (synchronous) parallelism is usually seen as a major advantage of synchronous reactive languages.

4.3 ESTEREL tools

The ESTEREL compiler consists of several pipe-lined processors (parser, submodule linker, automaton expanser or boolean equation translator, executable code generator) and uses several internal code formats, some of which are shared with other synchronous reactive formalisms. There are two compiling targets for ESTEREL. One generates combinatorial boolean equations, the other sequential finite state machines. In both cases appropriate internal format descriptions exist, as well as postprocessors translating these formats into executable code. ESTEREL compilers are also able to translate description in to C, ADA, or LISP.

In addition, the ESTEREL environment contains various tools [Berry92][Roy90]:

- A parallel debugger.
- A graphical simulator with on-line source visualization of automatons.
- A proof system developed in the same team as ESTEREL, which works at the automaton level and allows for analysis, reduction and comparison of systems (against their specification). This proof environment is very useful, in the case of communication protocols, for specification and partial verification like deadlock detection, path analysis and branch elimination.
- In the boolean equation framework ESTEREL is also interfaced to the hardware synthesis system Sis developed at the university of Berkeley. It allows for semi-automatic code optimization.
- Other optimizers also exist in the automaton production line.
- An graphical interface to the environment is also distributed by the Ilog company.

Together, all these tools provide a very powerful environment for the development of complex reactive systems.

5 Experiments

In order to evaluate ESTEREL in the context of HIPPARCH, we have performed a series of experiments in order to verify the suitability of ESTEREL for communication protocol development. This section describes four of these experiments, and discusses how to use ESTEREL in order to make it efficient for communication protocol specification and development. These experiments focus on different objectives in order to provide a range of experience:

- The first experiment focuses on protocol description. We used ESTEREL in order to understand how a communication protocol can be specified and verified, and then to see whether this specification is generic enough to be readable and reusable for implementation purposes.
- The second experiment analyzes how to tailor a protocol description using the requirements transmitted by the protocol user. These requirements are presented as ESTEREL signals
- The third experiment consists of a modular implementation of the TCP protocol. It focuses on implementation and modularity problems. We also studied the performance issues related to the use of ESTEREL. The building blocks defining the basic communication functions were implemented in the C language (based on BSD-TCP).
- The fourth experiment consists of an implementation of a communiction subsystem which was automatically generated based on high-level specification of application requirements. ESTEREL is used to describe the control part, and ASN.1 is used to describe the data structures. This experiment explores the possiblity of expressing the communication requirements of a particular application and of generating an implementation of the corresponding communication subsystem.

5.1 Protocol specification and verification

5.1.1. Structure of the specification

As a finite (control) state machine, a communication protocol can be completely described by a collection of transition requirements presented as conditional instructions:

When the protocol receives the event EV-x **in** the state ST-y,
if predicates P1 to Pn are verified,
do actions A1 to An
and go to state ST-z

which typically refer to the internal states, at a low-level of abstraction. It can be translated in ESTEREL by

```
await EV-x do
   present ST-y then
      if [P1 and P2 and .... Pn] then
         call A1; call A2; .... call An;
         await tick; emit ST-z
end (if, present, await)
```

This monolithic style of description in not natural in ESTEREL, as its lacks modular structure and proves far too concrete to our aims of communication protocol design. It means that the designer has designed the Finite State Machine before he designed the protocol. A more natural way to design a protocol, and to describe it in ESTEREL, consists in a description of the temporal relations between the protocol events.

```
After having sent event EV-x then wait for event EV-y or EV-z
case
   EV-y do ....;
   EV-z do ....;
endcase
```

Using this approach, the FSM that describes the protocol control does not appear in the description ; but it is automatically created, analyzed, and managed by the ESTEREL compiler that will produce a set of boolean equation or an automaton (that describes the FSM of the protocol). In this case, an ESTEREL specification will have the following structure :

```
module protocol description
   .......
   await event
      call p0; call p2; .... call pn;
      case p0 to pn in
         call A1; call A2; .... call An;
         await next events
         .....
      endcase
   endawait;
   .........
endmodule description
```

Because of the "reactive" hypothesis, the predicate evaluation or the actions processing will be done simultaneously. Thus, from a functional point of view, the sequence "instruction 1 **;** instruction 2 **;** .... **;** instruction n" ("**;**" is the sequential operator) is equivalent to "instruction 1 || instruction 2 || .... || instruction n", where in-

struction 1 to n are processed in parallel. The theoretical behavior is different : in the case of sequential operator, a causal relation will be kept between instruction delimitated by "**;**". In the context of a specification that will be used later for implementation, the parallel operator will be preferred to the sequential one when there is no causal relation between instructions.

5.1.2. From specification to implementation

Using the formal specification of a protocol to deduct an implementation requires to fill the gap between the abstraction level of the specification (which only describes the protocol behavior) and the implementation (which has a very low level of abstraction). This gap can be filled introducing in the specification informations related to the implementation environment (Operating System primitives available, interfaces with the host environment, memory management, high performance implementation techniques available). From a single specification, it should be possible to generate different implementations (integrating local optimizations). To successfully transform a specification in an implementation, the specification must have been designed with this objective in mind ; that means for example that the specification must avoid all the instructions, constructions, or declaration that could later limit the range of possible implementations and optimizations.

Using the opaque data types of ESTEREL, it is quite natural to introduce in the specification the information that will make it an implementation. "Deliver incoming data on the fly if not corrupted" is a protocol function description in natural language. Using a pseudo-ESTEREL syntax, this description is :

var data_to_deliver : data**;**
procedure deliver (data)**;**
function not_corrupted (data) **: boolean;**
if not_corrupted (data_to_deliver) **then** deliver (data_to_deliver) **else nothing;**

In ESTEREL, data, data_to_deliver, deliver, and not_corrupted must be declared, but not defined. The same description at the implementation level could be (using a pseudo language) :

type buffer : **array** [1..N] **integer**
type buffer_address : **pointer on** buffer
type data : **record** [number : **integer**, reference : **integer**, adress **:** buffer_address]
var data_to_deliver : data
if not_corrupted (data_to_deliver) **then** deliver (data_to_deliver) **else nothing;**

procedure deliver (**var** data_to_deliver : data)
begin
% code describing the way data are delivered to the application
end
function not_corrupted (**var** data_to_deliver : data) **: boolean**

begin
% code describing the detection of corruption algorithm
end

The only difference between the specification and the implementation is the data type description, and the procedure and function code (written in a data description language) have been linked to the ESTEREL specification. Other implementations could have been designed using different data structures ; even a hardware implementation starting from boolean equation produced by the ESTEREL compiler (and the associated hardware synthesis system) is possible. Passing from the control description to the implementation is done automatically by the code generator of the ESTEREL compiler. Data structures and algorithms to be attached to the ESTEREL description have to be described in a .h (data types declarations) and in a .c (procedures and functions code) modules. There can be different .c and .h, considering different implementation environments. The protocol designer just has to describe to the ESTEREL compiler which module he wants to link to the ESTEREL module before generating its executable code.

5.1.3. Application to a simple protocol

Complex protocols have been designed in ESTEREL [Berry 91][Castel 94][Diot 94], which illustrate its typical specification style. For the sake of this presentation we shall use a more simple, "toy" transmission protocol. The protocol is initially waiting for data to transmit from the protocol user. The data is checked and discarded when corrupted. A valid data is immediately transmitted down the network and an acknowledgment is sought. A positive acknowledgment indicates successful data transmission ; upon a negative one, the data is retransmitted. The lifetime of a given data is bounded a timer ; on time out, data is discarded and transmission fails. Data are identified by a number and an application flow identifier. A formal description of the protocol written in ESTEREL is presented on Figure 2.

```
module protocol:
%% declarations
type        NUMBER, FLOW_ID, NDU, DATA;
input       DATA_TX, ACK, NACK, TIME_OUT;
input       DATA                    (DATA);
input       NUMBER                  (NUMBER);
input       FLOW_ID                 (FLOW_ID);
output      START_TIMER;
output      SEND_NDU                (NDU);
function    CORRUPTED               (DATA, NUMBER, FLOW_ID) : boolean;
procedure   DESIGN_NDU              (NDU)           (DATA,      NUMBER,
FLOW_ID);
procedure   DISCARD_TX_REQUEST      ()              (NUMBER, FLOW_ID);
procedure   SAVE                    ()              (NDU);
procedure   DISCARD_NDU             ()              (NUMBER, FLOW_ID);
procedure   FIND_NDU                (NDU)           (NUMBER, FLOW_ID);
relation    DATA_TX # ACK # NACK # TIME_OUT,
```

```
            DATA_TX => NUMBER, DATA_TX => FLOW_ID,
            DATA => DATA_TX, DATA_TX => DATA,
            ACK # DATA, NACK # DATA,TIME_OUT # DATA,
            ACK => NUMBER, ACK => FLOW_ID, NACK => NUMBER, NACK =>
FLOW_ID,
            TIME_OUT # FLOW_ID, TIME_OUT # NUMBER;

% body
var NDU : NDU in
    loop
        await DATA_TX do
            if  CORRUPTED (?DATA, ?NUMBER, ?FLOW_ID) then
                call DISCARD_TX_REQUEST ()(?NUMBER, ?FLOW_ID)
            else
                call DESIGN_NDU (NDU)(?DATA, ?NUMBER, ?FLOW_ID);
                call SAVE ()(NDU);
                [
                    emit SEND_NDU (NDU)
                ||
                    emit START_TIMER
                ];
                do
                    trap RETRANSMISSION in
                        loop
                            await
                            case NACK do
                                call FIND_NDU (NDU)(?NUMBER,
?FLOW_ID);
                                emit SEND_NDU (NDU);
                            case ACK do
                                exit RETRANSMISSION
                            end
                        end
                    handle RETRANSMISSION do
                        call DISCARD_NDU ()(?NUMBER, ?FLOW_ID);
                    end
                watching TIME_OUT
                timeout
                    call DISCARD_NDU ()(?NUMBER, ?FLOW_ID);
                end
            end
        end
    end
end.
```

Figure 2 : formal specification of a simple protocol (Transmitter side) written in ESTEREL

As mentioned before, information carried by incoming events is represented by a valued input signal (flow identifier, numbers, data). So, an interface has to be provided between the ESTEREL specification an its nesting host environment to :

- parse external input signals. Information transmitted by this parser to the implementation are input signals that describe the protocol generic parameters (numbers, counters, context identifiers) ; or the information carried by the incoming event (number, user data, identifiers, addresses, etc.).
- queue incoming events that could be received during the treatment of the current event (because of the synchrony hypothesis made by ESTEREL).
- adapt the implementation of the protocol to the data exchange rules of the host environment (sockets, streams, etc.).

The event parsing could be done in the ESTEREL description ; but parsing is more related to the implementation than to the protocol description. A clear separation between implementation related tasks and the description of the protocol behavior (or protocol specification) will make the derivation of the implementation from the specification easier. Parsing is closely linked to the interface between the protocol and its environment. Its description consists of memory management tasks, masks, and data manipulations.

Even if simple, this protocol specification makes use of all the ESTEREL constructions that are classically used to describe a complex communication protocol :

- *Trap* is the way to write conditional loops in ESTEREL. It is equivalent to "*Do* action *until* condition ; *when* condition *do* exception". Where there can be more than one condition to leave the loop.
- *Case await* allows a process to wait simultaneously on more than one input signal. The difference with the OCCAM *ALT* instruction is that *Case await* is deterministic : the first incoming event processed is not necessarily the first chronologically, but it is the first encountered while checking sequentially all the input signals.
- Parallel and sequential operators (respectively || and **;**) are very useful at the specification level. These operators will be interpreted at the code generation step.
- *watchdog* is one of the most original instructions of ESTEREL. A watching statement defines a limit for the execution of its body ; if the body terminates strictly before the limit, the whole watching statement terminates normally. If the body is not terminated when the limit occurs, the body is instantly killed without being executed at that time and the watching statement terminates. With respect to the synchrony hypothesis, the body does not take any time to execute.
- *relations* are part of the declarations in an ESTEREL module. Relations only apply to input signals. Relations give a way to describe to the ESTEREL compiler the behavior of the functional environment ; in other words, relations define the

way input signal are related to each other. Relations, and their role in module compiling are described in section 5.2.

5.1.4. Verification

We now describe several easy verification techniques which are possible with the ESTEREL programming environment owing to its sound semantic interpretation into FSMs.

Formal verification usually requires the user to choose a specification formalism in which to express the properties to validate. This formalism should be both expressive and simple, so that *model-checking* can be performed by evaluating the property on the FSM model, while not *too* expressive, so that differences can not be found in between intuitively equivalent models.

Existing formalisms, like modal temporal logics and its various extensions, impose yet another syntax on the user. We shall try to provide a simpler framework by following an approach in which correctness specification are themselves expressed as FSMs, and so homogenously with the programs to be verified. Then model-checking consists in establishing precise graph -or language- theoretic relations in between two objects of the same nature. The concerns of proper expressiveness and simplicity are now obviously fulfilled by this identity of programming and specification models.

Still, the advantage of modal temporal logic is to allow partial specifications, where only fragments of the full program behaviour is concerned. We shall thus introduce some counterpart in the specification automata approach: *observation* (or more generally *abstraction*) will allow the user to abstract away from many details in the program and concentrate only on those aspects which are of relevance to the current property .

Interestingly this approach, based on reduction of the program model, constructs smaller models and thus reduces their complexity. In many practical case the resulting abstract models become manageable by graphical display systems so that the user can do away with independent specifications altogether and just verify that the abstract graph structure obtained automatically and directly from the inplemented program itself has the intended behaviour. Also in case it does not, a rich diagnostic information is provided by the actual result which then departs from the expectations.

As an example of reduction one can think of compound procedural transactions. For instance in the world of protocols, a connection phase can consist in: send a connection request ; then wait for confirmation or busy notice, possibly under timeout guard ; on busy notice, try requesting again after some time ; when confirmed, notify connection success. At a higher level this is only one action, *connect*, so that the description is much simpler. We shall now describe our practical reduction techniques on the previous example. These techniques are embedded in the AUTO veri-

fication tool and its AUTOGRAPH graphical editor, fully interfaced to the ESTEREL environment.

The first most obvious validation is the mere graphical visualization of the underlying automaton. This was done for our toy example in figure 3. But in general models are too large to be simply contemplated. Then we use *observation* criteria to obtain *partial view* reductions which scale down the description to picture only behaviors on a handful of time-related signals. On the extreme we reduce meaningful sequences of behaviors to single abstract behaviors, and even proceed by refutation when introducing *undesirable* abstract actions. All these transformations aim at producing explicit and readable informations on a FSM model, far more simple than the original one but keeping in accordance with the single model framework. Further analysis techniques are also possible on reduced models, since their relevance to the original description is fully understood.

We now give example of such verifications by reduction on our small example. Due to the utter simplicity of the original automaton reductions will not be impressive, but the reader should consider that in larger examples the reduced automata sizes could stay of the same magnitude as global systems would grow much bigger.

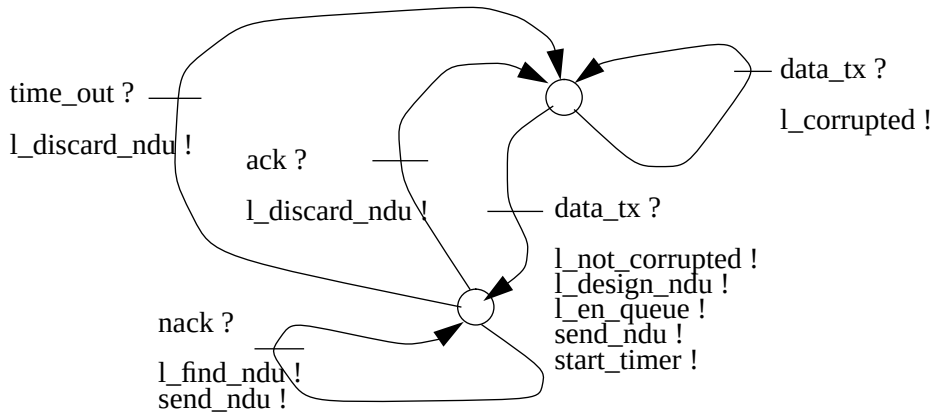


Figure 3 : automaton produced by the ESTEREL compiler. Input (output) signals bear "?" ("!") suffix. Local signals bear a "l_" prefix. These signals appear optionally in the displayed automaton .

A first type of reductions consists in *hiding* most input/output signals. Let us prove that a NDU is periodically created and discarded (it is never tentatively discarded when not effectively created). We then hide all signals except **DISCARD_NDU** and **DESIGN_NDU**. The result is displayed in figure 4, where **tau** represents an invisible action, of which transitions could in fact be erased.

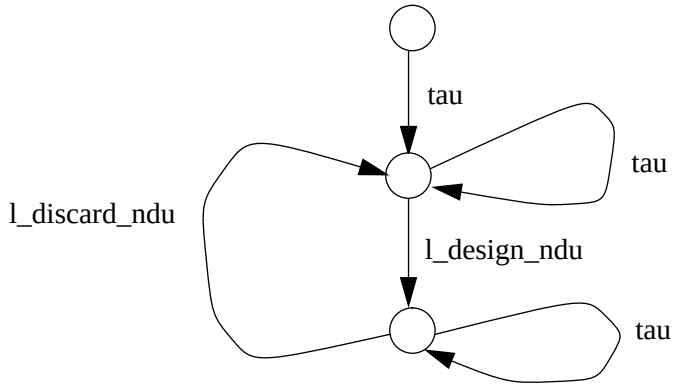


Figure 4 : reduction of the automaton for validation purposes.

A second type of reduction introduces the abstract action (to be refuted):

bad = /DATA_TX?:(not /ACK?)*:/TIME_OUT?:(not /DATA_TX?)*:/ACK?

It consists of a regular expression (with Kleene iteration star and ":" for sequencing and possibly "+" for alternative), based on behavior predicate selectors for single (concrete) steps. For instance /***DATA_TX?*** stands for any transition label containing reception of this input event, wile usual boolean operators (*not, or ...*) modify the predicates accordingly. The bad action considered here would be represented by concrete behaviors in the concrete model only if acknowledgment could be considered even after time-outs (and before the next data reception). The verification succeeds by proving the abstract action to be unfeasible. Otherwise one gets a clear counter example where its exact way of performance is recorded. Similarly the abstract action

bad = /DATA_TX?:(not(/ACK? or /TIMEOUT?))*:/DATA_TX?

is refuted exactly when a new data is not accepted unless the previous one has been either acknowledged or timed out.

The previous examples only sketch the method. Further reductions may be obtained by state quotient (minimisation) relative to behavioral equivalences (bisimulation, language equivalence), or by transition elimination according to context offers which describe possible input events provided by the environment through time. Also, abstraction with several actions (we only used one "bad" action here) allows richer abstract analysis of behaviors.

5.2 Formalism and Optimization

The ESTEREL compiler can be used to automatically optimize a protocol description and design the proper FSM implementation. Customization to the applica-

tion will use the same feature of the ESTEREL language and compiler than optimization.

The basis of optimization in ESTEREL is the *relation* declaration. ESTEREL allows a notion of dependence *relations* between input signals. *Relations* provide the ESTEREL compiler with a knowledge of the predictable co-occurrences of different input signal as offered by the functional environment. Basically two inputs can be exclusive, an input may force another (or no relation can hold in between them). The compiler uses existing (declared) relations to perform important optimizations. The syntactic declarations are :

- exclusion : *A # B* means signals A and B cannot be present together at the same instant.
- implication : *A => B* means if the signal A is present, the signal B is also necessarily present at the same instant. Conjunction of *A => B* and *B => A* (that we will note *A<=> B*) means A and B are always present together, in the same instant.

The previous protocol has four different events. The relation

data_tx # ack # timer # nack

means that none of these events can occur (and can be processed) simultaneously. This relation has to be introduced in the ESTEREL description of the protocol to obtain the automaton of figure 3. Without this relation, extra transitions would have been created by the compiler for simultaneous receptions of events on input signals.

The protocol events carry information : data (*dt*), application flow identifier (*flow-id*), data number (*num*), etc. These informations will be described in ESTEREL as input signals too. They will be synchronized on the input events using relations. The relations

data_tx => dt ; data_tx => num ; data_tx => flow_id;

timer # dt ; *ack # dt ;*

instruct the compiler that a "data_tx" primitive carries always a number, a flow identifier, and user's data ; on the other hand neither timers nor positive acknowledgment s carry user's data. So, the more accurate the description of the relations between input events is, the more optimized the produced automaton will be.

A customized communication protocol is obtained in ESTEREL by introducing new constraints on the input signals that will select or leave out subsets of the full described behaviors, according to the level of reliability or service required by the application. In the previous protocol, for example, possible customizations concern: preserve or disable the retransmission and/or use or don't use positive acknowledgments. These possible optimizations (by contextual reductions) of the protocol de-

scription will be achieved in ESTEREL defining a set of relations which correspond to the proposed options :

- no positive acknowledgment will be said by the relation : nack # tick ; (1)
- no retransmission will be said : ack # tick ; nack # tick ; time_out # tick ; (2)

Tick is an input event which exists in every ESTEREL module. *Tick* is present at each instant. *signal # tick* means signal will never occur. Consequently, all the branches of the automaton related to *signal* can be removed by the compiler.

The role of the Protocol Compiler is to add the set of relations that corresponds to the application characteristics

- relation (1) will reduce the original automaton (figure 3) to the one described figure 5a.
- relation (2) will reduce the automaton to a single state / two transitions (one for reject and the other for normal transmission) automaton presented figure 5b.

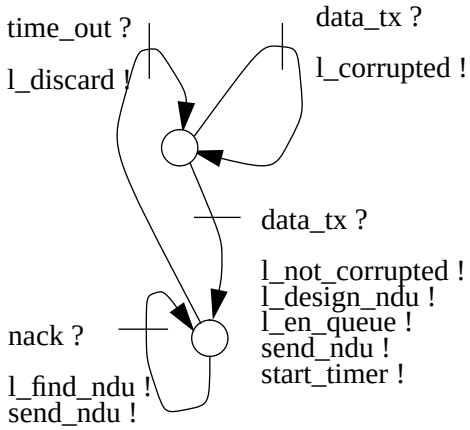


Figure 5a

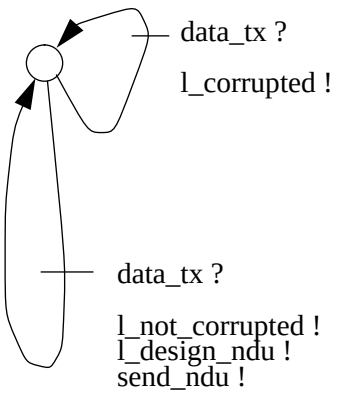


Figure 5b

Figure 5 : customization of the protocol.

This customization approach is very interesting because there is almost no need to add extra information in the protocol description. The work of the Protocol Compiler is just to select amongst the input signals which are to be kept, and then simply call the ESTEREL compiler.

The way the protocol compiler will analyze the application to extract Application Requirements and design the corresponding set of relation as not been studied today.

5.3 Automated implementation of modular protocols

Synchronous languages were specifically designed for implementing reactive systems. Protocols are good examples of reactive systems; they can be seen as "black boxes", activated by input events (such as incoming packets) and reacting by producing output events (such as outgoing packets). Synchronous languages are used to implement the control part of a program; the computational and data manipulation parts are performed by functions implemented in another language (for example C in our implementation), which are invoked by the control program.

As we have seen, ESTEREL programs are composed of parallel or sequential modules which communicate and synchronize using signals. The output signal of a module is broadcast throughout the whole program and can be tested for presence and value by any other modules. This communication mechanism provides a lot of design flexibility, because modules can be added, removed or exchanged without perturbing the overall system. A module is defined by its inputs (mutable activation signals), sensors (immutable activation signals) and outputs (signals emitted). Module inputs can be the outputs of another one (modules executed sequentially) or external inputs (such as incoming packets). The design of an ESTEREL program is then performed by combining and synchronizing the different elementary modules using their input, sensor and output signals.

The modularity of ESTEREL programming has been demonstrated and illustrated in [Castel94] where we describe the implementation of a data transfer protocol using ESTEREL. The specification of our protocol is very similar to that of TCP protocol (we omit the connection establishment and termination phases, but implement the complete error and flow control mechanisms). We address how to design the building blocks and how to combine them to generate the required protocol. Reusability and flexibility were our main focus here. The building blocks must be designed so that they are meaningful to the designers and so that changes in the protocol specification only induce local changes in the architecture and the code. The overall goals of this case study were thus to test the validity of our approach and to give an insight on building-block contents.

In this work, we implemented our protocol incrementally and ran it at each step to test its correctness. We started from a simple protocol and added modules step by step until we accomplished all required functionalities: according to the precepts of Object Oriented Programming we followed the rule one functionality-one module. Our final specification is composed of three main modules : the SEND, RECEIVE and CONNECTION modules. Each of them is composed of several parallel submodules wich implement the different functionalities of the protocol, such as flow-control, computation of the roundtrip time or management of the retransmission timer (cf. Appendix A for more details). As a result of this programing style, the final program obtained is very modular; we can easily remove or exchange modules without

modifying the overall program structure. The isolation of the different functionalities into separated and concurrent modules thus leads to very modular structures.

5.3.1. ESTEREL Compilation Phase

An ESTEREL module gets activated on the reception of one or several input signals , executes some actions and emits one or several output signals. It uses local variables (invisible to the others modules) and communicates with other modules using signals (global variables). The ESTEREL compiler generates a sequential automaton in a target language (C in our implementation) from a parallel specification by resolving resource conflicts. It assigns a code to every elementary action (e.g. assignment, test) of each module in the program text, and serializes these codes such that actions are always performed on the latest emitted signals within an instant. For example, if a module **A** needs the value of a signal **sig1** for its internal processing and **sig1** is modified and emitted in the same instant by module **B**, then the ESTEREL compiler serializes theses two components such that the code modifying and emitting **sig1** be scheduled before the code using its values. If no schedule can be found, the program is rejected. This is what is called a causality error. Signals do not appear in the automaton. They are implemented as global variables, available to all modules that declare them as inputs or outputs. Emitting a signal consists of updating the corresponding global variable; reading a signal consists of accessing its value.

5.3.2. Performance Issues

It is very difficult at this point of the project to assess the performance of programs generated from an ESTEREL compiler. In [Castel94], we show that the use of ESTEREL does not necessarily imply bad performance.To demonstrate this point, we compared the C code obtained from our ESTEREL specification with a handcoded version of TCP-BSD. Following the analysis approach described in [Clark89] we chose to focus on the input processing analysis, which seems to be the bottleneck in protocol processing. Our protocol is not a fully functional TCP, so it is dangerous to compare the input processing costs too closely and use comparison benchmarks such as instruction counts. We instead compare the sequence of actions executed on the packet input paths of both implementations. We believe that this coarse grain comparison should give a fair indication of the performance of the code generated by ESTEREL. We observed that the actions, their order of execution and the code executed on packet reception are very similar for both implementations. No specific overheads appear in the ESTEREL C code. Within a state, the code produced by ESTEREL is very compact.

Despite this similarity, some differences exist in the way these two programs are structured: the BSD-TCP implementation is "Action Oriented", while the ESTEREL implementation is "State Oriented". In fact, the BSD code is composed of a sequence of actions preceded by their condition of activation. This sequence of <condition, action > is processed each time an input event occurs. For example, the condition corresponding to the header prediction algorithm is tested each time the input

processing procedure is invoked. This is not always a desirable and time optimal feature.

The ESTEREL code is structured into states. All these states are distinct and only contain the actions possible in that particular state. For example, in the retransmission state the code corresponding to the header prediction algorithm does not appear, because no header prediction is possible in that state. The receive paths in the ESTEREL program should then be more time efficient, because no unnecessary tests are performed. However the price to pay for this performance improvement is a code size increase.

The code size of the BSD-TCP is optimal: code sharing is performed whenever possible. In the ESTEREL program, no code sharing is performed. So if an action is executed in several states, its code is duplicated in each of those states, which can lead to very large compiled programs. As a matter of fact, our ESTEREL program which is about 10000 lines long produces an 11 state automaton; the compiled code size of this automaton is about 100 kbytes, 4 times larger than the BSD one.

These preliminary results are very encouraging. This analysis shows that there is no inherent reason to believe that the ESTEREL code should not perform at least as well as the BSD-TCP version. In addition there is still the possibility of tuning the ESTEREL code (or modifying the ESTEREL compiler) to combine these approaches in a more optimal way.

5.4 Application constraint specification

The previous section shows the possibility of having a modular and flexible transport protocol. In this work we focus essentially on the application level, and try to understand how to benefit from knowledege of the behavior of a distributed application to improve the communication subsystem by adapting it to the application constraints.

Traditional distributed applications generallly demand transparency from the underlying networks. They run over generic communication systems that they view as *black boxes* and which often offer only a single comunication model (e.g. TCP/ IP or Remote Procedure Call [Birell84]). This is however not sufficiently general, since for example a file transfer application requires different communication constraints on synchronization and transmission than a multimedia application: a file-transfer application is reliable and not real-time , while a multimedia application can tolerate some loss and out-of-order delivery of data packets and may need to adapt to the dynamic network state. Recent research calls this transparency into question, pointing out how communication needs may depend on the application [Hutchinson91, Hoschka93, Zhang89] and how such *black box* communication systems do not enable the specification of those requirements. We have therefore studied the possibility of automatically generating a Remote Operation System Tailored to Applications

Requirements (ROSTAR) [Chrisment 94], by taking into account knowledge of the application control.

Our approach is based on two major axes.

- Let the application express its own constraints in a formal way in order to keep a certain level of abstraction (the success of RPC has validated the utililty of this approach).
- Generate a comunication subsystem directly from the formal specifications.

The protoype that we have implemented (: Remote Operation System Tailored to Applications Requirements) has demonstrated the feasability of the first two points.

5.4.1. Formal Specifications

To address the first point, we require a language which offers the ability to specify the communication control constraints:

- The internal scheduling of the application: (task parallelism and sequencing).
- The timing constraints i.e the delays relative to specific input or output events.
- The interactivity with the user (pause / start / stop) which constitutes a source of (less critical) timing constraints which may also affect the behavior of the communication system.
- The interaction with the network according to the type of synchronization mechanism: synchronous, asynchronous or isochronous.
- The occurence of exceptions and their associated handlers.

The ESTEREL language suits the requirements described above since it is especially designed to program the control part of reactive systems. That fits well with the distributed applications, which have to react to events coming from the user and from the network.

ESTEREL considers data as opaque. So we used another more appropriate language, ASN.1, for the specification of the data structures exchanged during the protocol. ASN.1 is a well-known standard for data presentation. Furthermore we have at our disposal the sources of an ASN.1 compiler, MAVROS [Huitema91], which generates the marshalling and unmarshalling routines directly from the specifications.

As ESTEREL is essentially an automaton specification language rather than a language dedicated to distributed applications, we have developped an upper level language, *ASN.1 extended*, which is easily comprehensible by a distributed application designer. This language is an extension of ASN.1 with the addition of a set of control primitives (cf Figure 6) . A set of *ASN.1 extended* specifications are translated into an ESTEREL program.

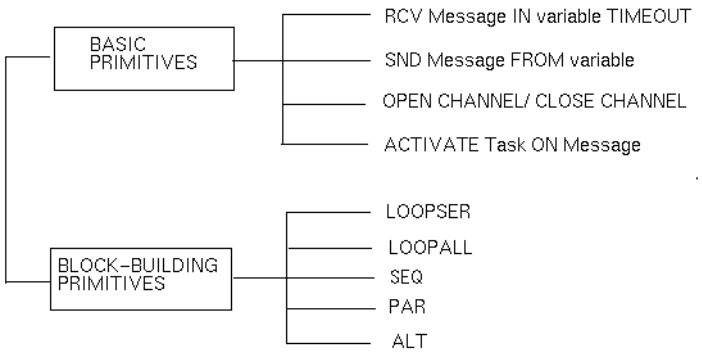


Figure 6 : *ASN.1 extended* language primitives

5.4.2. Automated Generation

The automated generation of the communication part of an application module is achieved through a multi-layered compilation approach (cf Figure 7) which includes and extends the functionalities of both MAVROS and the ESTEREL compiler. The multi-layered compilation approach is described in more detail in Appendix B.

The designer must specify the data and control specifications of the distributed application:

- The ASN.1 data specification which defines the data structures exchanged during the protocol.
- The *ASN.1 extended* control specifcation which is particular to each module of an application. A distributed application is considered to be composed of different modules located on different hosts.
- The compiler then generates:
- From the data specification, the marshalling and unmarshalling routines.
- From the control specification, a control automaton which uses the data structures defined in the data specifications. and which interacts with the (pre-existing) transport protocol to define how certain functions (e.g. flow control, error control) should be used. In a further step, the transport protocol automaton will also be automatically generated from user specifications.

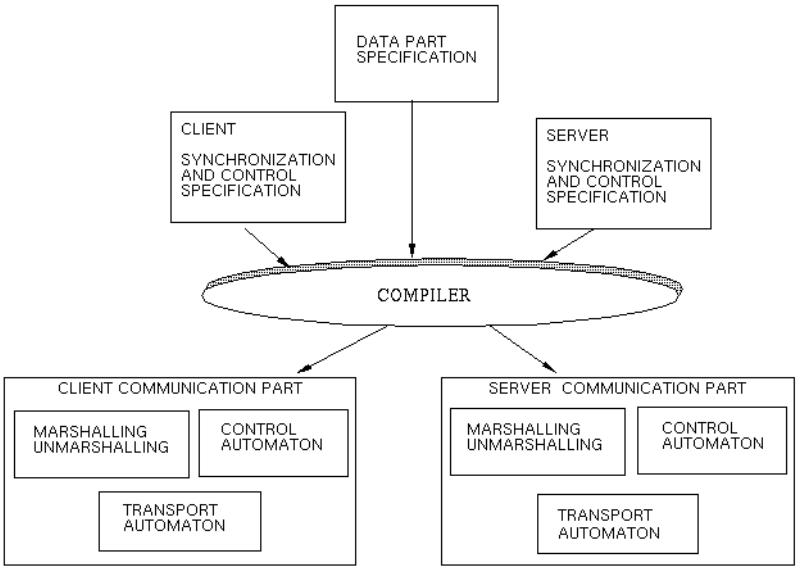


Figure 7 : Generation of Communication Subsystems with ROSTAR

In conclusion, the feasibility and the usefulness of our basic approach was demonstraged with the development of a prototype version of the overall system. However, we think a better use of network characteristics can be achieved by fully integrating the application part and the communication part within the same address space. This extension is presently under consideration.

6 Using ESTEREL for Protocol Development

We have explained why we chose ESTEREL as a language for protocol engineering. We have also shown some experiments using ESTEREL in the context of the HIPPARCH project. This section synthesizes the previous discussions. It studies more formally what a language must provide to support protocol engineering, and, based on the previous experiments and on this theoretical analysis, it analyzes the strengths and the limitations of ESTEREL.

6.1 Characterizing communication protocol development

In communication protocol development, one can see two groups of constraints : the one linked to the description of communication protocols, and the one

linked to system engineering. The language we use for communication protocol engineering must be able to express both groups of constraints easily. We will now enumerate and briefly describe constraints attached to each of these two topics.

6.1.1. Protocol specification characteristics

Finite State Machine (FSM) description

The control part of a communication protocol is essentially a Finite State Machine (FSM). As we showed earlier, a protocol can be completely described by a suite of conditional instructions of the following type :

When the protocol receives the event EV-x in the state ST-y,
when predicates P1 to Pn are verified,
do action A1 to An

Reactivity to external events

A communication protocol processes "events" that can be received through three interfaces :

- the interface with the protocol user (which can be the application),
- the interface with the underlying network (through which the network is accessed),
- the timer manager (which sends events on timeout).

The number of events is defined, and limited. When an event is received, it has to be processed immediately. Possible relations and interactions between events are also known. A communication protocol is consequently, based on its local behavior, a reactive system.

Atomicity of event treatment

In the current ESTEREL implementation, the treatment of a protocol event cannot be interrupted until it has completed. This is justified by the parallelism and mutual exclusion problems encountered in the case of an interruption by another event. If interrupts are allowed, the protocol has to define precisely where and when such interruption is possible. For example, it is possible to interrupt the treatment of data during the checksum evaluation if a timeout occurs to invalidate the data. But the same interruption is not allowed while updating the size of the flow control window. So, when the treatment of an event is not atomic, it is composed of a small number of atomic actions, which is essentially equivalent. This level of atomicity must be described at the protocol specification level.

Event treatment interruptions issued by a process involved in the event treatment are more relevant. For example, one could decide to stop the data treatment as soon as packet corruption has been detected. This level of atomicity need not be described in the protocol specification.

Determinism

The discussion of protocol determinism is not easy. It could be the subject of a full report. Some people consider that protocols are deterministic systems because their behavior in a particular state is always deterministic (it is linked to predicate evaluation). Others argue that, depending on resource availability, the behavior of the protocol can not be reproduced exactly with the same sequence of external events; that makes a protocol non-deterministic.

Parallelism

There are various degrees of natural parallelism in communication protocols. The problem is that, depending on the implementation environment, all these degrees of parallelism cannot always be implemented together. We have identified three different types of parallelism :

- connection level parallelism : in the case of connection oriented protocols, two events addressing different connections can be processed in parallel with no mutual exclusion problems. Within a connection, incoming and outgoing data flows can be processed in parallel, with possible effects on the connection context updating (depending on how the protocol has been designed).
- pipeline treatment in layered architectures.
- functional parallelism within the treatment of an event : some protocol functions, such as flow control and error control, can be processed in parallel. Others, such as error control and data delivery, have to be processed sequentially. There are additional cases where functions can overlap (e.g. checksum evaluation and encoding can be overlapped during the same data transfer).

Communication

Communication among parallel or sequential processes can be realized via different mechanisms. Using one of them is an implementation choice that must be made only at the end of the protocol engineering process. These mechanisms may be based on the use of shared memory, FIFO, or rendezvous (as in CSP, communication takes place at the same time as synchronization). For protocol engineering purposes, a description language must be able to express that two process have to communicate without specifying other information about the implementation mechanisms.

Synchronization

The problem of synchronization is very close to that of communication. Synchronization of processes may be required :

- before the communication of information between these processes. In this case, the communication is said to be synchronous (this is the only type of communication offered in CSP-based languages).

- to wait for the end of parallel treatment before a new process can be started. In this case, there is no explicit communication step.

In protocol description, synchronization should be expressed separately from communication so that the mechanisms can be used independently, although it should be noted that there is often a certain level of implicit interdependence.

Data processing

The problem of data processing is well known in the world of communication protocols; its most cost effective part is data transfer. It is generally admitted that data processing is not only the most expensive part of an event treatment, but it is also the task that most limits protocol performance. Data processing is typically implementation dependent; consequently, these issues must be transparent at the specification level.

Timer management

Because they are inherently reactive, reliable protocol implementations require the use of timers. Timers ensure that a protocol will not stay "too long" in the same state (waiting for an event) ; in this case, timers generate outputs which are treated as protocol events. Timers are also used to evaluate properties like the round trip time, or the latency delay.

The effect of a non adapted timer manager on performance has been shown frequently. It is important to have an efficient way to express timer constraints at the specification level, and to implement them at the implementation level.

Operating System (OS) support

As with timers, a communication protocol makes heavy use of Operating System primitives in order to allocate resources, manage parallelism, and provide facilities for process communication and synchronization. Primitives offered by classic Operating Systems like UNIX tend to considerably slow down the protocol performance. Communication protocol requirements for OS primitives are simple, and they can be implemented simply using a dedicated OS, or a real-time OS kernel. OS primitives do not appear at the specification level ; they are primitives of the implementation level.

6.1.2. Automated generation of protocols

Abstraction levels

In our approach, all application requirements will be integrated in a single automatically generated protocol description. This description will be used to verify and to implement the protocol. From a single description, different implementations should be generated (integrating local optimizations). That means the original description must have a high level of abstraction, to avoid any premature choices about how the protocol will be implemented. The same high-level specification must apply

to each stage of the development process ; it must accomodate the various levels of abstraction up to and including the final implementation. The following example shows the different abstraction levels of a specification and an implementation. The protocol function described, in natural language is "deliver incoming data in total order". This description using a natural language is a first level of abstraction. Using a formal pseudo-language, this description will become :

```
var data_to_deliver : data                  % data to deliver
var current_data: data_idf                  % identifier of the data to deliver
var last_delivered_data : data_idf          % last delivered data

if current_data = (last_delivered_data + 1)
then deliver (data_to_deliver) else wait
```

Note that data_idf, deliver, and wait are not defined. The same description at the implementation level could be (still using a pseudo-algol-like language) :

```
type buffer : array [1..N] integer
type buffer_address : pointer on buffer
type data : record [number : integer,
                    reference : integer, adress : buffer_address]

var current_data : integer                  % data to deliver
var last_delivered_data : integer           % last delivered data
var data_to_be_delivered : data

if current_data = (last_delivered_data + 1)
then deliver (data_to_be_delivered)
else save (data_to_be_delivered)

procedure deliver (var data_to_be_delivered : data)
begin
% code describing the way data are delivered to the application
end

procedure save (var data_to_be_delivered : data)
begin
% code describing how data are buffered in a chained list structure
end
```

Data structure expression

The definition of data structure is implementation-linked. At the specification level, data structure types need not be known. In the previous example, the type "data" is not defined in the specification. The advantage of this formal description of the protocol is to be unambiguous, yet remain easily readable. At the implementation

level, the "data" type is completely described by a record type. But it could have been another type (an array of byte for example) without any change on the specification, since at specification level it is identified only by name and not by structure. It is consequently important to use a description language that offers facilities to accomodate both levels of description, and also to pass automatically from the specification level to the implementation level.

Modularity

The implementation level design of a system is facilitated if a system is described in term of modules, and in term of rules (or relations) that explain when these modules are used and how they can be interleaved (parallelism, synchronization, overlapping). This approach aids in the integration of application and environmental constraints, which in turn yields the most efficient implementation.

Development and analysis environments

The basic environment required to reliably develop protocols and tools with a particular language includes a compiler, a debugger, and a simulator. An additional benefit is ease of complier modification ; it would be optimistic to believe that, for protocol engineering purpose, a language will be used indefinitely in its original syntax and semantics.

An analysis environment complements a development environment with verification and optimization tools. Having a serious analysis environment simplifies the development of tools dedicated to protocol engineering. The ESTEREL language environment includes tools which are very useful for protocol engineering.

6.2 Discussion of key ESTEREL properties and limitations

Our experiments confirmed our *a priori* reasons for choosing ESTEREL. But if the various experiments we performed using ESTEREL for protocol specification, implementation, and optimization confirmed that the language is well adapted to protocol engineering, it also revealed some limitations that suggest that the language, or the compiler, will eventually have to be modified to fit comfortably with the requirements of our approach to protocol design and implementation.

6.2.1. Issues

Properties of synchronous reactive formalisms

- It is reactive ; a reactive system has successive configurations, reached by successive transitions (or: reactions).
- It is synchronous ; each single reaction consists of the "synchronous" reception of several events (inputs), together with an instant emission of output events in response.
- It is a finite state machine description language.

Control description language

ESTEREL only provides for the description of the control part of protocols ; data types are opaque. The compiler checks the coherency between types (or names of types), but does not attach any data structure to them. A data description language like ASN1 or XDR must be used in association with ESTEREL to describe the linearized version of data structures and data manipulation primitives.

A consequence of opaque types is that every operation on data structures or variables of an ESTEREL description has to be described functionally within the module. For example, to increment a number (with type NUMBER) one would write : INCREMENT (NUMBER). NUMBER := NUMBER + 1 would not be significant as "+" and "1" are not defined as type NUMBER.

Time and Exception Handling in ESTEREL

There is no special type or signal to describe time in ESTEREL. Time is considered as any other external event, and must be described by an external signal. ESTEREL's reactivity and synchronism hypotheses are applied to the time signals just as to any other signal. That means timer management procedures are external to the ESTEREL specification of the protocol, and that they must be described using the data description language or, for example, the C language. Time signals are bound to their external primitives during the last compiling step (the implementation generation step).

However, time in ESTEREL is multiform: any signal may be processed as an independent "time unit", so that the time manipulation primitives can be used uniformly for all signals. In ESTEREL, it is easy to write:

await 3 meters:
do
    <task>
watching 100 WHEEL_REVOLUTION.

This is one of the strengths of the ESTEREL programmming style: physical time can also be treated as a standard signal. But since all timing constraints are based on events/signals counts, it is not possible to specify an absolute time.

ESTEREL also has a powerful exception mechanism which is fully compatible with concurrency, unlike in asynchronous programming language. A watching statement defines a limit for the execution of its body; if the body terminates strictly before the limit, the whole watching statement terminates normally. If the body is not terminated when the limit occurs, the body is instantly killed without being executed at that time and the watching statement terminates. With respect to the synchrony hypothesis, the body does not take any time to execute. If the body describes an asynchronous task wich can take time (with an exec statement), this task may be interrupted and a kill signal is generated.

Readability

As can be seen from the appendices, the ESTEREL specifications are quite easy to read, assuming that the specification is written following some protocol description rules (which will be described in Appendix B). The only reproach that can be made to ESTEREL is the structure of declarations that have to be repeated in each submodule, even though they are not used by the compiler (the compiler analyzes the module declarations only once, in the main module). Moreover, adding a description of the protocol data unit using a specialized language will provide a complete and powerful environment for protocol formal specification.

6.2.2. Limitations

The limitations of ESTEREL in the context of communication protocol engineering need more attention, even if they are not of a nature to make protocol description and implementation impossible. These limits are more to be seen as problems that have to be discussed with the ESTEREL research group in order to understand what solutions can be found, or whether suitable modifications or extensions to the compiler might be possible.

Compilation: modularity and optimization

Separate compilation of ESTEREL modules is impossible. The main consequence of this is to limit overall flexibility while designing and customizing the protocol description. The ESTEREL compiler optimization criteria are not those we would prefer to use in the context of ILP. Although evident in the specifications, real parallelism does not exist in ESTEREL, and most of the compiler activity is to arrange (i.e. sequence) the different modules of the ESTEREL description in order to :

- minimize synchronization and wait between modules.
- execute the modules in an order that preserves atomicity.

For example, if one writes :

```
await signal_start do
    Do
        process P
    watching signal_S_wrong
end
||
await signal_start do
    evaluate_signal_S
    if TRUE then emit signal_S_right else emit signal_S_wrong
end
```

that means that the evaluation of the signal S and the execution of P should be done simultaneously ; but that the process P has to be interrupted and stopped immediately if the signal S is corrupted.

The ESTEREL compiler will choose to process the evaluation of the signal S first, and not to start the process P if the result of the evaluation is not TRUE. This is, from a specification point of view, completely different, and it in fact negates the performance gains that we hope to achieve. Consequently, one of the main activites in the protocol compiler design will be to adapt the ESTEREL compiler to support our implementation and optimization constraints.

Verification exhaustiveness

Verification is not exhaustive in ESTEREL. Since ESTEREL describes only the control part of the communication protocol, the verification is also limited to the control part. In fact, the ESTEREL verification tool allows verification of specific paths in the protocol automaton. For example, one can verify that there is no deadlock in a protocol specification, or that a packet of type A will be discarded if received after an event of type T. But it is impossible, using the pure ESTEREL verification tool, to verify that data will be ordered before delivery to the application (sequence number is carried by a signal, and signal types are not known in ESTEREL). These verification facilities can however be complemented with a simulation tool provided with the ESTEREL environment.

Despite these limitations, we consider that ESTEREL and its test environment are a good compromise for verification tools for the following reasons:

- The LOTOS and ESTELLE verification tools, which are today state of the art, are theoretically exhaustive. They are using a technique called "combinatorial explosion" which increases spectacularly the protocol description ; so spectacularly that they generally become too complex to be verified. The only practical verification approach in these environments for the protocol we describe would be the one proposed by Gerard Holzmann in [Holzmann 91], which is in fact not exhaustive.
- In our automatic implementation approach, the protocol will be designed automatically, using pre-verified elementary communication blocks. The only thing that will have to be verified is the protocol control automaton ; what can be done easily with the ESTEREL verification tool.
- An ESTEREL module can be compiled in the BLIFF formalism [Touati 93] (using the release 4_41 of the compiler). There are very powerful verification tools provided with BLIFF.

Relation semantic

The semantic of relations is not sufficiently powerful in ESTEREL. Modifying this semantic would help in minimizing the number of rules that explain how to write a protocol description in ESTEREL (see Appendix C) ; in simplifying the customization of the protocol specification to the application requirements ; and in increasing accuracy in the protocol description.

Two types of improvements can be proposed :

- **Relations between external and local signals**. In the current release of ESTEREL, the definition of relation is used to constraints the possible relations between external signals. Relation between external and local signals would be very useful in case of protocol customizing, where local signals are used to describe application requirements. Such a type of relation would simplify the protocol customizing step, and make the description of the protocol more clear.
- **The "OR" operand between signals**. Consider two different types of events, DT for Data, and ED for expeditive data ; they will be represented as external input signal on the protocol description. Both events carry a NUMBER attribute, which will also be described as an external input signal. Using ESTEREL the relations between these three events can be described as follows :

  DT # ED ; DT => NUMBER ; ED => NUMBER ;

  which only partially describes the real relation between these three events. Another relation is missing :

  NUMBER => (ED or DT)

  This relation cannot be described using ESTEREL. ED and DT numbers will have to be identified by other means.

7 Conclusion

We have now verified that ESTEREL is a good candidate for protocol description and implementation. However, we have not yet chosen the data description language we will use with ESTEREL, and we have not yet derived the detailed design of the protocol compiler (as depicted in Figure 1).

The possible choices of a data description language are very limited. Languages we have identified in this report as "semi-formalisms" are rare. An early evaluation indicates that ASN.1 and XDR are the most serious candidates. We believe that making a choice at this step of the project would be premature. For the moment, we will continue to experiment with C as a data description language, even if this high-level assembler is not very appropriate.

On the other hand, the analysis of ESTEREL, and the experiments we performed, have helped us to have a clearer idea of the structure of the Protocol Compiler we would like to design (see Figure 8). In a first step, a subset of a communication module library written in ESTEREL will be customized either by hand or (semi-automatically). This library contains the specifications corresponding to the requirements of "sample" applications and may be "refined" to meet the specific requirements for a specific application. If the application requirements are completely different from what is in the library, the corresponding specifications should be written

from scratch. The result of the customization step cited above is a protocol specification (in ESTEREL) describing the desired communication subsytem for the application.

In a second step, the ESTEREL specifications are fed into the ESTEREL compiler which will generate a sequential automaton. It is also possible to use knowledge of the application requirements to control the configuration of the generated protocol automaton at this step, although it is preferable to express the ApRe in the first step to either modify or select specific modules from the library. Although validation is not a main goal of our design, tt is possible to verify the generated automaton using the ESTEREL environment tools described in section 4.3; this might require a refinement of the specification and an iteration of the generation procedure.

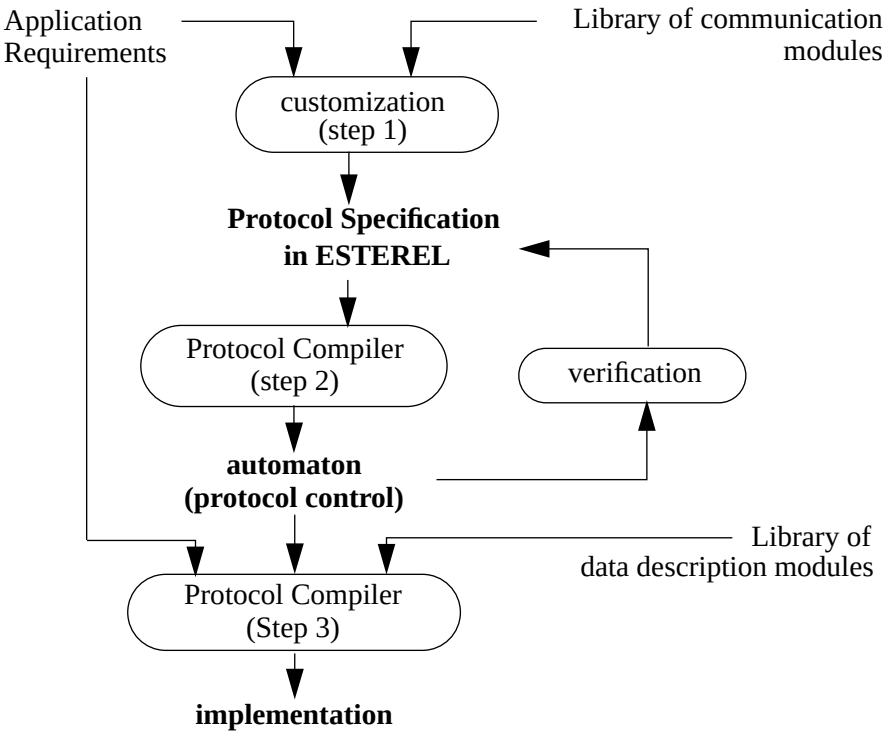


Figure 8 : Structure of a Protocol Compiler based on ESTEREL descriptions

In the final step, the implementation of specific communication functions (containing both the algorithms and the data structures corresponding to the data manipulation part) is linked into the protocol automaton in order to produce an integrated implementation of the communication subsystem in the target language. Further

experience is needed to assess the suitability of ESTEREL for a final "production" implementation of such a communication subsystem.

Our experimental results to date have concentrated on understanding the properties and features of ESTEREL, and on validating our original assessment of its suitability for the specification and development of communication protocols. Although we have not specifically concentrated on the equally important aspects of performance, preliminary results suggest that there are no inherent obstacles posed by the ESTEREL implementations; this leads us to believe that with proper adjustment and tuning we can achieve sufficient control over the performance aspects to facilitate our future work with ALF and ILP. Furthermore, the additional support provided by the associated development environment will facilitate our prototype development process as well as the validation of any experimental protocols.

8 References

[Abbot 92] M.B. Abbot, L.L. Peterson, "A Language-Based Approach to Protocol Implementation", IEEE/ACM Transactions on networking, September 1992.

[Airiau 90] R. Airiau, J.M. Berge, V. Olive, J. Rouillard, "VHDL, du langage à la modelisation", Presses Polytechniques et Universitaires Romandes, CNET/ENST 1990.

[Belina 89] F. Belina, D. Hogrefe. "The CCITT specification and description language SDL". Computer Networks and ISDN Systems. North-Holland. Vol. 16, No. 4, pp. 311-341. March 1989.

[Berry 91] G. Berry and G. Gonthier. "Incremental development of an HDLC entity in Esterel". Comp. Networks and ISDN Systems. Vol. 22, pp. 5-49. 1991.

[Berry 92] G. Berry and G. Gonthier. "The Esterel Synchronous Programming Language: Design, Semantics, Implementation". Journal of Science Of Computer Programming. Vol. 19, Num. 2, pp. 87-152. 1992.

[Birell 84] A. D. Birell and B. J. Nelson. "Implementing Remote Procedure Call". ACM Transactions on Computer Systems. pp. 39-59. February 1984.

[Bochman 87] G. V. Bochmann. Usage of Protocol Development Tools : The result of a Survey. IFIP conference on PSTV VII, H.Rudin and C. H. West editors. Elsevier Science Publishers. 1987.

[Bochmann 90] G. V. Bochmann. "Specification of a simplified Transport Protocol Using Different Formal Description Techniques. Computer Networks and ISDN Systems. North-Holland. Vol.18, pp 335-377. 1990.

[Boudol 89] G. Boudol, V. Roy, R. de Simone, and D. Vergamini. "Automatic Verification Methods for Finite State Systems". In the proceeding of CAV Grenoble : Process Calculi, from Theory to Practice: Verification Tools. Vol. LNCS 407. 1989.

[Boussinot 91] F. Boussinot and R. de Simone. "The Esterel Language". Proceedings of the IEEE, special issue on "Another Look at Real Time Programming". Vol. 79, pp. 1293-1304. 1991.

[Castel 94] C. Castelluccia and W. Dabbous. "Modular Communication Subsystem Implementation using a Synchronous Approach". To appear in Usenix Symposium on High Speed Networking. Oakland, August 1994.

[CCITT 87] CCITT "Specification and Description Language", Recommandation Z.100, Contribution Com X-R15-E, 1987.

[Chrisment 94] I. Chrisment and C. Huitema. "Remote Operation System Tailored to Application Requirements". IFIP International Conference ULPAA '94. Barcelone. June 1994.

[Clark 89] D. D. Clark, V. Jacobson, J. Romkey and H. Salwen. "An analysis of TCP processing overhead". In IEEE Communications Magazine, June 1989.

[Clark 90] D. D. Clark, D. L. Tennehouse. Architectural Considerations for a New Generation of Protocols. Proceedings of ACM SIGCOMM. 1990.

[Corbin 91] J. R. Corbin. "The Art of Distributed Applications". SUN technical Reference Library. Springer-Verlag Editor. 1991.

[Danthine 92] A. Danthine, "A New Transport protocol for the Broadband Environment", invited paper at IFIP Workshop on Broadband Communication, Estoril (Portugal), January 20-22, 1992.

[Diot 94] C. Diot, "Comment spécifier un protocole ou un ensemble de protocoles en ESTEREL", Internal Report (in French language), INRIA Sophia Antipolis, RODEO project. 1994.

[Garavel 90] H. Garavel, J. Sifakis, "Compilation and Verification of LOTOS Specifications", Proceedings of the 10th PSTV IFIP Conference, Ottawa (Canada) 1990.

[Harel 87] D. Harel. "StateCharts: A Visual Approach to Complex Systems". In SCP Journal. 1987.

[Hoare 79] C. A. R. Hoare, "Communicating Sequential Process", Communication of the ACM, April 1979.

[Holzmann 91] G. J. Holzmann. " Design and Validation of Computer Protocols". Prenctice Hall International Edition. 1991.

[Hoschka93] P. Hoschka and C. Huitema. "Control flow graph analysis for automatic fast path implementation". In Second IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems, Williamsburg, Virginia, September 1993.

[http 94] Tk Meije environment MAN pages. WWW server http://zenon.inria.fr:8003/meije/meijetools.html. Sophia Antipolis. 1994.

[Huitema 91] C. Huitema. "MAVROS : Highlights on an ASN1 Compiler". INRIA. Internal working paper. 1991.

[Hutchinson91] Norman C. Hutchinson and Larry L. Peterson. "The *x-Kernel*: An Architecture for Implementing Network Protocols". IEEE Transactions on Software Engineering, Vol. 17, No. 1. Jan 1991.

[IEEE 87] IEEE Computer Society "IEEE Standard VHDL Language Reference Manual", IEEE standard, P.1076, 1987.

[IEEE 91] Proceedings of the IEEE, special issue on "Another Look at Real Time Programming".Vol. 19, Num. 2. 1992.

[ISO 86] ISO/OSI "ESTELLE - A Formal Description Technic Based on Extended State Transition Model", Juillet 1986.

[ISO 87a] ISO/OSI "LOTOS - A Formal Description Technic Based on the Temporal Ordering of Observational Behavior", ISO standard 8824. Geneva, 1987.

[ISO87b] ISO/OSI "Specification of Abstract Syntax Notation One (ASN 1)", Geneva, July 1987.

[Madeleine 89] E. Madelaine and D. Vergamini. Auto}: A Verification Tool for Distributed Systems Using Reduction of Finite Automata Networks". Proc. FORTE'89 Conference. Vancouver. 1989.

[Merlin 76] P. M. Merlin. A Methodology for the Design and Implementation of Communication Protocols. IEEE Transactions on Communications. Vol. COM-24, No. 6. June 1976.

[Partridge 92] C. Partridge and S. Pink ; "An Implementation of the Revised Internet Stream Protocol (ST II). Internetworking : Research and Experience ; Vol. 3, pp. 27-54, 1992.

[Roy 90] V. Roy and R. de Simone. "Auto and Autograph". Proceedings of CAV ' 90. R. Kurshan Editor. June 1990.

[Smith 83] F. D. Smith, C. H. West. Technologies for Network Architecture and Implementation. IBM Journal on R&D. Vol. 27, No. 1. January 1983.

[Touati 93] H. Touati and G. Berry. " Optimized Controller Synthesis Using Esterel". Proceedings of the International Workshop on Logic Synthesis IWLS '93. Lake Tahoe. 1993.

[Zhang 89] L. Zhang. "A New Architecture for Packet Switching Network Protocol". MIT PhD Thesis. 1989.

Appendix A Communication Subsystem Implementation With ESTEREL: A Case Study

In this appendix, we describe the implementation of a data transfer protocol using ESTEREL. The specification of the protocol that we implemented is very similar to that of the TCP protocol (we omit however the connection establishment and termination phases). We address how to design the building blocks and how to combine them to generate the required protocol. The goals of this case study were to test the validity of our approach and to give an insight on building-block contents.

Reusability and flexibility are our main goals. The building blocks must be designed so that they are meaningful to the designers and so that changes in the protocol specification only induce local changes in the architecture and the code.

Communication subsystems are structured in three parts :

- the send module, that handles outgoing frames
- the receive module, that processes incoming frames
- the connection module, that handles connection variables and states

Each of these components can be decomposed into finer grain modules. The protocol functions are considered as atomic and their dependencies are defined by their input, sensor and output signals.

B.1 Building Blocks Description

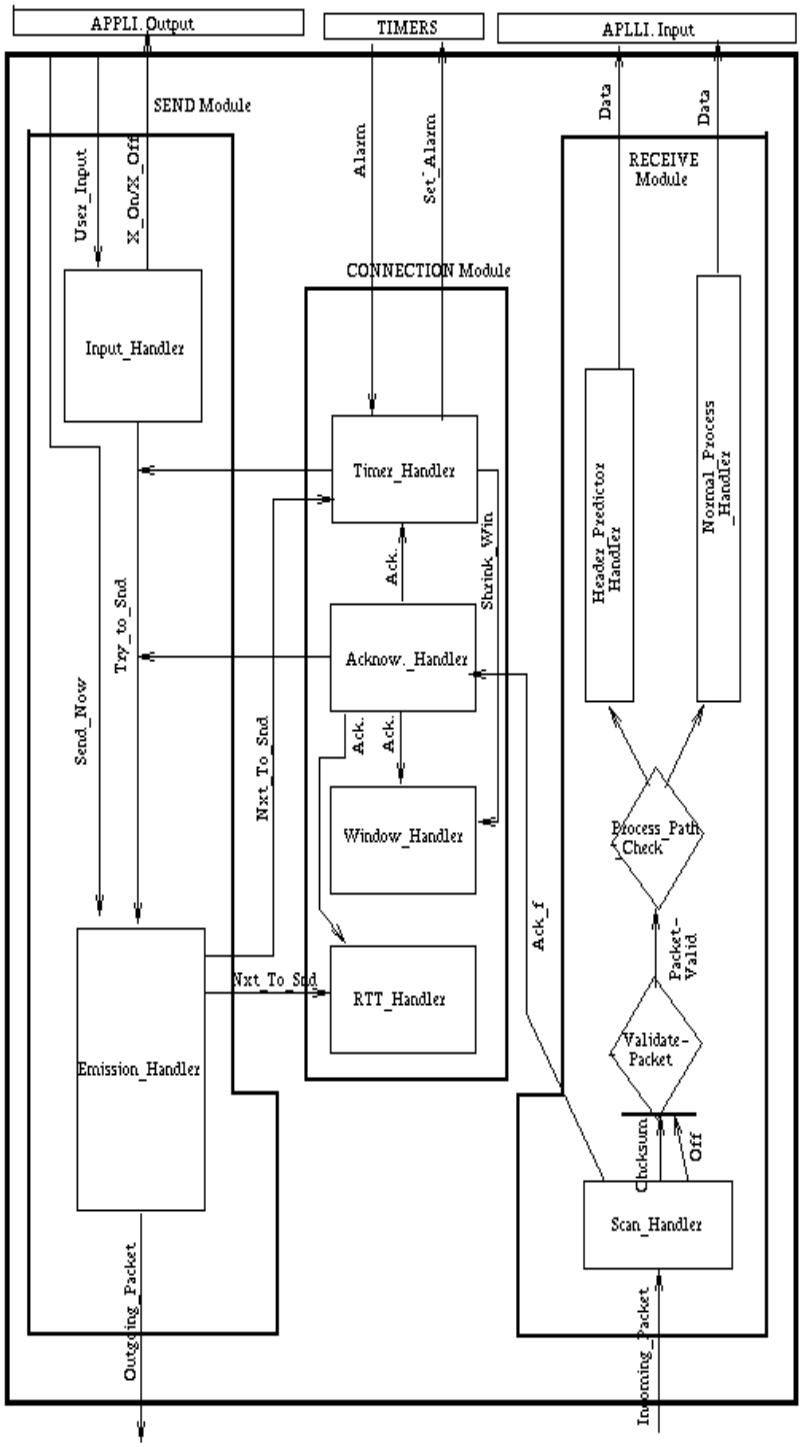
We implemented this example incrementally in order to satisfy the modularity property that we are aiming for. We started from a simple protocol and added modules step by step until we accomplished all required functionalities. Following the precepts of Object Oriented Programming we followed the rule *one functionality-one module*. An overview of the whole subsystem is shown on next page. For clarity purposes, only the principal modules have been described and displayed. Our data-transfer protocol is structured in three main concurrent modules :

The Send Module

This module is composed of two concurrent submodules:

- The *Input_Handler* module receives data from the application. If enough space is left in the internal buffer, they are copied to it and a "Try-to-Snd" signal is broadcast, otherwise incoming data are unauthorized until an acknowledgment signal frees up some buffer space.
- The *Emission_Handler* module transmits packets on the network. It can received two types of inputs: the "Try-to-Send" input will try to send packets by evaluating the congestion window size, the silly window avoidance algorithm and the

number of bytes waiting to be sent; it may then send one or several packets. The "Send-Now" input will force the sending of a packet even though the regular sending criteria are not satisfied (this is used to acknowledge data for example). If the module decides to send packets, the checksum is performed and the header is completed.


APPLI. Output
TIMERS
APLLI. Input
SEND Module
RECEIVE Module
CONNECTION Module
User_Input
X_On/X_Off
Alarm
Set_Alarm
Data
Data
Input_Handler
Timer_Handler
Header_Predictor Handler
Normal_Process _Handler
Ack.
Shrink_Win
Try_to_Snd
Acknow._Handler
Send_Now
Nxt_To_Snd
Ack.
Ack.
Window_Handler
Process_Path Check
Packet-Valid
Nxt_To_Snd
RTT_Handler
Ack_f
Validate-Packet
Emission_Handler
Checksum
Off
Scan_Handler
Outgoing_Packet
Incoming_Packet

The Receive Module

This module processes the incoming packets. It is composed of several submodules that can be executed concurrently:

- When a packet is received, its header is scanned by the *Scan-Handler* module and all the header fields are broadcast.
- The *Validate-Packet* module waits for "Checksum" and "Off-bit" signals to validate the incoming data (checksum and Off-bit field conformance tests are performed). If the packet is non-valid it is rejected at this point, otherwise the data are processed (the acknowledgment field is processed concurrently by the *Connection module*).
- A test is performed (in the *Process-Path-Check* module) to decide whether the packet should follow the "header prediction" path (*header-predictor* module) or normal path (*Normal-Process-Handler* module). In the normal path, the data are processed and delivered to the application; the flow control parameters are updated. In the fast path, two cases are possible: (1) A pure acknowledgment packet is received. Buffer spaces are freed up and the sequence number of the next data to be acknowledged is updated. (2) A pure in-sequence data packet is received. The data is directly delivered to the application. The sequence number of the next expected data is updated.

The Connection module

This module is composed of the following submodules that are executed concurrently: the *RTT_Manager* module, the *Timer_Handler* module, the *Window_Manager* module and the *Acknowledgment_Manager* module.

The *RTT_manager* module computes the round trip time of the connection. When a packet is emitted, no other packet belonging to this connection is in transit and we are not in a retransmission phase then a timer is started. When this packet is acknowledged, then the timer is stopped and a new RTT value is computed.

The *Window_manager* updates (concurrently) the congestion window and the send window (corresponding to an evaluation of the space left in the receiving buffer of the remote host). When an acknowledgment signal is received (from the *Acknowledgment_Manager* module), the *Window_manager* increases the congestion window either linearly or exponentially according the slow-start algorithm, and the sending window is either update to the value of the ``Win" field (emitted by the *Scan_Handler* module) or decreased by the number of bytes acknowledged. If a signal corresponding to a window shrink request (from the *Timer_Handler* module) is received, the congestion window is set to 512 bytes (value of the Maximum Segment Size). The *Acknowledgment_Manager* module handles the acknowledgment information received from the incoming packet. If the acknowledgment sequence number (which corresponds to the last bytes received by the remote host) received is greater

than the latest bytes sent or less than the last already acknowledged bytes then it is ignored, otherwise the acknowledged value is updated.

The *Timer_Handler* module manages the different timers. When a packet is emitted and no other packets are in transit, the Retransmission timer is set. When this packet is acknowledged and other packets are in transit it is restarted, otherwise it is reset. If the timer expires before the acknowledgment of this packet is received, retransmission and window-shrink requests are emitted.

All the modules just described have been implemented in ESTEREL and compiled into an automaton.

B.2 Modularity issues

The modularity of our approach has been demonstrated and illustrated by our case study. In fact we implemented our protocol incrementally and ran it at each step to test its correctness. As a result of this programming style, the program obtained is very modular; we can easily remove or exchange modules. For example by simply discarding the *Window_Handler* submodule we synthesize a sliding window data transfer protocol (with a window of fixed size). If we set the window size to one packet, we implement a Stop-and-Go protocol. The window flow control can also be exchanged by a rate control mechanism by simply modifying the *Emission_Handler* module. The isolation of the different functionalities into separated and concurrent modules leads to very modular structures.

B.3 Application Programming

The protocol generated was then used to implement a file transfer application in order to validate the conformance of our protocol implementation with its specification. In our model, the protocol is implemented at the user level and linked with the application. The protocol is completely integrated with the application; actually the protocol is a task of the application. The application and the protocol can then share the same memory space. We implemented this interaction using Light-Weight Processes which allow the definition of multiple tasks within a single UNIX process and provide light-weight context switching between tasks.

Our application (contained in one process) is composed of 4 tasks :

- the application task : This is the task implementing the application. It notifies the protocol task when some data have to be sent.
- the I/O task : This is the task responsible for the incoming packets. When a new packet from the network is received, the I/O task notifies the protocol task that an incoming packet is waiting to be processed.
- the protocol task : This task processes the outgoing data (coming from the application) and the incoming frame (coming from the I/O task) according to the syn-

thesized protocol. It performs what we called earlier the ESTEREL ``Execution Machine".

- the timer task : This task manages the timers used by the protocols and the application.

All these tasks have the same execution priority, except the I/O task which has an higher one to avoid loss of incoming packet. A task with higher priority may preempt the others. Tasks with the same priority are executed on a round-robin discipline: a task is executed until is done or blocked on an I/O, the following task on the list is then executed.

All these tasks synchronize with each other. When a task has nothing to do, it sleeps. When another task needs its assistance, it is woken up. For example, when the protocol task has neither outgoing data nor incoming packet to process, it sleeps. When a packet arrives, the I/O task awakes it. The synchronization is then very natural.

Appendix B The ROSTAR multi-layered compilation approach

- The automated generation of the communication part of an application module is implemented by a packaging system which regroups three parsers/compilers (cf Figure 9):
- the MAVROS ASN.1 compiler [Huitema91]: From the data specification in ASN.1 and from the module control specification in *ASN.1 extended*, MAVROS generates the encoding and decoding routines in C and a file MODULE.macros. This file contains some information related to the control/synchronization macros.
- MACROS Analyzer: Our parser takes the file MODULE.macros as input, and analyzes it so as to generate the corresponding ESTEREL code and two interfaces: one interface between the ESTEREL module and the application module and another interface between the ESTEREL module and the transport automaton. ESTEREL describes a synchronous world where reactions are instantaneous, while the application and network describe asynchronous worlds. So these interfaces permit the integration of both worlds. The ESTEREL generated code includes three main modules:

  - **a reception module:** which reacts when the network is ready to receive and which asks for unmarshalling messages. According to the type of received messages, a given signal is sent to the specific application module.
  - **an emission module:** which asks for marshalling messages and which buffers them until a signal from the network specifies that the network is ready to send. Then the message is sent.
  - **a specific application module:** which depends on the control specification. Presently, our prototype generates only the control automaton, not the transport automaton; it directly uses the UDP/IP transport via the socket interface.
- the ESTEREL compiler [Berry92]: This component generates the control automaton in C.

A general makefile the produces the module code for the target machine.

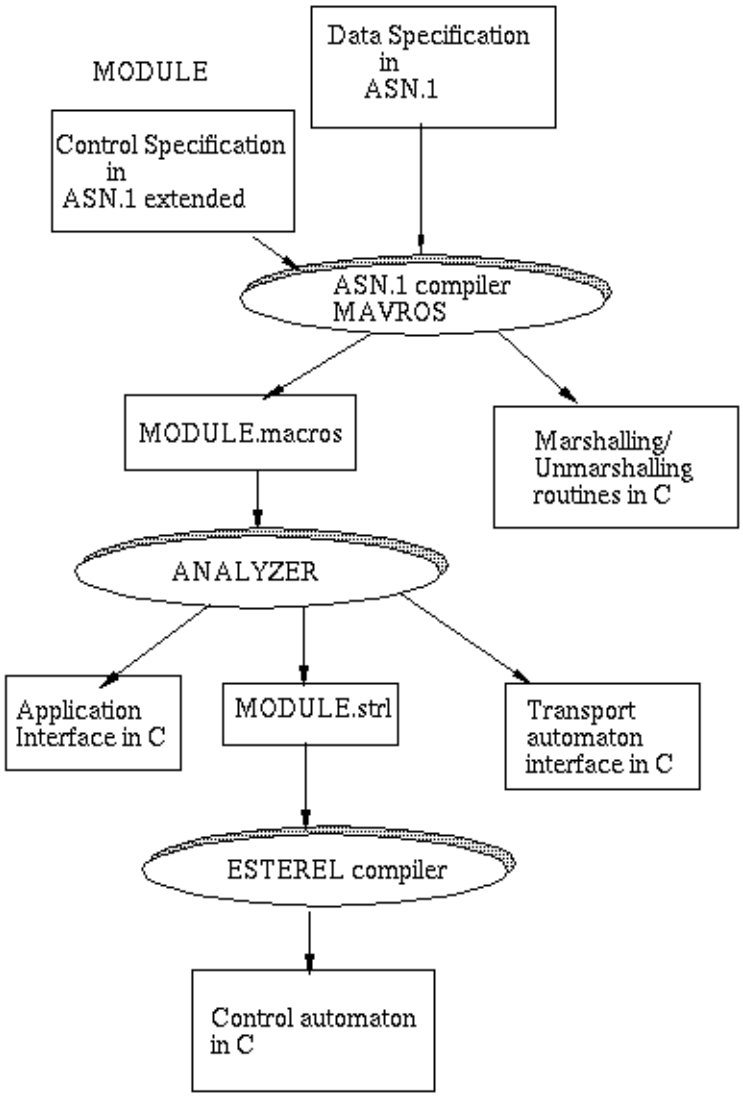


Figure 9 : Multi-layered compiler structure

Appendix C Constrained Use of ESTEREL: Programming Rules

Most of the rules we briefly describe in this subsection have been established during the various experiments we made. They will not be justified individually in this report. These rules try to :

- make the description more readable ;
- minimize the complexity of the protocol automaton (in some cases, non reachable transitions can be created by the compiler)
- preserve implementation choices at the specification level.

There is no case where these rules really reduce the power of protocol description in ESTEREL. These rules are :

- A module that starts at an (ESTEREL) instant t must end in the same instant.
- Each protocol event type should be described in ESTEREL by a pure external signal (not valued). Generic information carried by these protocol events are represented using valued external signal. These signals can be either input or output (or both).
- Each module must start by an "await" construction on one of the input signals that represent the protocol event types, or possibly on a local signal.
- In an "if condition then action else other-action" construction, "action" and "other-action" must include emissions of local signals.
- Each significant parameter must be represented using a local or external signal. Shared variables are prohibited.
- The value of a signal is unique during an instant. If it has to be updated for the following instant, the "await tick" instruction must be used.
- The notion of state must not be represented by a valued signal in ESTEREL.
- Use the instruction sequence "loop await S" instead of "every S».