On the improvement of Estelle based automatic implementations

O. Catrina [a*], A. Nogai [b]

[a] *Institut National des Télécommunications, 9 Charles Fourier 91011 Evry France. e-mail: tavi@hugo.int-evry.fr*

[b] *Politehnica University of Bucharest, 1-3 Bv. I. Maniu, Bucharest 77202 Romania. e-mail: anogai@elcom.pub.ro*

Key words: Automatic protocol implementation, Estelle, transport protocol

Abstract: The main challenges for the tools which derive implementations from formal descriptions are to enhance the efficiency and facilitate the integration in various implementation contexts. We study in this paper the Estelle based implementations. Using the Estelle Development Toolset (EDT), we obtained a realistic implementation of a complex transport protocol, XTP 4.0, functionally comparable with the hand-coded reference implementation. This offered an experimental ground for a survey concerning the automated implementation. Based on runtime measurements, analysis of the Estelle model and of the tool's support, we studied the factors influencing the performance and the solutions to improve it.

1. INTRODUCTION

The use of the formal description techniques (FDT) in the development of the telecommunication systems is continuously extending. For protocol design, validation and test generation, the advantages of the FDT-based methods became obvious. The main challenge remains the complexity of the systems.

However, the ultimate goal of the protocol development process is to obtain the implementation. The automated derivation from the formal specification can ensure that the implementation complies with the specification and can greatly reduce the development and maintenance effort. Therefore, the

---

[*] On leave from Politehnica University of Bucharest, Romania. This work has been partially supported by the KIT-IDEMCOP action.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35394-4_29

S. Budkowski et al. (eds.), *Formal Description Techniques and Protocol Specification, Testing and Verification*

© IFIP International Federation for Information Processing 1998

adequate support for implementation, as part of a consistent development methodology, can be the cutting edge of an FDT-based tool. Besides the conformance, the implementation must match the efficiency requirements of the application (runtime properties, code size). Although tool support has been available for a long time, the applications are still limited. The main cause is the relatively modest efficiency, compared with the current hand-coding methods. Furthermore, part of the implementation often cannot be obtained automatically: operations not supported by the specification language (e.g., checksum computation, packet encoding and decoding) or not detailed in the formal specification ("local matters" in the protocol definition, e.g., related to resource management). Also, a non-negligible effort is often necessary for integrating the implementation in a particular context (e.g., to interface it with other components, implemented by hand).

The aim of our study was to investigate these difficulties, by obtaining a realistic implementation from an Estelle specification, using EDT (Estelle Development Toolset) [11]. The Xpress Transport Protocol (XTP) 4.0 was an ideal example [15]. XTP 4.0 is a challenge for the implementer: it covers the functionality of existing protocols (TCP, UDP), includes new features needed by modern applications (e.g., multicast) and can accommodate various communication paradigms, media requirements and network properties. We already had a (virtually) complete Estelle specification, verified by simulation [3]. A hand-coded C++ implementation of the protocol, SandiaXTP, was available for comparison [14]. It runs on UNIX machines, in user space, on top of IP or UDP, and provides an application programming interface (API) adapted to the XTP extended functionality.

We used EDT to implement XTP 4.0 from the Estelle specification, for the same environment. The implementation, called E-XTP, provides full XTP functionality to real UNIX applications, can inter-operate with SandiaXTP and has a similar API (E-XTP is an enhanced version of the implementation presented in [4]). Section 2 introduces the EDT implementation model and section 3 contains an overview of the resulted XTP implementation.

The derivation of E-XTP offered a valuable experimental ground for our survey concerning the improvement of the Estelle based automated implementation. We made a joint analysis of the runtime measurements, of the Estelle model and of the tool's runtime environment, to identify the factors affecting the efficiency and to evaluate their influence. We studied pragmatic improvement solutions, including the integration of efficient hand-coding techniques, implementation aware specification styles, optimisation of the tool's libraries. The results are summarised in section 4, structured in two parts: communication issues and system management. Part of the solutions have been experimented during the derivation of the XTP implementation, others are currently being studied for future integration in EDT. Conclusions and further work are presented in section 5.

2. EDT AUTOMATED IMPLEMENTATION

2.1 The C code generation

EDT maps an Estelle system module instance (SYSTEMPROCESS or SYSTEMACTIVITY) to a process of the target operating system and implements the children modules as procedures. It generates a separate program for each body of an Estelle system module in the specification [12].

First, the Universal Generator tool (Ug) splits the original specification, e.g., *spec.stl*, in several separate specifications, one for each system module body, e.g., *mb_k.stl*. Each specification *mb_k.stl* contains a system module from *spec.stl*, embedded (as unique child) in a new Estelle system module, automatically generated by Ug. The role of the embedding module is to transfer messages between the original system module and the environment.

Next, for each specification *mb_k.stl*, the Estelle to C compiler (Ec) generates a C program and a *make* file. The C code generated by Ec is independent of the target implementation platform. Moreover, the same C code is used to obtain the simulator program, with the EDT simulation library, and the implementation, with the EDT implementation library. One can thus be confident that the implementation reproduces the behaviour verified by simulation.

We can continue either by immediately producing a prototype of the distributed system, or by further refinement, to obtain a final implementation. For the first case, EDT can automatically distribute the system's components on user selected hosts. The set of Estelle system module instances and communication links from *spec.stl* is mapped on a set of processes communicating via TCP/IP. Alternatively, we can adapt the implementation to a particular context and improve its efficiency, by customising the primitives in the tool's libraries. The XTP implementation was obtained using the second approach.

2.2 The interface with the environment

The embedding module added by Ug is the interface between the contained module and the environment. Each external interaction point of the original module is attached to an internal interaction point of the embedding module. The transitions of the added module perform the adaptation between the Estelle message based communication and any inter-process communication mechanisms provided by the target operating system (e.g., sockets, message queues, signals, etc.). For this purpose, the EDT implementation library offers four generic primitives, *mxinit*, *mxwhen*, *mxoutput* and *mxsend*, used by Ug as shown in this example:

```
INITIALIZE
  BEGIN
    mxinit;          { IPC mechanism(s) initialization }
  END;
```

```
TRANS {Outgoing interactions processing: for each interaction point and type}
  WHEN s_p_sap.Data_req
  BEGIN
     mxsend(1, 0); { send Data_req using an IPC mechanism }
  END;            { s_p_sap identifier =1, Data_req interaction identifier =0 }
{ ... and the transitions for the other outgoing interactions }

TRANS {Incoming interactions processing (2 interaction points in this example)}
  ANY x : 0 ..1              { for x = any of the interaction points }
  PROVIDED mxwhen(x)         { a message is received for interaction point x }
  BEGIN
     mxoutput(x);            { output the corresponding Estelle interaction
  END;                       { to the interaction point x }
```

This approach avoids any modification of the original Estelle specification and offers maximum flexibility for implementing the interface. The user can choose interface functions from the tool's library or develop customised versions. The restriction is that the IPC mechanisms used for input events must match those known by the kernel function which detects the events, described in the following.

2.3 The dispatcher function

The dispatcher function, provided by the EDT library, is the core of the process which implements an Estelle system module. It contains an infinite loop, in which the process waits for an event, selects the fireable transitions (system management phase) and then fires them (execution phase).

The process blocks waiting for the arrival of incoming messages from the environment or the expiration of running timers (corresponding to enabled delayed transitions). The detection of an event wakes up the process, which starts to execute the loop. A timer management function updates the list of running timers and marks the delayed transitions which become fireable. Then, a set of transitions is selected and fired. The set may contain several transitions, for the SYSTEM-PROCESS attribute, and a single transition, for the SYSTEMACTIVITY attribute

The process continues to run as long as fireable transitions exist, i.e., until it achieves the treatment of the event which woke it up and those which occurred in the mean time. Eventually, the process blocks waiting for another input or time-out event. During a run, the process fetches (at most) one incoming message from each input port and achieves their treatment before letting other messages in. Therefore, an implicit flow control on incoming messages is provided. This is a useful property for an implementation, which is not guaranteed by the Estelle semantics, due to the asynchronous communication model with unlimited queues. As a side effect, the parallelism between the Estelle sub-modules is reduced.

3. XTP AUTOMATIC IMPLEMENTATION

3.1 Overview of the specification and the implementation

The goal of the XTP design was to obtain a connection oriented transport protocol with extended functionality and flexible configuration, adaptable to the fast evolution of the networks and the applications (high speed networks, distributed applications with transactional, multipeer and multimedia communications). The protocol kernel is a toolkit of mechanisms with orthogonal functionality. They can be independently enabled, to obtain combinations permitting various communication paradigms. The protocol definition [15, 2] is not accompanied by a transport service definition. It only suggests how to use the toolkit to obtain a basic set of service types: unicast and multicast connection oriented or datagram service, unicast transaction service, etc.

We adopted an implementation context similar to that of SandiaXTP [14], the reference XTP 4.0 implementation, hand-coded in C++: UNIX environment, UDP/IP sockets at the data delivery service interface, UNIX domain sockets and XTP-specific application programming library at the transport service interface. Figure 1 shows E-XTP and SandiaXTP running on hosts connected to Internet. Figure 2 is a closer look at a host running E-XTP, with the XTP process providing connections for 2 transport service user processes.

The block XTP corresponds to the original XTP Estelle specification. The block XTPI is the embedding Estelle module produced by Ug. The Estelle module XTP describes an XTP entity. It only performs management tasks. For each communication, an XTP entity keeps a state record, called a *context*. A communication (any type of service) requires an *association* of contexts from the participating XTP entities. A CONTEXT module is the state machine which controls the activity at one association endpoint. The CONTEXT instances are dynamically created by XTP, for the duration of an association. The module PACKET_MNGM is the interface with the underlying connectionless network service. It submits outgoing packets to the network layer and decodes incoming packets and routes them to the destination CONTEXT instances.

The Estelle specification covers almost completely the XTP features presented in the informal specification. The mechanism for allocating CONTEXT instances to users, a "local matter", was not specified in Estelle (it was left to the C primitives of the interface). An interaction of the transport service interface just carries the index of the source or destination CONTEXT instance. The specification remained of moderate size (8000 lines), compared with the rich XTP functionality, by exploiting the XTP design principles (mainly for multicast) and by using a parametric transport service. The Estelle support for complex algorithms was essential for covering the XTP functionality, e.g., for managing the receiver group state in a multicast transmitter.

The core of the implementation consists of the C code generated by the Estelle compiler from XTPI (20000 lines) and the EDT library functions

which implement the Estelle model (inter-module communication, transition selection, etc.). The size of the executable code is 210 kilobytes, similar to that of SandiaXTP 1.5.

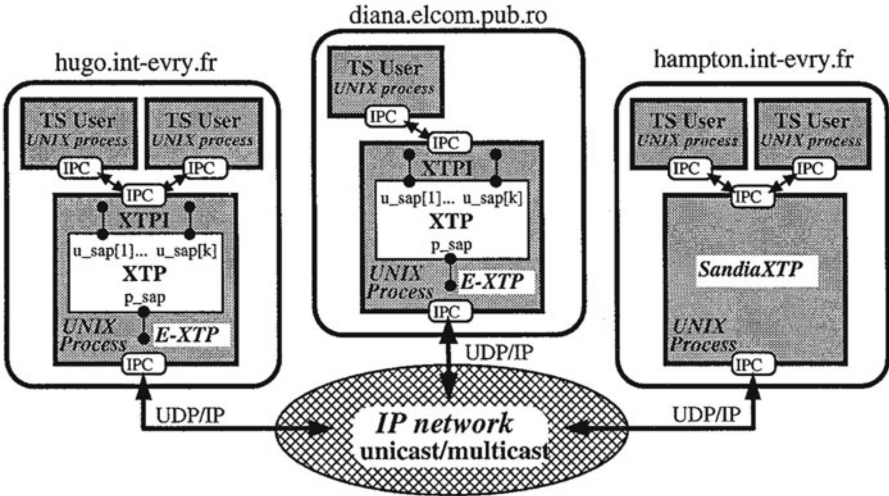

Figure 1. Hosts running E-XTP and SandiaXTP (TS = transport service).

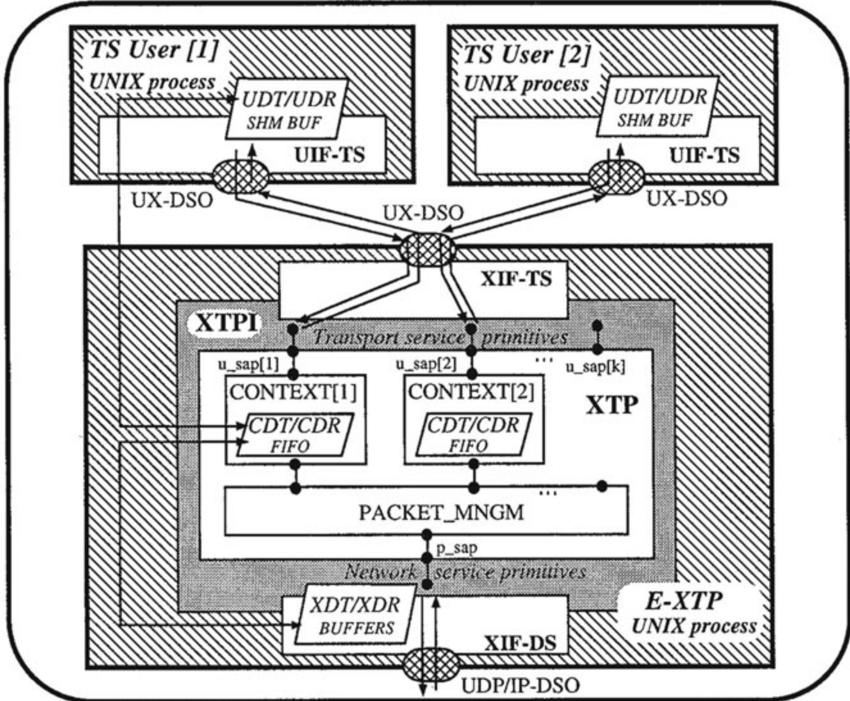

Figure 2. E-XTP structure and the interactions with the user processes.

UIF-TS is the interface of the user process for access to the transport service. It provides an API library [5] similar to that of SandiaXTP. *UIF-TS* is a state machine driven by messages received from E-XTP and by function calls made by the application program. Two communication mechanisms are involved: message transfer, via a UNIX domain datagram socket (*UX-DSO*), and shared memory, for sent and received data units. *UIF-TS* takes all the data types necessary for communicating with E-XTP from two C header files generated by the Estelle compiler from the specification.

XIF-TS is the interface of the XTP process with the transport service user processes. It consists of four C primitives, which are customised versions, for the XTP transport service interface, of the four EDT generic interface primitives (section 2.2). *XIF-TS* provides the adaptation between the Estelle inter-module communication mechanism and the UNIX domain datagram socket (*UX-DSO*), used for exchanging messages with the user processes. It is also responsible for the allocation of CONTEXT instances to users and for routing the messages between the user processes and the CONTEXT instances.

XIF-DS is the interface of the XTP process with the underlying data delivery service. It consists of four C primitives, customised versions for the UDP/IP interface, of the four EDT generic interface primitives. *XIF-DS* provides the adaptation between the Estelle inter-module communication mechanism and the UDP datagram socket (*UDP/IP-DSO*). It also performs the adaptation between the XTP packets representation in the Estelle specification and the real XTP packet formats, as well as the checksum computation.

UDT/UDR-SHM are shared memory buffers of the *UIF-TS* interface, used for data transmission and data reception, respectively. They are created by the user process and shared with the XTP process. These buffers permit the data transfer between the user process and an XTP context with a single data copy. *XDT/XDR-BUF* are collections of data buffers used by the *XIF-DS* interface. They store the data segments of the outgoing XTP packets and the received XTP packets, respectively. The transfer of the XTP data packets between the XTP contexts and the data delivery service requires a single data copy.

A CONTEXT instance contains the pair of buffers *CDT/CDR-FIFO*. *CDT* stores the transmitted data until acknowledgement. *CDR* stores the received data until the recovery of the lost segments and the delivery to the user. They existed in the Estelle specification, but the Pascal procedures for managing them are replaced by C primitives, aware of the *UIF-TS* and *XIF-DS* buffers.

3.2 E-XTP versus SandiaXTP

E-XTP and SandiaXTP provide similar XTP functionality, with similar API functions and can inter-operate. Both are user space implementations for UNIX environment and use sockets and shared memory for inter-process communication. However, there are important differences in the state machine

design and some protocol functions are performed using different policies.

Figure 3 shows throughput measurement results for SandiaXTP and E-XTP. They are made for a one-way data transfer, with Sparc stations Ultra 1, running Solaris 2.5, on 10 Mbit/s Ethernet. The following protocol settings are used: flow control, error control and checksum enabled; rate control disabled; no segmentation, acknowledgement at end of window. For these typical settings, the two implementations offer quite similar throughput values. E-XTP seems better tuned for multicast communications. Running in user space is a major handicap for both of them. As shown in the next section, about 50% of the processing time per data unit goes to system calls.

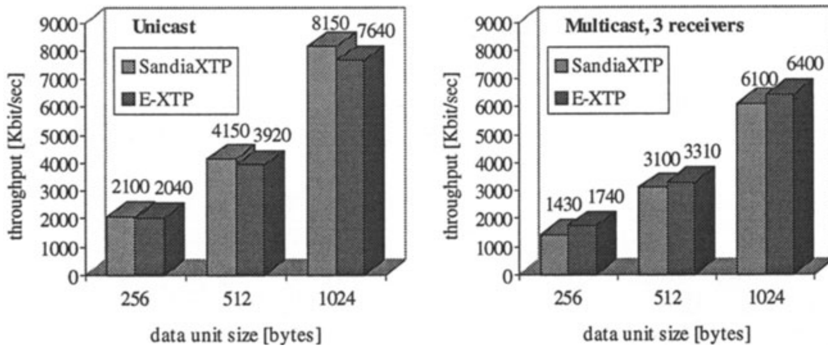

Figure 3. Throughput of E-XTP and SandiaXTP (user buffer to user buffer) as a function of the data unit size. The window size is 24 data units.

4. THE FACTORS AFFECTING THE EFFICIENCY

4.1 Communication mechanisms

Data and packets manipulation is the most obvious cause of overhead in a layered protocol architecture [6]. The various optimisation techniques used in hand-coded implementations [13] are based on the following principles:

- Avoid moving the data units by storing them in shared memory buffers and passing pointers to buffers between the communicating entities.
- Manage the variable size of the data units (insertion and removal of layer specific headers) without copying them, using either off-setting or scatter-gather. In the former case, a data unit is stored in contiguous memory, in a buffer accommodating the maximum data unit size, at a variable offset. In the latter case, the components of a data unit are kept in individual buffers and a descriptor indicates the number of buffers and their addresses.
- Integrate data processing (checksum computation, encryption, etc.) for multiple layers, to reduce the number of memory cycles [1].
- Anticipate the contents of the packet's or message's components, to minimise processing when sending them.

However, the integration of these optimisations in the automatic code generators is complicated by the FDT constraints and the difficulty to determine when and what some particular technique is useful and can be safely applied.

For a system modelled in Estelle, the communication consists (mainly) of asynchronous message exchange. The mapping of module instances to processes or procedures and the scheduling algorithm affect the implementation of the communication mechanisms. We refer in the following to the EDT model (a typical solution). Two kinds of communication can be identified:

- external communications, with the environment of the process;
- internal communications, with (sub)modules within the process.

For the first category, the implementation must rely on the inter-process communication (IPC) mechanisms offered by the target operating system. The tool must provide interface libraries for various operating systems and IPC mechanisms or/and enough flexibility to allow the user make the necessary adaptations. The latency of the IPC mechanisms is often the main cause of inefficiency. Shared memory and scatter-gather, if available and exploited by the tool's libraries, can avoid or minimise the data copying overhead.

The efficiency of the internal communications is mainly affected by the FDT restrictions. A general (always safe) solution cannot avoid copying the message parameters. For the implementations obtained with EDT, the transfer of an interaction implies a single copy of the parameters. First, an interaction buffer is allocated from a common pool and the parameters' values are copied to it. Next, the buffer's address is appended to the destination queue. The interaction's parameters are directly accessible to the receiving transition. The buffer is released when the execution of the receiving transition terminates.

However, the modules often just forward received information, possibly adding or removing some parts. This is due to the information transfer principles in layered protocol architectures and to message routing within the protocol entities. In such cases, the copy operation is redundant and very harmful to the implementation's performance (e.g., for a non-trivial protocol entity architecture, the packets are repeatedly copied). Queue management and message routing have a relatively more modest contribution.

Here is a simple example of packet transmission:

```
VAR p: pdu_type;
BEGIN
    build_pdu(p);
    OUTPUT ip_out.pdu(p);
END;
```

First, the packet is built in a local variable. Then, the OUTPUT statement is executed, allocates a buffer and copies the variable. The packet's fields (control, data segment) are copied twice, first to the variable, next to the interaction buffer. Obviously, the operations should be done in the reverse

order: one should first allocate the buffer, then use it to build the packet. For instance, the local variable *p* could be mapped to an interaction buffer. This requires a code generator able to detect that the variable is used as a parameter in a single OUTPUT statement and it is not modified afterwards.

The remark also inspires a different specification style. It is often possible to have most of the contents of the packet already prepared, as part of the local state. One can define the components of the packet as parameters of the interaction and let the OUTPUT statement assemble them (which is also more favourable for optimisations using scatter-gather or off-setting). Here is another example, a data transmission using the suggested solutions:

```
WHEN ip_in.data_req(sdu)
  BEGIN
    put_buf(sdu);              { append the data to the transmission buffer }
    IF can_send THEN BEGIN     { flow control permits transmission }
      crt_header.sequence := (crt_header.sequence + 1) mod MaxSeq;
      OUTPUT ip_out.dt_pdu(crt_header, sdu);
    END; { else: send when a received CNTL packet advances the window }
  END;
```

With this specification style, the transfer of the interaction causes only one copy of the packet's fields. The example also introduces another element involved in data manipulation: the transmission buffer, responsible for storing the data until the transfer is acknowledged. The copy operations to/from this buffer can be avoided if the data are stored in shared memory buffers and both the message transfer management and the transmission buffer management are aware of this. The operations depend on the protocol rules, unknown to the code generator. However, there are solutions which do not require special code generator capabilities and satisfy the generality required from a specification. We can make the *sdu* type a buffer descriptor (which can be copied without harm) and use C primitives for protocol buffer management.

In conclusion, several strategies can be used for integrating optimised data manipulation techniques in the automatic implementations.

One approach relies on the code generator for detecting particular situations when it can use an optimisation, instead of the general (always safe) method. Such a solution is proposed in [9]. Alternatively, the code generator could be guided with annotations inserted in the specification, indicating explicitly the context of the desired application of some optimisation technique. We are currently studying this solution, which seems more flexible and realistic, as it relies on the user's knowledge of the protocol's functionality.

Another approach uses C primitives for buffer management provided by the tool's library. It is the only solution which can address particular features of the implemented system and of the implementation context. It can complement the previous approaches, limited to general message transfer issues.

The latter strategy was used for E-XTP. One goal was to optimise the data path between the user buffer and the UDP/IP interface, with external and internal communications and send/receive buffer management. Another goal was to treat packet manipulation issues: encoding, decoding, checksum computation. The FDT data types and operators do not allow the use of real packet formats in the specification. Hence, the protocol machine works with an internal packet format (also "readable" by the simulator, for validation). The adaptation between the internal format and the real XTP format and the checksum computation are made by the primitives of the UDP/IP interface.

However, these problems mainly concern the control information in the packets. For data segments, the size and the order are the only issues which interest the protocol state machine. Therefore, it only knows a descriptor indicating the size of the data unit and the buffer which holds it. A transmitted data segment joins the header only when moved to UDP/IP and a received packet is split as soon as it is retrieved from UDP/IP (using scatter-gather). All the user data manipulations are done by a set of C primitives, responsible for moving the required amount of data between the *CDT/CDR* ring buffers in the CONTEXT module and the *UDT/UDR* user buffers or the *XDT/XDR* buffers in the *XIF-DS* interface (figure 2). Two copy operations are implied by this scheme (solutions with less copy operations were difficult to apply to the existing specification, requiring the redesign of the data streams management). The buffer management primitives used in the CONTEXT module cooperate with the interface primitives in the XTPI module. Hence, efficient solutions offered by the implementation context could be used: shared memory buffers at the upper interface and scatter-gather at the UDP/IP interface.

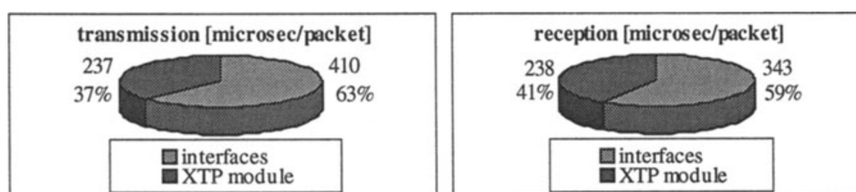

Figure 4. Contribution of the interfaces to the total processing time per data unit, for the XTP implementation (1024 bytes data units, Sparc station Ultra 1).

The experiments with E-XTP indicate a 300% increase of the user-to-user throughput, just by avoiding the copy operations for user data [4]. Figure 4 points out another major cause of throughput limitation: the latency of the socket interface. The values include three system calls: an exchange of messages with the user process, via the UNIX domain datagram socket, and the transmission or the reception of the XTP data packet, via the UDP socket.

E-XTP uses a connectionless underlying service. The lower interface just maps Estelle interactions to UDP/IP socket datagrams. This can be done by

primitives from a general purpose library, which call user defined functions for application specific message processing. In general, the mapping can be much more complex, e.g., for a connection oriented service accessible through an API library like TCP sockets or TLI. The flexibility of the EDT approach to external communications can be an important advantage in such cases.

4.2 System management overhead

An Estelle specification describes an hierarchically structured collection of module instances. The system modules occupy the top of the hierarchy. They run independently and synchronise with each other only by message exchange. Within a system module, further synchronisation rules are imposed: priority of a parent module over the children modules and, depending on the module attributes, either parallel synchronous or asynchronous interleaving execution.

According to the operational semantics, a system module performs cyclically a management phase, to identify the fireable transitions, followed by a transition execution phase. This is the usual approach adopted by the Estelle-based implementation tools. The transition selection algorithm is complex and can cause an excessive overhead. Various solutions to reduce this overhead have been investigated [7, 8]: optimisation of the algorithm defined by the operational semantics, structural transformation of the specifications, identification of favourable specification styles, different scheduling policies, etc.

As pointed out in [13], two main classes of models are used in hand-coded implementations: the server model and the activity thread model. With the server model, each system component processes events in an endless loop. The set of components communicate asynchronously and a scheduler offers to each one a fair chance to run. With the activity thread model, the components are implemented by procedures and communicate synchronously. The output of a message from a component is implemented by a call to the procedure which treats that message in the destination component. The system treats one event at a time, as a sequence of procedure calls (i.e., the "activity thread").

The existing automatic implementation tools, including EDT, use the server model, which better matches the Estelle and SDL semantics. The activity thread model can provide a considerable performance improvement, when applicable. However, there are major contradictions between the Estelle model and the activity thread implementation model. Hence, excessive restrictions for the Estelle specifications are needed to permit its use [10].

The current EDT implementation kernel uses a (relatively) optimised transition selection algorithm. It performs a single-pass search in the tree of module instances. For each instance, it looks for fireable transitions using table-based decodification (input/state table) followed by programmed selection (checking of provided clauses, interaction point identity, priorities). The algorithm is fully compliant with the Estelle operational semantics and does

not impose any restrictions concerning the Estelle language features.

One of our goals was to evaluate the influence of the system management overhead on the implementation's performance and its dependence on the Estelle model features and specification styles. The analysis was based on measurements made by probes inserted in the dispatcher loop (section 2.3). The time-stamps provided by these probes allow the calculation of pairs of values representing the execution duration for a transition and the duration of the preceding system management phase. The histograms in figure 5 show results of the measurements for E-XTP (average values). They were made on a Sparc station Ultra 1, for the following settings: SYSTEMACTIVITY attribute, one way transfer with 1024 bytes data units, error control, flow control and checksum enabled, no segmentation, one acknowledgement packet for 10 data packets. Figure 6 shows the overall contribution of the management phase to the processing time per data unit, done within the XTP module (computed from the collected samples, taking into account the relative firing rate).

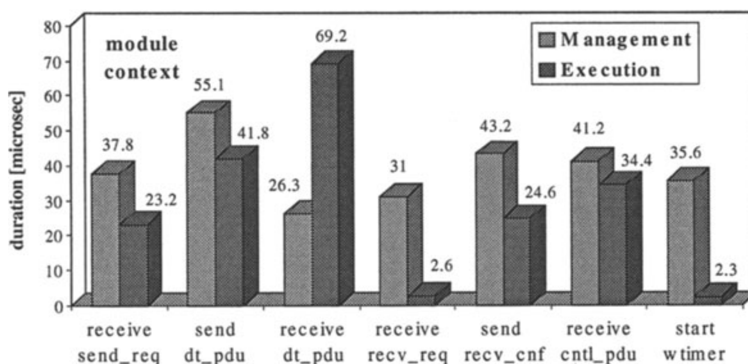

Figure 5. Management and transition execution duration for the data transfer, in a CONTEXT module instance.

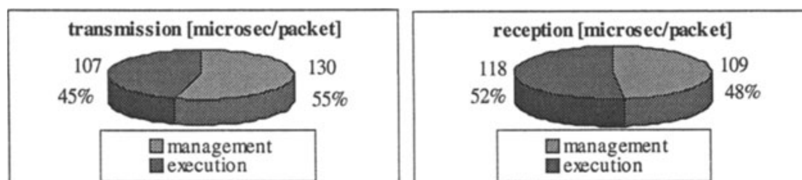

Figure 6. Contribution of the system management duration to the processing time per data unit, in the XTP module.

The joint analysis of various measurements, of the transition selection algorithm and of the specification, summarised in the following, points out the main causes of this relatively important overhead, as well as some solutions.

Due to the synchronisation constraints and the necessity to provide a certain fairness, the scheduler maintains a relatively complicated, tree-like, list of

module instances, reflecting the hierarchy. A selection implies a traversal of this list, often partial, but always starting from the root. For each instance, the fast, table-driven decoding, based on main state and input (if any), offers in general two sets of transitions, not just a single transition. One set contains the input transitions for the received interaction(s). The other set contains spontaneous transitions (no input), including delayed transitions, used to model timers. A slower, programmed selection must be used to choose a transition from these sets, by checking the PROVIDED clause, the priority and (for input transitions only) the interaction point.

The influence of the synchronisation rules on the selection duration depends on the depth of the hierarchy and the module attributes. An adequate choice of the attributes can reduce the overhead per executed transition. The SYSTEMPROCESS or PROCESS attributes require the visiting of all the children instances (when the parent has no fireable transition), but all the offered transitions are executed before the next management phase. The SYSTEMACTIVITY or ACTIVITY attributes allow the termination of the search at the first module instance offering a transition (with some fairness rule). For a high degree of concurrency between children instances, the former attributes are the preferred choice, otherwise, the latter.

The spontaneous transitions, mainly the delayed ones, are a major cause of overhead. They should be avoided, but two Estelle features demand their use.

The first one is the constraint to indicate the next main state in the transition header, before processing the received interaction. If the next state cannot be determined by a simple condition, tested in a PROVIDED clause, a spontaneous transition must be added just to change the main state. This artificial solution can be avoided by relying on variables and the PROVIDED clause, instead of the STATE declaration and the clauses FROM and TO. However, in this case, the table-driven decodification works on input only and a larger set of transitions remains for programmed selection. The possibility to change the state in the transition's body and to use output interactions in procedures would reduce to a large extent the need to use spontaneous transitions.

The second source of spontaneous transitions is timer modelling, using delayed transitions. A timer is associated to each delayed transition and runs while the transition is enabled. The enabling of a delayed transition corresponds to a "start" timer command and the disabling to a "stop" timer command. The transition is fired when the timer expires. The main state and/or a provided clause can be used to control the timer's activity. The inefficiency is due to the fact that the start/stop commands are *implicit* and the scheduler must check systematically the conditions of the delayed transitions to detect if a timer must be started or, conversely, if a running timer must be stopped. It is often necessary to restart a timer before expiring. Such a "restart" timer command, can be obtained by combining a "stop" and a "start". However, two computation steps are necessary: the first one to disable the delayed transition,

the second one to re-enable it. Another spontaneous transition must be added for the second step (e.g., the transition "start wtimer", in figure 5).

The presence of multiple interaction points, individual queues and transition priorities complicates the selection of the input transitions. First, there is a list of queues to be visited. Next, all the queues must be visited, even after finding an interaction, to check the priorities. Therefore, the preferable specification style is to indicate common queue as default and use individual queues only when really necessary. A common queue is especially important for multiplexer modules (e.g., PACKET_MNGM in the XTP specification) and usually there is no inconvenience. This kind of functionality is present very often in the systems and requires arrays of interaction points and transitions parameterised using the ANY clause. The selection algorithm and the code generator must treat these features with special care.

The excessive overhead of the outlined algorithm is due to the complex search of the transitions to fire. It is a general method, able to deal with the worst case. The solution we are studying is to replace the search by an event based management of a list of instances scheduled for (transition selection and) execution. If there are no spontaneous transitions, the list can be organised based on transferred messages: once a message is output, the receiving instance is inserted in the list. The solution can be extended for spontaneous transitions, if the PROVIDED clauses do not contain primitives or exported variables. In such a case, they can only be enabled by the execution of another transition of the same instance. The spontaneous transitions are checked only in this situation. If anyone is fireable, the instance qualifies for the scheduling list. It seems possible to obtain a considerable improvement for limited restrictions, acceptable by many applications.

5. CONCLUSIONS

The derivation of E-XTP from the XTP 4.0 Estelle specification, allowed us to asses the current EDT support for automatic implementation. E-XTP is functionally comparable with the hand-coded implementation SandiaXTP and inter-operates with it. They have similar, but relatively moderate performance.

The automatic implementation methods can match the requirements of the modern high speed network environment only if they integrate the highly efficient techniques already used in hand-coded implementations [9, 13]. Of main interest are the solutions used for data and packets manipulation and for mapping the system's components (e.g., entities in a layered protocol architecture) to processes or threads of the target implementation environment.

We experimented the optimisation of the data stream implementation using libraries of primitives which integrate the internal and external communications and the protocol buffer management. The approach is pragmatic and flexible. The libraries can be reused for applications having the same imple-

mentation context. Further benefits can be obtained by adding automatic code generator support for general message transfer optimisation.

Our analysis pointed out the Estelle features responsible for the excessive system management overhead. It also showed that the scheduling algorithm can be substantially improved, for widely acceptable constraints. The efficiency can approach that of the activity threads, without the excessive restrictions imposed by the strict application of this model [10].

The solutions used in the specification must take into account implementation issues and must be correlated with the support offered by the tool. A good basis for the implementation could be obtained without loss of the specification's generality, using just an adequate specification style. The adaptation in the implementation phase is expensive, harmful and should be avoided.

REFERENCES

[1] M. Abbot, L. Peterson. Increasing Network Throughput by Integrating Protocol Layers. *IEEE/ACM Transactions on Networking*, vol. 1, no. 5, 1993.

[2] J.W. Atwood, O. Catrina, J. Fenton, W.T. Strayer. Reliable Multicasting in the Xpress Transport Protocol. *Proceedings of the 21st Conference on Local Computer Networks (LCN'96)*, Minneapolis, Minnesota, USA, 1996.

[3] O. Catrina, E. Borcoci. Estelle specification and validation of XTP 4.0. Deliverable 2 for Task 2.1, Copernicus Project COP62 (COP#62/WP2/2), 1996.

[4] O. Catrina, E. Lallet, S. Budkowski. Implémentation automatique d'XTP à partir d'une spécification Estelle. *Ingénierie des Protocoles - CFIP'97.* Hermes, Paris, 1997.

[5] O. Catrina, E. Lallet, S. Budkowski. Automatic protocol implementation using Estelle Development Toolset. Research Report 971001, INT Evry, France, 1997.

[6] D. Clark, V. Jacobson, J. Romkey, H. Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications*, 27(6), 1989, pp. 23-29.

[7] R. Gotzhein, et al. Improving the Efficiency of Automated Protocol Implementation Using Estelle. *Computer Communications* 19, 1996.

[8] T. Held, H. König. Increasing the efficiency of computer-added protocol implementations. *Protocol Specification Testing and Verification XIV.* Chapman & Hall, 1995.

[9] R. Henke, H. Konig, A. Mitschele-Thiel. Derivation of Efficient Implementations from SDL Specifications Employing Data Referencing, Integrated Packet Framing and Activity Threads. *SDL'97: Time for testing - SDL, MSC and trends.* Elsevier, 1997.

[10] R. Henke, A. Mitschele-Thiel, H. Konig. On the influence of semantic constraints on the code generation from Estelle specifications. *Formal Description Techniques X and Protocol Specification, Testing and Implementation XVII.* Chapmann & Hall, 1997.

[11] Estelle Development Toolset (version 4.1). Institut National des Télécommunications, Evry, France. http://alix.int-evry.fr/~stan/edt.html.

[12] E. Lallet, S. Fischer, J.-F. Verdier. A new Approach for Distributing Estelle Specifications. *Formal Description Techniques VIII.* Chapman & Hall, 1995.

[13] L. Svobodova. Implementing OSI Systems. *IEEE Journal on Selected Areas of Communication*, vol. 7, no. 7, 1989, pp. 1115-1130..

[14] T. Strayer. SandiaXTP User's Guide. SandiaXTP Reference Manual. Sandia National Laboratories, USA, 1996.

[15] Xpress Transport Protocol specification revision 4.0, XTP Forum, Santa Barbara, CA, USA, 1995. For the revised multicast procedures, see [2].