Semi-Automatic Implementation of Transport and Session Protocols

Gregor von BOCHMANN *
Université de Montréal, Dept. d’Informatique et de Recherche Opérationnelle, C.P. 6128, Succ. A, Montreal, Que. H3C 3J7, Canada

The paper describes experience with the use of formal protocol specifications in the protocol implementation process. As formal description techniques (FDT) for OSI protocols are being standardized, formal OSI protocol specifications in these FDT’s become available on a trial basis. The technical issues involved in the use of such specifications for the automation of the implementation process are discussed, and the experience with a semi-automated implementation approach for the OSI Transport and Session protocols is described.

Keywords: Protocol implementation, formal description techniques, Transport protocol, Session protocol, automated implementation.

1. Introduction

Methods for formally specifying communication protocols and services received much attention recently (see for instance [16]). Such methods become important in relation to their use for protocol design validation, protocol implementations and testing. It seems that some of these methods have advanced enough to make them usable in the design and implementation of real systems involving real-life protocols, including standards such as those developed by ISO or CCITT. The interest in formal specifications is stimulated by the fact that the standardization community of ISO and CCITT realizes that the use of formal description techniques (FDTs) for the specification of protocols and service standards has certain advantages. In particular, formal specifications tend to be more precise than descriptions given in natural languages. This simplifies the validation, implementation and testing efforts. Work is underway within ISO and CCITT to develop FDTs for specifying OSI protocols [11,23].

Formal protocol specifications, like informal ones, are used for the following purposes:

(a) They serve as a “reference” specification, i.e. a specification of a communication service or protocol which serves as the authoritative reference for all other activities.
(b) Protocol and service specifications are used for the validation of the design of the protocol of a given layer, by comparing the service provided by the protocol entities and the communication service below with the service specification of the layer in question.
(c) The protocol specification is used for the elaboration of an implementation.
(d) The protocol specification is used during the validation (debugging, testing) of an implementation, and for assessing its conformance with the protocol specification.

Experiments with automated tools for the above
activities have been reported in the literature. Such tools become important when formal specifications are used for real-life protocols which are usually sufficiently complex to make some automation desirable.

This paper considers the automation of the protocol implementation activity (point (c) above). It is assumed that the protocol specification is given in an extended finite state machine formalism [5], such as Estelle [12]. Using such a formalism, a protocol entity executing the communication protocol in question is described as one or several interconnected machines which interact through input/output interactions. The behavior of each machine is described as a finite state transition machine extended with interaction parameters and additional state variables. The relation of the state transitions with these parameters and state variables is described using a programming language notation (Pascal in the case of Estelle).

In section 2 of this paper, general issues and design choices for protocol implementations are discussed. Also different objectives for the implementations are considered. Section 3 describes a general implementation strategy which is based on the extended state machine formalism. For this implementation strategy, an FDT compiler has been developed which translates a formal specification into appropriate Pascal code which can be incorporated into a Pascal program implementing the protocol specification. Several real-life implementations of the ISO-CCITT Transport protocol and an implementation of the Session protocol are discussed in section 4. Most of them were obtained using the FDT compiler mentioned. A comparison between an ad hoc implementation approach and the use of the FDT compiler is also made. Finally, section 5 gives a short discussion of the results presented in the paper, and a comparison with other related work.

2. Issues in protocol implementations

In communication software design, it seems natural to model the structure of the software modules in some way along the lines of the layered structure of the protocol architecture. This architecture often follows the OSI Reference Model, or a subset of the layers defined in that model. Usually several levels of protocols are involved in a given communication system. The communication software must be written in such a way that

(a) all properties defined in the protocol specification are satisfied by the system (this means the system conforms to the protocol specification), and

(b) properties not defined by the protocol specification are chosen and implemented in such way as to make the resulting system useful; in particular the following issues must be addressed:

- efficiency of operation: communication delays introduced, maximum throughput obtainable, memory requirements, etc.
- appropriate interfaces to the user programs,
- appropriate interfaces to the underlying data transmission facilities, usually through the I/O facilities of the operating system.

We assume in the following that an implementation of the protocol is to be obtained based on a formal specification of the protocol(s) given in an extended state transition formalism, such as Estelle. In this case, the properties of an implementation not defined by the specification usually relate to

- expressions, statements, functions, or procedures not explicitly defined, or
- the nondeterminism in the specification due to the fact that in a given state and for a given set of input interactions to be considered, there may be more than one of the defined transitions which are candidates for execution. Nondeterminism may also be introduced by spontaneous transitions which may be executed provided that the present state satisfies a specified condition without involving any input.

The complete protocol specification for a given system consists usually of several “extended finite state machines” (sometimes called “modules”), one or several for each protocol layer. It is therefore important to determine how the interactions between these different modules is realized in the implementation. Usually the specification defines in which manner the different modules are connected with one another. Some of these modules also interact with the rest of the system (the user
or the I/O system for communication). Important
design decisions relate to the manner in which
these different interactions are realized. The im-
plementation strategy discussed in section 3, for
instance, automatically provides certain alterna-
tives for the interactions between modules, and
provides for a framework in which the interactions
with the remaining part of the system can be
realized in a flexible manner, depending on the
interfaces provided by the operating system.

Another important design decision is the ques-
tion of how many processes are used to implement
the protocol system, and how these processes are
supported by the operating system. Extreme possi-
bilities are to use one process per module in the
protocol specification, or alternatively, to imple-
ment all modules within a single process.

Based on a formal protocol specification, pro-
tocol implementations can be obtained automati-
cally, as for instance discussed below. Automated
implementations can be useful for different pur-
poses, such as the following:

(a) For providing an operational system, which
may be used for various applications requiring
the communication services provided by the
protocol(s).

(b) For performing simulated executions of the
protocol(s): This may be useful during the
design of the protocol for analysing the logical
correctness of the protocol [14,21], or for mak-
ing performance simulations [22]. Performance
simulations are in particular useful for de-
termining optimal parameters for a protocol
implementation which should satisfy certain
performance objectives.

(c) For analyzing the observed behavior of an
other protocol implementation, in order to test
whether the latter conforms to the given pro-
tocol specification [14,21,26].

It is important to note that for each of these
different purposes of automated implementation,
different design decisions seem to be appropriate
for the structure of the implementation approach,
in particular in respect to the realization of the
different possible implementation choices not de-
defined by the specification (see above). An imple-
mentation approach suitable for obtaining oper-
ational protocol implementations (point (a) above)
is described below.

3. An Implementation Strategy

An implementation strategy for the implemen-
tation of higher-level protocols is described in
[9,19]. Based on a formal specification of the
protocol to be implemented, several stages of re-
finement are distinguished. In a first stage of
refinement, the formal specification is completed
with such details that are implementation depen-
dent, but that can be formulated in a manner
independent of the operating environment in which
the implementation is to run. These details may
relate to the handling of user and/or peer proto-
ocol errors, the choice between different simulta-
nceously enabled transitions, or the handling of
spontaneous transitions. In a second stage, those
details are added to the specification which are
dependent on the particular environment in which
the program operates. These details may relate to
the way the program communicates with other
programs in the system, or to the use of operating
system resources.

The detailed specification must then be trans-
formed into corresponding procedures in an im-
plementation programming language. For the im-
plementations discussed in section 4, this was
done in one case in an ad hoc manner, in the other
cases by the use of an FDT compiler [13] which
translates a formal specification into a set of Pas-
cal procedures. The structure of the implementa-
tion obtained by this translation is further de-
scribed in [9].

4. Experiences with Transport and Session Proto-
col Implementations

Experience with two Transport class 0/2 proto-
col implementations is described in sections 4.1
and 4.2. The first implementation was based on a
formal specification. An ad hoc implementation
approach was chosen, not necessarily following
the principles described in section 3. A second
implementation, also based on the same formal
specification was obtained following the strategy
described in section 3 and using the FDT com-
piler. A comparison of these two implementations
is given. Both implementations run on a PDP-11
computer under the RSX operating system.

Subsequently, implementations of the class 2/4
Transport protocol and a simple Session protocol
were made. For both of these projects, the implementation strategy of section 3 and the FDT compiler were used. These projects are briefly described in the sections 4.3 and 4.4.

4.1. An ad hoc Implementation of the Transport Protocol Based on a Formal Specification

The structure of the first implementation (for more details see [19]) is shown in fig. 1. The Transport entity is a single task in the operating system, communicating through operating system primitives with a task providing the Network service, and several user tasks which may establish one or several Transport connections with remote systems through the Transport entity.

The interactions between the different tasks is based on message exchange provided by the operating system. However, the user data is not directly included in these messages, rather pointers to data buffers are passed between the processes. The logical behavior of the Transport entity and its program structure was derived in an ad hoc manner from a formal specification of the protocol [6] given in a version of Estelle.

The spontaneous transitions were handled in an ad hoc manner. The code corresponding to a given transition was directly included in those input transitions after which the spontaneous transitions in question should be executed. The result of this transformation was that the program has the form of a loop which performs the processing for the incoming interactions, one after the other.

The protocol implementation was tested using the interactive Transport protocol tester developed earlier [24]. The class 0 part of the implementation was also tested by an automatic tester [7,10] executing test sequences which are believed to provide a relatively exhaustive validation of Transport protocol implementations [17].

The experience of this implementation [19] showed that the availability of a formal specification significantly simplifies the implementation process; however, only part of the implementation is directly related to the formal specification. Much time was spent in the development of the interfaces with the operating system for interaction with the user processes and the Network communication service, including buffer management. Another important part, not included in the formal specification, is the coding and decoding of PDUs. The size of the Pascal source code for these different program sections is given in table 1 above (column A).

4.2. A Semi-automatic Implementation of the Transport Protocol

In order to evaluate the usefulness of an FDT compiler for the automatic generation of parts of a protocol implementation, the same formal specification that was the basis for the ad hoc implementation described above was also used for generating semi-automatically an implementation using the FDT compiler. The same buffer mana-

Table 1

<table>
<thead>
<tr>
<th>Part of program</th>
<th>Number of program size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>source lines (in octets)</td>
</tr>
<tr>
<td>(a) PDU de- and en-coding</td>
<td>3000 3000 11,940 11,940</td>
</tr>
<tr>
<td>(b) Code corresponding to the transitions of the formal specification</td>
<td>3000 5500 17,800 29,306</td>
</tr>
<tr>
<td>(c) Buffer management and O/S interfaces for intertask communication</td>
<td>3000 3000 3974 3974</td>
</tr>
<tr>
<td>(d) Run-time support routines</td>
<td>1000 1400 2324 3452</td>
</tr>
<tr>
<td>(e) main program</td>
<td>1000 400 6468* 3282*</td>
</tr>
</tbody>
</table>

* Including static variables
agement and intertask communication routines were used in order to make the comparison between the two implementation approaches more meaningful. The resulting program sizes are shown in table 1. (column B). It is noted that only part (b) is generated by the FDT compiler, and part (d) is the standard set of procedures used as runtime support for the compiler-generated procedures. The parts (a) and (c) are the same in the two different implementations. As table 1 shows, the transition code generated by the compiler is larger than the corresponding code of the hand-coded implementation, but it turned out to be of a more regular structure. This part of the program represents 53 percent of the total program size; the fixed support routines represent another 6 percent of the code. These figures are similar to those quoted in [4].

As the table shows, the buffer management and intertask communication routines are relatively complex. However, the FDT compiler allows the integration of several separately specified modules into a single Pascal program. The implemented Transport protocol entity, for instance, consists of one “mapping” module and several “AP” modules. Also, the specification of the protocols for several layers may be compiled into a combined, single program (task). This would reduce the intertask communication overhead associated with an implementation where each layer protocol would be implemented in a separate program.

A comparison of the runtime efficiency of the two implementations yielded the following results. The hand-coded implementation was always faster than the one obtained with the compiler. The ratio between the maximum throughput obtainable with the two respective implementations ranged between 1.16 and 1.5 for data transfer with a simulated network connection, between 1.08 and 1.8 for data transfer through the real network, and between 1.5 and 1.6 for connection establishment and disconnection. These numbers correspond to different tests involving either a single or several connections, and different classes of protocol. The interpretation of these numbers is complicated by the fact that both implementations use overlays because of the small addressing space available on the PDP-11. The larger size of the compiler generated implementation leads to additional overlay swapping, which may explain part of the efficiency difference.

4.3. Implementation of the Transport Class 4 Protocol

An implementation of the Transport protocol classes 2 and 4 is in progress. This project uses the implementation strategy which is described in section 4.2. However, the program runs on a VAX computer under the VMS operating system. This larger computer was chosen because of the memory limitations of the PDP-11 computer.

The formal specification developed by ISO [20] was used as the basis for this implementation. During the different stages of this work, a number of difficulties and problems with the specification were identified. We thank W. McCoy (from NBS, Washington) for helping us in the resolution of these issues. It seems that the identification and resolution of these issues was one of the useful side effects of this implementation project.

4.4. Implementation of a Simple Session Protocol

In parallel with the implementation of the class 4 Transport protocol, an implementation of a simple Session protocol was made in the same operating environment using the same implementation strategy. In the lack of a suitable formal specification of the OSI Session protocol, we developed a new formal specification including the Session kernel functions, two-way alternate and simultaneous data transfer and release functions. An attempt was made to use in the formal specification as much as possible the names and identifiers used in the ISO Session standards. In contrast to the Transport protocol program which handles multiple connections, a single copy of our initial Session implementation handles only one connection. However, it is very easy to configure other kind of program structures which could support multiple simultaneous connections.

5. Discussion and Conclusions

As discussed in this paper, the availability of the formal specification of a protocol can be useful for the validation of the protocol design, as well as for protocol implementation and testing. This paper discusses, in particular, the semi-automatic implementation of protocols based on their formal specification given in an formal description
technique (FDT) based on an extended finite state machine formalism, such as Estelle. It is important to note that a protocol specification usually leaves important design decisions unspecified; these design decisions must be made for each implementation of the protocol depending on the particular requirements for that implementation.

Specifications in Estelle sometimes tend to appear "implementation oriented", in the sense that they seem to imply certain design decisions which could be considered a matter of implementation. Implementations using these decisions can be obtained semi-automatically, as discussed in this paper. However, it is conceivable that other implementations would be built which use different, but equivalent mechanisms. The automatic generation of such implementations is much more difficult, as it is related to program transformations.

Other specification languages, such as Lotos [15], which are intended for more abstract specifications, would usually leave more design decisions to the implementation phase. This, clearly, makes the automatic generation of efficient implementations a more difficult task.

The approach to protocol implementations discussed in this paper is related to many other efforts in this area [3,4,18]. In contrast to the latter, our FDT compiler accepts a language very similar to an emerging FDT standard [12] and allows the integration of arbitrarily many modules within a single program implementation.

As discussed in section 4 in relation with the Transport protocol implementations, the FDT compiler produces readable code which is relatively efficient in space and runtime. It could therefore be used for many protocol implementation projects, provided that a formal specification of the protocol is available. However, it is also clear that it would not be used in cases where a high-performance implementation is desired.

Further experience with the semi-automatic implementation approach is planned. Areas which would profit from further research include the following:

(a) The automatic inclusion of testing facilities within the generated implementations.
(b) Improvements in the code generated for handling interactions between module instances and for the initialization of the module interconnection structure. An implementation language with less strong typing rules than Pascal may be useful for some of these aspects.
(c) Adaptation of the FDT compiler to the final version of the FDT language standard, when the latter becomes available.
(d) An integration of the FDT language and compiler with PDU coding and decoding facilities based on a standardized notation, such as defined in [1,2].

Acknowledgements

I would like to thank J.M. Serre and C. Antonescu who did most of the work described in this paper. M. Maksud and E. Cerny also contributed to this work. This work was funded by the Natural Science and Engineering Research Council of Canada, and the Department of Communications Canada.

References


