UNIX STREAMS Generation from a Formal Specification

Paweł Rychwalski and Jacek Wytrębowicz*

Institute of Computer Science, Warsaw University of Technology,
Nowowiejska 15/19, 00-665 Warsaw, Poland

Abstract. This paper describes a new idea of rapid protocol implementation starting from its formal specification, namely to generate Unix STREAMS modules. We have exercised this idea using Estelle formal specification technique. The generator was written for Linux system. The paper demonstrates how the semantic problems were resolved and gives some conclusions from generations we have performed.

Keywords: automatic code generation, Unix STREAMS, formal description techniques, Estelle

1 Introduction

Using formal description techniques in engineering of telecommunication protocol is common. For example an IEEE document that standardize a protocol contains SDL documentation in an annex. The most popular specification languages for protocol design are SDL, Estelle and Promella. All of them give an extended finite state machine (EFSM) model of a protocol. This model suits engineer demands for protocol validation, verification and test generation very well. Rapid prototyping i.e., automatic code generation from a formal specification into a working program is required not only due to time to market competition but also to obtain as correct implementation as the specification is. A designer has the opportunity to automatically generate a code when he uses a development environment for SDL or Estelle, like Tau SDL Suit from Telelogic (www.telelogic.com) or Estelle Development Toolset from INT (http://www-lor.int-evry.fr). These generators were not worked out to meet high efficiency or to follow a specialized interface of an operating system used for protocol implementations.

On the other hand, D. Ritchie has proposed STREAMS - an efficient mechanism for protocol implementation for Unix at 1984, [1]. This mechanism is included in every commercially distributed Unix operating system, e.g., Sun Solaris [2]. Thus most protocol implementations for these systems are Unix STREAMS modules. Designers of new protocol implementations have to determine STREAMS mechanism for Unix systems.

* The research was partially supported by KBN grant No 7 T 11 C 013 20.

H. König, M. Heiner, and A. Wolisz (Eds.): FORTE 2003, LNCS 2767, pp. 1–14, 2003.
© IFIP International Federation for Information Processing 2003

Despite of the fact that both Unix STREAMS and formal description techniques exist about 20 years, there were no attempts to lead out a STREAMS implementation from a formal specification. Several existing works about protocol implementations starting from its specification show, that generated implementation can be efficient. O.Catrina and A.Nogai [3] have compared the efficiency of a generated XTP implementation versus a hand-written one. J.Thees [4] and J.Bredereke [5] have analysed different heuristics, which could be applied during generation. R.Gotzhein et al. [6] analyzed how specification style can influence the efficiency of generated code. P. Langendörfer and H. König [7] proposed an interesting extension to SDL called iSDL, which consists in providing some annotation to control the way of code generation.

In [8] and [9] authors propose a new mechanism called "activity threads", which passes messages by procedure calls. This mechanism is very interesting, however it is not applicable to Unix STREAMS e.g., the STREAMS technique is based on message passing via queues not by procedure calls.

Because a STREAMS implementation is efficient due to its concept and integration with operating system, we have decided to check the feasibility of a STREAMS module generation from an EFSM specification. To carry out this work, we have selected the Estelle specification language and Linux operating system by the reason of easy access to related tools.

There is not enough place in this paper to give a complete explanation of Estelle specification technique nor Unix STREAMS, hence we refer a reader who is unfamiliar with them to [10] and [2]. In this paper we compare the Estelle semantics versus STREAMS semantics, we describe the selected translation model and we give some insights into a generator we have developed for Linux. In conclusions we present some results collected from efficiency tests of generated modules.

2 Estelle Semantics versus Streams Semantics

We can use both Estelle and STREAMS to implement network protocols, so it is obvious that there are some similarities between them. The main ones are:

Modules – In both models, a protocol instance is represented by a module.
Datagrams – Both Estelle and STREAMS modules exchange data in datagrams.
Queues – The datagrams are queued at the entrance of a module.

While the idea of passing from an Estelle protocol specification to STREAMS implementation seems to be quite natural, there are many difficulties to approach it to reality. The problem arises due to differences between Estelle formal semantic and STREAMS operational semantics. The first difference is the range of specification: in Estelle you specify a whole communication system, while one STREAMS module handles just one protocol. However, the most important is the possibility of communication with other modules: in Estelle, one module can

communicate with any number of other modules, while in STREAMS one module can exchange data with just two other entities: upper and lower layers of protocol. The other differences include, but are not limited to:

Dynamism – Estelle modules can be dynamically created and removed by another module, while a STREAMS module can only be pushed into and popped from a stream by a user-level process, which is an external (from module's point of view) entity.

Message queues – Estelle allows interaction points to share a common queue, while in a STREAMS module, each "interaction point" has its own queue.

Exported variables – This kind of communication between Estelle modules cannot be represented as communication between STREAMS modules. The only communication mechanism for STREAMS is message passing.

Module synchronization – There can be a synchronization between Estelle modules defined in a specification by using module attributes (i.e., *process*, *systemprocess*). There is no synchronization between STREAMS modules, they work independently from one another.

Non-complete specification – In an early project stage or for documentation purposes it is useful to employ any-type, any-value or other similar constructs. Of course it is impossible to derive any implementation from them.

These aspects cannot be translated into STREAMS directly for all specifications. That is why there should be some requirements on the input Estelle code, that will allow it to be translated into semantically equivalent STREAMS module. Chapter 3 describes them shortly.

Queues

Both Estelle and STREAMS modules have queues, yet the queue models obviously differ from each other. Estelle queues are "logical" and unbounded, while STREAMS ones are "physical" C structures with limited capacity.

The limit of a STREAMS module queue can be quite high. Furthermore, the queue limit is not a size of a fixed array (in C-language sense), because queues are implemented as dynamic lists. It means that a module can put any number of messages into its own queue, even if it is over its high water mark. The only situation when the queue limit is checked is during passing the message further to another module (*canputnext()*), and it is the target module's queue that is checked.

The overflow of a STREAMS queue can be avoided by slowing down the sending module. It corresponds to the behavior of independent entities with no assumptions about their processing speed in a specification. Thus Estelle system modules and activity modules with unbounded queues have the same behavior as STREAMS modules using the *canputnext()* function to avoid message loss.

EFSM Semantics

The semantics of STREAMS tells nothing of internal module behavior, that is how it interprets and handles incoming messages, except for some standard guidelines. In particular, there are no restrictions on handling messages of user-defined structure.

All of Estelle EFSM elements can be easily translated, or rather implemented in a STREAMS module. Internal variables of any type can be allocated in the module. Such variable can store a message parameter, the current automaton state, or any other internal data. Timers (for *delay* clauses) can be easily implemented using the library *timeout()* function[1]. A transition can be implemented by a simple C function. A module can select and execute one or many transitions when it gets control, i.e., its *put()* or *service()* function is called or the timer completes. The module returns from its *put()*, *service()* or timer callback function when there is no ready transition to execute or when it must slow down to avoid overflowing of a destination queue.

Specification languages allow to describe a non-deterministic behavior, because it is useful for documentation and for model analysis. An implementation should work in a deterministic manner, thus any non-deterministic statement is translated to a deterministic code, e.g., the Estelle *delay(a,b)* clause is implemented by a timeout function that is set to the *a* value.

3 Translation Model

To make generation of STREAMS implementation from an Estelle specification possible, we have to impose some restrictions on an input Estelle specification. They are the following:

- The first and the most important one is the range of specification: STREAMS module will be generated from just one Estelle module body definition.
- The second restriction is on the number of interaction points and their queue disciplines. To match the semantics of a STREAMS module, Estelle module definition must have exactly two external interaction points, with individual queues.
- As the STREAMS module cannot “export” its variables to another module, the input Estelle module body cannot have any exported variables.
- Input Estelle specification must be complete.

Since STREAMS modules are independent and non synchronized entities, the best equivalent of them in Estelle specification are system modules. Moreover, system modules cannot have exported variables. That allows us to simplify the requirements for semantically correct translation:

> A STREAMS module can be generated from a single body definition from a complete Estelle specification, when the body’s header is labeled

[1] The *timeout()* function is a request for STREAMS scheduler to call a (given in a parameter) callback function in the module after a specified amount of time. See [2].

system and it has exactly two non-array interaction points with individual queues.

When an Estelle module meets the above constraint, we can translate all elements from its body definition into STREAMS module code. The following subsections describe these elements and tell how they should be translated.

Dynamic Module Creation

Since there are no restrictions on the internals of a STREAMS module, it can implement a model of dynamic hierarchy of Estelle modules. This means that one STREAMS module can represent a whole subtree of Estelle modules. Usually, the more complex specification structure is, the less efficient implementation is. A dynamic structure causes some system overhead for memory allocation/deallocation. That is why a dynamic module hierarchy should be very carefully implemented in order to achieve desired efficiency.

Messages

The only module in the Estelle hierarchy that can communicate with "external world" is a system module. It has two external interaction points, which will be mapped into STREAMS queues. It means that a STREAMS message received by the generated STREAMS module (called e-module further in this paper) will be passed to EFSM engine of the system module. An interaction outputted via an external interaction point of this module will be translated into STREAMS message and sent into the stream. Such message will need to have a special, recognizable structure, so it can be understood by other e-modules. These messages are called e-messages further in this paper.

Interactions that are exchanged between Estelle modules within the module hierarchy will not be translated into STREAMS messages at any point. They should be implemented as some C structures used only by the EFSM engine. Thus possible Estelle "attachments" that join external interaction points between a child and his parent module should be implemented as C structures without involving any STREAMS mechanisms.

EFSM

There should be some common EFSM engine code for all e-modules. Within one e-module, the engine must have two versions: one for the topmost system module, sending STREAMS messages, and one for the children modules that do not communicate with external entities. The engine should keep track of current states of all modules.

A STREAMS module, and thus the EFSM engine, can run only when it receives a STREAMS message or a *timeout()* callback function has been called by the STREAMS scheduler. While it is not a problem for the engine of the system module, the engines of children modules can be triggered only when the

system module gives them control. The following algorithm, which works the same way for e-messages received from lower as from upper layers, is the best way to solve this problem:

1. The topmost module receives an interaction in an e-message or a *timeout()* callback is run (for delayed transition).
2. The topmost module runs its transitions until there is no fireable one, which may include firing some spontaneous transitions. All messages are queued in appropriate places: the messages to external modules in a special buffer, the messages to children modules in their queues.
3. Other modules in the hierarchy run their transitions, until there is no fireable one left: all automatons wait for an interaction, or there are delayed transitions.
4. STREAMS messages buffered earlier in the topmost module are sent into the stream (via *putnext()*, or *putq()*) and so the control is passed to another STREAMS module.

All Pascal code elements, like type definitions, constants, functions, etc. should be translated into appropriate C-language constructs. Module variables should be stored in the e-module's private data structure. There is a special place for pointer to such a data in STREAMS module structures.

Module Attributes

An Estelle module that should be implemented as STREAMS module has to be asynchronous to the others. Hence it should be attributed as *system*. If that Estelle module has any children modules, they can be attributed without any restriction. The EFSM engine of the parent module is responsible for a correct synchronization of children's EFSM engines.

Nondeterminism

Nondeterminism is used to define a class of acceptable behaviors, or to express nondeterministic behavior of an actor or an environment external to designed protocol. Thus we can remove nondeterminism during implementation. Any efficient deterministic behavior of an e-module satisfies the corresponding Estelle specification.

Protocol Layers

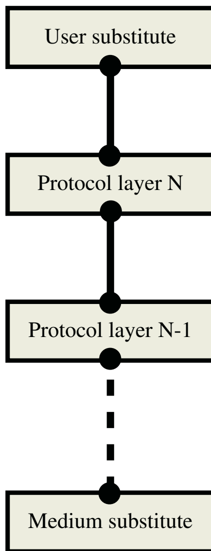
It is virtue that a specification language can express many different kinds of implementation or ideas. It is useful for documentation or analysis purposes. Although, no one expects nor needs, that a given implementation technique could be used to implement any abstract specification. Ritchie has conceived STREAMS for efficient protocol implementation on the base of the OSI ISO model. Thus a specification that respects this model can be easily translated into STREAMS implementation.

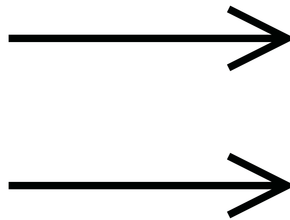
We know many Estelle specifications, which were written by our colleagues by ourselves or by others. Frequently there are modules that do as an unreliable medium, a protocol user, an observer. A designer writes them for validation and verification purposes. For that reason we argue that the designer should explicitly point which module represents a protocol to be implemented. We can also notice that designers specify co-working protocols separately and analyze them independently. The reason is simple - their conceptualization does not have to go in the same time.

A single generator run should translate one Estelle module body definition into one STREAMS module. The generated module has a well-defined STREAMS interface (e-messages), so it can exchange data with other modules and applications, which should understand these messages.

Results of few subsequent generations (that can for example represent different layers of protocol) can be put in a single stream. The source body definitions may be defined in one Estelle file, but they don't have to. A generator should analyze just the indicated body definition and its contents, and should pay no attention to the initializing code at higher level.

Figure 1 demonstrates how modules from a single Estelle specification can be mapped into STREAMS modules. Note that there is no need to split the specification before generating source code for STREAMS.





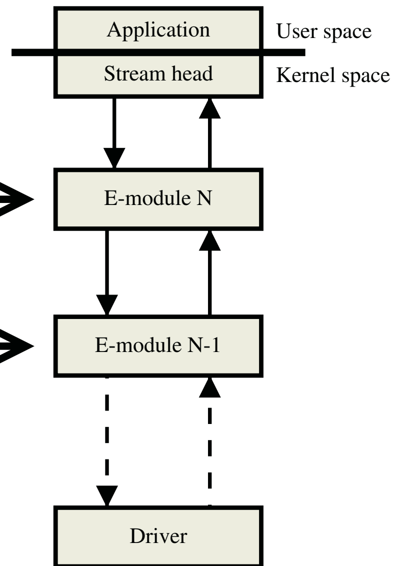


Fig. 1. Translation of multiple protocol layers

4 Generator for Linux

The target platform for our generator is the Linux system. We have selected it due to the openness of its source code and the possibility to exercise on system level. Unfortunately there is no STREAMS subsystem in Linux kernel. We have selected a freeware Linux STREAMS library, which is not yet complete. [11]

We have based the generator on EDT - Estelle Development ToolSet [12], as it is the only environment for Estelle that has support and maintenance. We have taken advantage from the EDT compiler - ec. Ec generates an *intermediate form* [13] from Estelle text, which we use as input for the generator. Thanks to it, we have neither to parse an Estelle text nor to check its syntactic correctness.

This chapter gives some insights into implementation of our generator.

4.1 Additional Constraints

To get first results, and to analyze the feasibility of the Estelle `->` STREAMS generation rather than focus on full implementation, we have added the following constraints for input Estelle specification (in addition to these mentioned in previous chapter):

1. The source module body definition cannot have any internal interaction points nor children modules.
2. There are two restricted statements: *any* and *forone*
3. *exist* factor in expression is not accepted.
4. Nested Pascal functions are not accepted.

All of these constraints can be relaxed in the next version of the generator. Translation of the above constructors into C code is laborious and time consuming but feasible. For example the generator from EDT that uses BSD sockets processes all of them. A collaboration with EDT providers would allow to reuse the Estelle compiler code to rapidly relax the mentioned constraints.

4.2 STREAMS Interface

With the first condition met, there is no need to implement any internal engine handling interaction-passing between children modules. Thus all of the interactions sent and received by the Estelle module can be translated directly into e-messages.

An e-message in our generator is a STREAMS message of `M_PROTO`[2] type. This message has normal priority. The first 4 bytes of its data block contain a unique magic number, so it can be identified by other e-modules. The rest of the data block contains e-message type and the interaction data: its name and its parameters, as presented on Fig. 2. There are two e-message types: `E_INTER` for interaction data, and `E_ERROR` for error notifications.

[2] Messages carrying protocol control information.

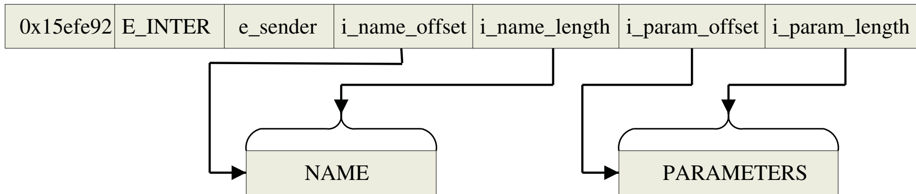


Fig. 2. E-message structure

The e-messages are recognized by their name. The designer has to keep that in mind when joining e-modules generated from different Estelle specifications into a single stream. He should make sure that Estelle modules that will be connected with each other should have the same channel definitions.

The application (user-level process that uses the stream) should understand e-messages of the topmost e-module in the stream, i.e., it should know interaction names of the highest level protocol specification.

The driver (lowest module in a stream) should understand e-messages from the lowest layer of protocol. However, this is rather not the case, because drivers cannot be generated directly from Estelle specifications and usually have their own interfaces. Because of this, we need a special "translating" module, which will accept e-messages and output driver-interface messages on its write side, and do the opposite on its read side[3]. In our generator, such a module translates e-messages into DLPI [14], which is used by the Linux STREAMS [11] driver, *ldl*.

4.3 Generated Module Structure

A STREAMS module generated from an Estelle specification consists of three parts:

The module skeleton contains all standard STREAMS module code (*put()*, *service()* functions, *M_FLUSH* message handling, etc.). The module entry points (*put* and *service*) call the functions from the other parts of the module. The skeleton is common for all e-modules. The only thing that changes in it is the module's name.

Automaton engine is common for all e-modules and is included in their code. This set of functions is responsible for detecting, processing and sending e-messages. It also handles *delay* clauses from Estelle transitions and of course keeps track of current automaton state.

[3] The "write" side of a STREAMS module sends messages downstream, from application to the driver, while the "read" side sends messages upstream, from the driver to the application. See [2].

Automaton body is a set of C functions that contain translated Estelle code from input file: functions and procedures, type and constant definitions, transition bodies. It also includes some data structures describing the e-module.

The following steps describe the way an e-module works. Algorithm is the same for messages going up and down the stream.

1. A STREAMS message is received.
2. The message's magic number is checked. If it's not an e-message, it's processed by the module skeleton code as a standard STREAMS message (usually passed further).
3. The interaction name is checked. If it's not known by the current module, an error notification message is generated upstream, to the application.
4. The interaction's parameters are validated. If they are not valid, an error message is generated.
5. On a base of a transition select table it is chosen a transition to be fired.
6. The transition's body is run. All generated interactions are translated into STREAMS messages and stored in a temporary buffer.
7. A check is made for spontaneous transitions going out from current state - recursion to point 5.
8. A check is made for delayed transitions going out from current state. If there are any, STREAMS *timeout()* call is issued.
9. All messages in temporary buffer are sent into the stream.

The automaton data contains a structure called transition select table. It is a two-dimensional C array, where the first dimension is the current automaton state, and the second dimension is the interaction number, determined by an interaction name. A special interaction number is defined for no interaction, i.e. for spontaneous transitions. Each element of the array is a list of transitions, sorted by priority (descending). Upon receiving an interaction, an appropriate list is searched, from highest to lowest priority. The first transition that has its provided clause met is being run. In this way we lose the non-determinism of the Estelle semantics, but still we assure that the transition with highest priority is fired.

Delay clauses are implemented with the use of STREAMS *timeout()* function, as described in chapter 2. If the automaton's state changes before the timeout callback is done, the timeout is canceled.

5 Efficiency Tests

We have performed some efficiency tests to see how generated STREAMS modules work. First test was made on Linux system, and its aim was to see how the number of modules in the stream (in other words, number of protocol layers) influences the overall performance of an application. Because its results were not as good as we have expected, we have carried on our tests on Solaris operating system, where STREAMS are an internal part of the kernel. These tests were aimed at comparing the efficiency of a generated module to the efficiency of a hand-written module with the same functionality.

Table 1. Measured transfer rate in KB/s - Linux

Stream structure	*loopback*	Ethernet
ldl, simple application	875	852
ldl-e2ldlmod, simple application	760	844
ldl-e2ldlmod-namesrv, simple application	665	616
ldl-e2ldlmod-namesrv, complex application	525	383
ldl-e2ldlmod-namesrv-resp, simple application	576	186
ldl-e2ldlmod-namesrv-resp, complex application	490	158

5.1 Linux Tests

On Linux system, we have measured average transfer rate of datagrams, each of 100 bytes size, using loopback and 10Mbit Ethernet link between 2 computers. The first one with Pentium III 550 MHz, 128 MB RAM, works under Linux RedHat 8.0. The second with Pentium 166 MHz, 64 MB RAM, with Linux RedHat 7.1.

The aim of the tests was to see how including a module into a stream influences the transfer rate, assuming that only the communication mechanism is considered, not any significant protocol processing. Thus we have written two simple Estelle specifications. The first (lower layer) was a "name service" protocol, translating LLC network addresses into logical addresses, and the upper layer was responsible for responding to received packets. To take measures we have built six configurations of a stream, and have run the test for loopback and Ethernet connection. Table 1 contains acquired results. In every configuration we have used the default LiS network driver, *ldl*. Between the driver and the lower e-module we have inserted a translation module, *e2ldlmod*. This separate STREAMS module translates e-messages into DLPI and vice-versa. It comes with the generator package, and cannot be integrated with generated modules in current version of the generator.

To control the stream, we have used two kinds of applications. The first was a simple single-threaded program that sent and/or received e-messages in a loop. The second one was multi-threaded and provided a handy API for handling the e-messages.

The transfer rates are low (compared to possible Ethernet transfer) because of the small packet size. The results show that the loss of efficiency from adding a new STREAMS module is about 10-20%. Lower efficiency of complex application was caused by the need of rewriting the data and storing it in temporary buffers to provide the required API.

5.2 Solaris Tests

For tests on Solaris, we have used two machines:

1. AMD Athlon 2,0 GHz, 392 MB of 333 MHz DDR RAM, 2,5 GB Samsung IDE Hard Drive, Realtek 8139 based 10/100Mbit Network Adapter

2. AMD K5-2 350 MHz, 64 MB PC100 RAM, 2,1 GB Samsung IDE Hard Drive, Realtek 8139 based 10/100Mbit Network Adapter

For networking we have used Fast Ethernet with 10/100Mb switch (Surecom EP-805X-R, 5 port).

The tested specification was a simple echo protocol, serving as both echo client and echo server: packets incoming from upper layer were flagged as "echo request" and sent to lower layer, while packets coming from lower layer and flagged as "echo request" were sent back with the flag changed to "echo response". Packets coming from lower layer and flagged as "echo response" were forwarded (with removal of the flag) to the upper layer.

As we have already mentioned, tests were performed on an automatically generated e-module and on a hand-written module with the same functionality. The generated e-module had to have the translating module, *e2ldlmod*, inserted into the stream as well. The purpose of this module was explained earlier for Linux tests.

The result of the test was an average file transfer rate between two machines. All results are presented in Table 2. Tests were made for two hardware configurations and with 3 different data packet sizes.

Presented results show the difference between manual and automated implementation. It is dependant on the packet size: the bigger the packet, the smaller relative performance loss. For 100-byte packet the generated module was about 20% slower than the manually written one, while for 800-byte packet it was only 10% slower. The difference between both implementations is shown on Fig. 3. Further increasing of the packet size may reduce the loss of performance even more.

The generated echo protocol consisted of two modules, one that performs the protocol (generated from Estelle specification) and one for translating messages from one STREAMS interface to another. Future versions of the generator may avoid this by integrating the translating functionality into the generated module, which should improve the performance of generated modules.

Table 2. Measured transfer rate in KB/s - Solaris

		Data transfer rate (kB/s)		
Hardware configuration	Echo module	100B packet	200B packet	800B packet
Machine 1 -> Machine 2	Generated	1085	1771	3197
Machine 1 -> Machine 2	Manually written	1446	2210	3630
Machine 2 -> Machine 1	Generated	806	1377	2722
Machine 2 -> Machine 1	Manually written	1028	1725	3032
Average	Generated	945	1574	2960
Average	Manually written	1237	1968	3331

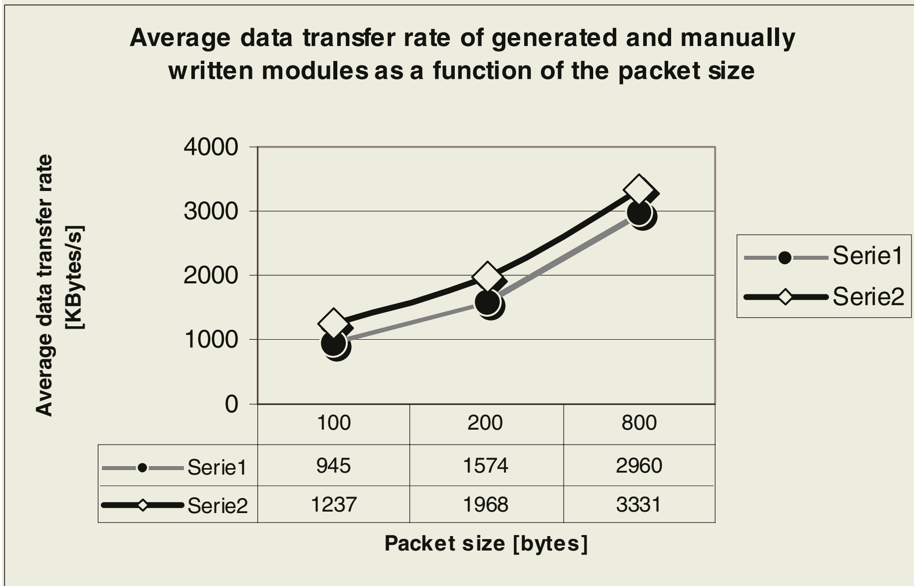


Fig. 3. Difference in transfer rate between generated (Serie 1) and manually-written (Serie 2) modules

6 Conclusions

The usual problem with automatic generation is the effectiveness of the output code. Most of existing generators produce programs that are significantly less effective than hand-written code. Moreover STREAMS implementations, that are the most commonly used technique in commercial Unix systems, are not addressed by these generators. This fact greatly reduces the usage of existing generators, and formal specification is mostly used only for documentation and validation purposes. The main idea of our work was to check the feasibility of STREAMS module generation. The obvious advantage of the Estelle `->` STREAMS generator is the simplification in design of rapid and correct implementation, taking advantage from formal specification and validation techniques.

The created tool demonstrates that the automatic STREAMS module generation from an EFSM description is possible. However a designer has to keep in mind this kind of implementation during the development of the specification. We have defined constraints that he should preserve.

The results of the performed tests demonstrate that further work should be done to obtain desired functionality and efficiency. First of all the assumed constraints (defined in chapter 4.1) should be relaxed. Secondly our tool should be optimized and enhanced. The enhancements could provide some guidance about cooperation with existing and STREAMS modules.

Whenever automatic generation of a STREAMS module is performed or not we argue that modeling and verification of the module functionality has to be performed in order to build a correct communication system.

Acknowledgment

We would like to thank Mr Marek Jóźwik for his help in realization of efficiency tests, especially those performed on Solaris platform.

References

1. Ritchie, D.: A stream input-output system. AT&T Bell Laboratories Technical Journal (1984) 63, 8 Part 2, s.1897-1910.
2. Sun Microsystems, Inc.: Solaris AnswerBook: STREAMS Programming guide. (1998)
3. O. Catrina, A.: On the improvement of the estelle based automatic implementations. (1998) in: S. Budkowski, A, Cavalli, E. Najm (Edts.), Formal Description Techniques (XI) and Protocol Specification, Testing and Verification (XVIII) [FORTE / PSTV], Kluwer Academic Publishers, Paris - France, pp 371-386.
4. Thees, J.: Protocol implementation with estelle - from prototype to efficient implementations. (1998) in: S. Budkowski, S. Fischer, R. Gotzhein: Proc. of the 1st International Workshop of the Formal Description Technique Estelle (ESTELLE'98), Evry, France,.
5. Bredereke, J.: Specification style and efficiency in estelle. (1998) in: S. Budkowski, S. Fischer, R. Gotzhein: Proc. of the 1st International Workshop of the Formal Description Technique Estelle (ESTELLE'98), Evry, France,.
6. Gotzhein, R., et al.: Improving the efficiency of automated protocol implementation using estelle. Interner bericht nr 274/1995 (1995) Fachbereich Informatik, Univeristat Kaiserscautern.
7. Langendörfer, P., König, H.: Improving the efficiency of automatically generated code by using implementation-specific annotations. (1997) Participants proceedings of the 3rd International Workshop on High Performance Protocol Architectures. HIPPARCH'97, Sweden.
8. R. Henke, H. König, A.M.T.: Derivation of efficient implementations from sdl specifications employing data referencing, integrated packet framing and activity threads. (1998) in: proceedings of Eighth SDL Forum, North-Holland.
9. Henke, R., Mitschele-Thiel, A., König, H.: On the influence of semantic constraints on the code generation from estelle specifications. FORTE/PSTV Osaka (1997)
10. ISO/TC97/SC21: Estelle: A Formal Description Techinque Based on an Extended State Transition Model. (1997) ISO 9074.
11. Gcom, Inc.: Linux STREAMS home page. (2002) ¡http://www.gcom.com/LiS¿.
12. S.Budkowski, et al.: The Estelle Development Toolset. Institut National des Telecommunications, Evry, France. (1998) ¡http://www-lor.int-evry.fr/edt¿.
13. Moraly, R.: Intermediate form utilization principles. INT. (1998) Document is available to download on the EDT distribution page.
14. Unix International: Data Link Provider Interface version 2.0.0. (1991)