The *eXperimental Estelle Compiler* - Automatic Generation of Implementations from Formal Specifications

Joachim Thees
University of Kaiserslautern
Postfach 3049
D-67653 Kaiserslautern, Germany
+49 631 / 205-4408
thees@informatik.uni-kl.de

Reinhard Gotzhein
University of Kaiserslautern
Postfach 3049
D-67653 Kaiserslautern, Germany
+49 631 / 205-3426
gotzhein@informatik.uni-kl.de

1. ABSTRACT

An important aspect of the application of formal methods in software practice is the automatic creation of efficient implementations directly from formal specifications. This often allows to save (or at least to simplify) the costly and error prone step of manual coding.

In this paper, we introduce the *eXperimental Estelle Compiler (XEC)*, a new implementation generator for the specification language Estelle. This tool is experimental in the sense that it has been developed as a platform for the performance-evaluation, optimization, and testing of implementation methods. The special structure of generated implementations allows a very flexible execution model, supporting extensive static and dynamic optimizations. Finally, we report on a case study with the Xpress Transport Protocol (XTP), including quantitative performance data of different implementation methods in comparison to other Estelle code generators.

1.1 Keywords

Specification Languages, Estelle, Protocol Implementation, Automatic Code Generation, Tool Support, Performance Optimization, Performance Monitoring, XTP.

2. Introduction

Main requirements on protocol implementations are *correctness* and *efficiency*. Formal Description Techniques (FDTs) are being advocated to secure the compliance with correctness requirements. With their formal syntax and semantics, they enable a precise description of systems and support the verification of functional and non-functional aspects. Nevertheless, the creation of efficient implementations is still largely based on hand coding, even if tools for an automatic generation of implementations directly from formal descriptions are available. This is caused by the reputation of these tools to create implementations with outstanding low performance and by the difficulties that arise when the generated code is to be integrated into real-world environments, such as operating systems, networks and user processes. There is still little work on quantitative performance-evaluations of automatically generated implementations or on quantifications of the optimization potential of such implementation tools. The performance quantification and optimization of automatically generated implementations could be an important factor to increase the acceptance of such tools and of FDTs in general for practical applications.

In this paper, we introduce the ***eX**perimental **E**stelle **C**ompiler* (*XEC*), a new implementation generator for the FDT *Estelle* [5, 2]. XEC is a fully operational Estelle compiler, which has been especially designed as a platform for the performance-evaluation, optimization and testing of implementation methods. Special features in this regard are its flexible execution model and its support for performance monitoring.

In Section 3, we introduce some of the concepts of Estelle. In Section 4, we elaborate how these concepts are represented by XEC and how this distinguishes XEC from other Estelle code generators. Section 5 reports on a case study with the *Xpress Transport Protocol* (*XTP*), including quantitative performance data of different implementation methods in comparison to other Estelle code generators.

3. The FDT Estelle

Estelle [5, 2] is an FDT, which is internationally standardized since 1989. It has been designed for the description of *distributed, communicating, concurrent systems*, in particular communication protocols.

3.1 The Structure of Estelle Specifications

Basically, Estelle specifications describe a dynamic set of communicating *extended finite state machines* (EFSMs) that operate concurrently. Each of these EFSMs is an instance of a *module*, which is the basic delimited structuring means of Estelle. Modules have a well-defined external interface, composed of a set of typed external *interaction points* (for communication via asynchronous message passing) and *exported variables* (for communication via shared variables). Internally, modules consist of their local internal description and a set of (also delimited) child-modules, building a hierarchical tree of modules (see Figure 1). The local part of the internal description is given by a set of state-components (e.g. local variables and an enumerated *control-state*) and a set of *transitions* (see Section 3.2).

```
specification S;
    {...}
    body A {...}
        body B {...}
            end;
        body C {...}
            end;
        end; {body A}
    body D {...}
        end;
    end. {specification S}
```

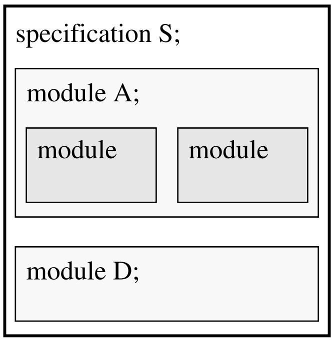

Figure 1. Static Module Hierarchy

During the execution of an Estelle specification every *instance* of a module may create new instances of child modules or terminate some of its child module instances. This leads to a dynamic tree of module instances, whose root node is an instance of the specification module[1] (see Figure 2).

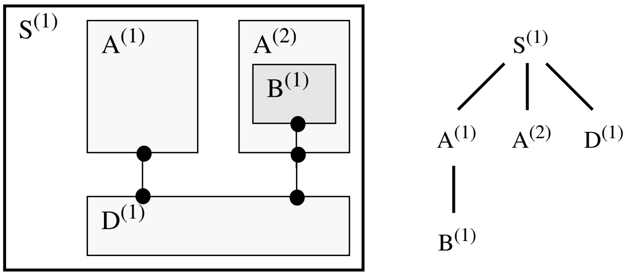

Figure 2. Dynamic Module Instance Hierarchy and Connection Structure

The dynamic module structure of Estelle is complemented by a *dynamic connection structure*[2]: Every module instance dynamically controls the connections between the external interaction points of its child module-instances and its own (local or external) interaction-points. The (bidirectional) communication between the final interaction-points of a chain of such connections is achieved by *asynchronous message passing*: a typed message (“*interaction*”) sent into one of these interaction points is appended to the unlimited FIFO-buffer associated with the interaction point at the opposite end of the chain of connections. Later on, the receiving module instance can accept this interaction by firing a *transition* with an appropriate WHEN-clause.

3.2 Transitions

Transitions are the means to describe the behavior of a module instance. Each transition describes the necessary local preconditions for its firing (by a set of *“transition-clauses”*, see Table 1) and the effect of its firing, which is mainly described by a Pascal-like statement block (see Figure 3).

Clause	Precondition to Firing	Effect of Firing
FROM	control state matches	—
TO	—	control state changes to given one
WHEN	type and destination of interaction in front of FIFO buffer (associated with given IP) matches	interaction removed from FIFO buffer
PROVIDED	boolean expression evaluates to `true`	—
PRIORITY	priority of transition	—
DELAY	transition enabled since stated number of time units	—
ANY	—	—

Table 1: Transition Clauses

The `ANY` clause allows to specify a set of transitions at once that only differ in the value of an ordinal constant, e.g. “`ANY i: 1..10`” defines 10 transitions that only differ in the value of *i*. The `ANY` clauses can be expanded syntactically.

```
TRANS
    FROM    unconnected
    TO      connected
    WHEN    IpToPartnerPM.ConReq(bConf: BOOLEAN)
        BEGIN
        nConnects := nConnects + 1;
        IF bConfirm THEN
            OUTPUT IpToPartnerPM.ConConf;
        END;
```

Figure 3. Example of an Estelle Transition

[1] The instance of the specification module is implicitly created at the beginning of the execution of the Estelle specification.

[2] In this context, the term “connection” subsumes the Estelle term “attachment”.

3.3 Execution Model

The execution of an Estelle specification basically consists of alternating phases of the *indeterministic selection* of a set of fireable transitions and the *atomic firing* of the selected transitions. The selection process recursively determines at most one fireable transition for every module instance. Only if no fireable transition can be found in a module instance, the selection is recursed into its child module instances. This implies a father-son-priority, i.e. a module instance can only select and fire a transition if its father instance has no fireable transitions.

3.4 Automatic Generation of Implementations from Estelle Specifications

One of the most important aspects of Estelle w.r.t. *practical applicability* is its implementation oriented type-structure and behavior description. In contrast to other FDTs that use property-oriented behavior definitions or abstract, recursive and axiomatic type-definitions, Estelle is relatively straightforward to implement: Estelle types essentially are Pascal types, and therefore have fixed structure; they can be directly mapped into the type-model of implementation languages such as C. The same holds for the behavior description of Estelle specifications, which is basically given by transitions containing a sequential, imperative, Pascal-like statement-block (see Figure 3). These transitions are also well suited for a direct implementation.

There are already some tools for the automatic implementation of Estelle specifications (e.g. EC [7] and PET/DINGO [8]). However, these tools either in fact create implementations with very low performance (DINGO, see Section 5) or are commercial (EC), and therefore cannot serve as a base for our performance evaluation and optimization experiments, because their sources are not broadly accessible.

From our previous extensive experiences with Estelle code generators (including EC, PET/DINGO), performance monitoring, and efficiency improvement through parallel implementation [4], we think that a *platform for experiments with efficient code generation* should support the following:

- Application of performance oriented coding schemes for generated code and run time system.
- Flexibility w.r.t. the generation of code and the execution model in order to support the application of static as well as dynamic-adaptive optimizations.
- Small conceptual distance between generated code and specification. This would allow to establish a predictable influence of the specification style on the generated code and, as a consequence, on its efficiency.
- Support for the integration of generated implementations into preexisting, non-Estelle environments.
- Support of differenciated performance monitoring and execution statistics with controllable overhead.

In order to obtain a platform with these properties, we have come to the conclusion that a *complete new development* of an Estelle code generator would be the best approach.

4. The 'eXperimental Estelle Compiler' *XEC*

To build a platform for the *testing, performance-evaluation, and optimization of implementation methods* for Estelle, we have developed the ***eX**perimental **E**stelle **C**ompiler (XEC)*. It should be noted that XEC is a fully operational and well tested Estelle compiler, which is "experimental" merely concerning its applicability as platform for experiments with implementation methods.

The implementation generator builds upon the Estelle compiler front-end PET [8] and translates Estelle specifications into objectoriented C++ code. The strictly object oriented modeling of the structure of Estelle permits a very flexible execution model, supporting extensive *static and dynamic optimizations*. Most of the dynamic optimizations can be implemented inside the object oriented run-time library for the generated implementations. This reduces the complexity of the code generator itself, and therefore also the risk of errors due to complex compiler optimizations.

Despite of their high abstraction level the automatically generated implementations are enabled to be efficient, as a result of performance oriented coding of the run time library and generated code, and especially by the selective use of inline code. In Section 5, we will see that XEC already with some few, first, and rather simple dynamic optimizations methods can compete with the Estelle compiler EC, which applies a lot of optimization techniques.

4.1 Representation of the Specification Structure

One of the most conspicuous properties of the code generated by XEC is the fact that it extensively preserves the syntactical structure of the underlying Estelle specification by mapping the hierarchical structure of the specification (consisting of nested modules, transitions, procedures, etc.) into an isomorphous structure of *nested C++ class definitions* (see Figure 4). This technique allows both to *preserve the structure* of the specification, as well as to *project the scope rules* of Estelle into the generated C++ code.

The nesting of C++ classes (i.e. the definition of classes as members of other classes) defines a hierarchical name space in which every (named) definition has a globally unique *qualified name*. For example the qualified name of the C++ class representing body B in Figure 4 is `SPEC_S::BODY_A::BODY_B`. Obviously, there is no need to create artificial extensions for identifiers (e.g. continuous numbers like in code generated by EC or DINGO) to make them unique[3]. This allows to project the

[3] The *classification prefixes* created by XEC (e.g. `SPEC_`, `BODY_`, `TYPE_`, `VAR_`, `CONST_`, `TRANS_`) are mainly used to increase the readability of the generated C++ code and could be omitted.

```
specification S;
    body A
        body B
            end;
        end; {body A}
    body C
        end;
    end. {specification S}
```

XEC

```
class SPEC_S {
    class BODY_A {
        class BODY_B {
        };
    }; /* class BODY_A */
    class BODY_C {
    };
}; /* class BODY_S */
```

Figure 4. Principle of XEC's Representation of Estelle Specification Structure

naming scheme of Estelle into C++ in a natural and unique way.

In particular this naming scheme enables several specifications to be compiled and linked separately into a single program. In such a constellation possibly multiple instances of these specifications can coexist in one single process. This is an important prerequisite for the effective application of the Estelle extension *Open Estelle* (see Section 4.5).

4.2 Run Time Library

Another technique used by XEC to reduce the complexity of the generated code is the consequent transfer of functionalities and mechanisms to the object oriented run time library. Especially the application of class- and function-templates allows to keep the generated C++ code on an *abstraction level* that is very close to the abstraction level of the original Estelle specification.

This shows up mainly in the compact and rather easy to read coding (typically one line of Estelle source code is translated into one line of C++ code) and in the type-safety of the generated code (it manages — in contrast to EC and DINGO — virtually without explicit type casts). Furthermore the generated C++ code does not contain any elements concerning a specific execution mechanism (e.g. how to select modules and transitions). These aspects are completely handled inside the run time library by means of *abstract, polymorphic base classes*, which are inherited by the classes generated by XEC to represent the *concrete* modules, transitions, and so on. The classes representing concrete modules and transitions just have to adapt the properties of their abstract base classes, e.g. by notifying the set of transitions (in case of modules) or by overloading some virtual functions representing transition clauses (in case of transitions; see Section 4.3).

These techniques permit a high abstraction level and a clear separation between the generated code, which models the explicit state- and behavior-definitions given inside the Estelle specification, and the run time library, which provides the mechanisms for its execution. This is an important factor for the reliable implementation of semantically correct dynamic optimizations, since it allows to implement most of them inside the run time library and therefore to minimize the more complex and error prone compiler modifications.

4.3 Representation of Estelle Transitions

As an example for the coding scheme of XEC, we will have a closer look at the representation of transitions. As already noted in Section 3.2, a transition consists of a set of *transition clauses* and a *transition block*. Transition clauses mainly describe the preconditions for the selection and firing of the transition. All clauses are optional, and there are default conditions and effects for the clauses that have been omitted: e.g. a missing PROVIDED-clause means "`PROVIDED true`", a missing TO-clause keeps the current control state, and so on.

To represent transitions as close to the Estelle source as possible and to acquire high flexibility for dynamic optimizations, XEC represents each concrete transition as an instance of a class, which overloads in each case a virtual function of its base class for every given clause (except the ANY clause) and one for the transition block (see Figure 5)[4]. Consequently, the code generated for a transition directly reflects the structure of the original Estelle transition: the transition is represented by a delimited structure (class) that contains an element (member function) for the transition block and each transition clause given in the Estelle specification.

Transition clauses not specified in the Estelle source are simply omitted in the generated code, since their respective virtual functions defined in the base class of the transition (e.g. `class SimpleTrans`[5] in Figure 5) already represent the defaults for each clause; for example the default function `SimpleTrans::checkProvided()` just always returns `true`, because a missing PROVIDED clause is always fulfilled. This makes it possible for the run time library to abstract from the actual clauses and to handle all transitions in a uniform way (see Section 4.4).

---

[4] The C++ code fragment shown in Figure 5 shows just the principles of the generated code. Among other things the original C++ code separates declarations and definitions of possibly extensive member functions, encodes static clauses (FROM, TO, and PRIORITY) as base class parameters, and contains additional class members, e.g. for optimization, profiling, visualization, and debugging purposes.

[5] To increase the efficiency of the transition selection process there are specialized base classes for transitions with WHEN clauses (`class InputTrans`) and DELAY clauses (`class DelayedTrans`).

```
trans
    from s1
    provided n <= max
    name test:
        begin
        n := n+1;
        end;
```

XEC

```
class TRANS_test : SimpleTrans {
  public:
      bool checkFrom(int nState)  {return nState == STATE_s1;}
      bool checkProvided()   {return VAR_n <= CONST_max;}
      void exec() {VAR_n = VAR_n + 1;}
}; /* class TRANS_test */
```

Figure 5. Principle of XEC's Modeling of Estelle Transitions

The representation of the transitions of a module instance as objects (i.e. instances of their defining classes) furthermore allows to implement ANY clauses simply by iterative instantiation of the class that defines the transition. Every instance represents one unique value of the domain of the ANY clause. A transition without an ANY clause has just one instance in its module instance. This allows once again to keep the abstraction of the transitions rather simple for the run time library, since ANY clauses need no special treatment as soon as the instances have been created. Furthermore, this technique allows to differentiate between the run time characteristics of the single instances with regard to adaptive dynamic optimizations.

4.4 Execution Model and Dynamic Optimizations

The Estelle semantics basically separates the execution of a specification[6] into two alternating phases: (1) The *selection* of a set of fireable transitions and (2) the *firing* of the selected transitions. Consequently, the creation of an optimized implementation has to cover both, the selection step and the execution step. Our approach to optimize an implementation is based on two orthogonal techniques:

- performance oriented and specification related coding and
- dynamic optimization of the transition selection process.

The current version of XEC makes no attempt to *actively* optimize the *execution* of statement blocks (e.g. transition blocks); instead, the generated code as well as the run time library are carefully designed to support an efficient execution and support the optimization techniques of current C++ compilers. Especially the aimed use of inline code and the natural and compact 1:1 encoding of the Estelle statements and expressions enables the C++ compilers to create specification related machine code and to apply the optimization heuristics originally designed for hand crafted C++ code.

Our performance monitoring and optimization experiments show that the selection process has a much higher optimization potential than the transition execution part: for practical applications in complex protocols (e.g. XTP, see Section 5), the transition selection typically consumes 60% to 90% of the total execution time when based on a straight selection process.

As a reference for optimization experiments, XEC also supports a *straight selection process*, which is similar to the one used by DINGO: basically it recurses through the tree of module instances and locally tests all transition instances to find a fireable one with maximum priority. The test for fireability and accordingly the firing of a transition is performed by evaluating each transition clause and by executing the transition block, respectively.

For most specifications, this method causes the number of transitions tested to be orders higher than the number of transitions fired. Depending on the module attributions and the degree of parallelism in the specified system, it is even possible that regularly all transitions have to be tested to select and fire only one of them.

More sophisticated selection methods allow to reduce the number of tested transitions significantly. The basic ideas are (1) to reduce the number of modules and transitions to be tested as far as possible and (2) to test these modules and transitions such that fireable transitions are found as early as possible (see Section 5).

4.5 Practical Application

XEC uses the *Portable Estelle Translator* (PET, [8]) as compiler front-end. PET translates an Estelle specification (e.g. the file '*x.stl*' in Figure 6) into an equivalent binary intermediate form ('*x.obj*'). XEC translates this representation to C++ ('*x.hxx*' and '*x.cxx*') and thereby creates a makefile to automatically compile and link the generated C++ code and the run time library with appropriate tools (e.g. the GNU C++ compiler *g++*).

The standard run time system for implementations created by XEC allows to use different *command line options*, e.g. to control the verbosity of the execution, to select different simulated or real time clocks for delay timers, or to control the creation of monitoring traces (see Section 4.6). During the execution, the run time system collects *statistical information*, e.g. the number and time consumption of transition tests and executions. Furthermore it is possible to interrupt the execution to get a well formatted *report* of the state of all module instances, including type and arguments of all interactions in input queues.

XEC does not only support the complete Estelle language, but is also capable to translate open systems specified in the

[6] Strictly speaking every *subsystem* (i.e. instance of a "system" attributed modules) executes this cycle and the subsystems cooperate by interleaving.

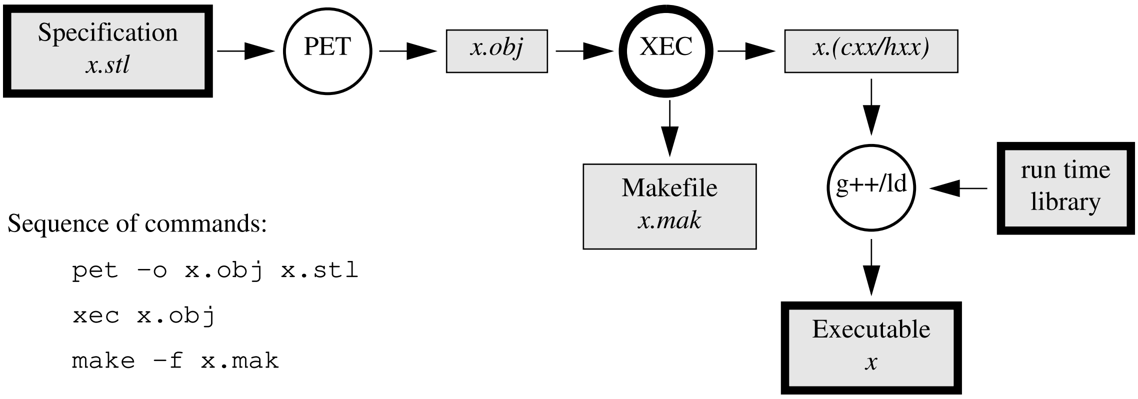

Figure 6. Translation of an Estelle Specification with PET, XEC and g++

Estelle extension *Open Estelle* [3, 9]. Open systems specified in such a manner have an external interface (in contrast to standard Estelle specifications), and therefore can directly interact with a manually coded or formally specified environment. In particular, XEC can translate open systems and environments separately down to machine object files, and can delay the fusion of open systems and environments until the linking step (*ld*).

The obvious naming scheme and clear structure of the generated code outlined in the previous sections also supports an immediate integration of generated code into hand crafted and finally also into preexisting environments (see also Section 5). For example, the compact representation of an Estelle specification as a tree of nested C++ classes makes it straightforward to create a program that concurrently runs several instances of (possibly different) Estelle specifications in *one single process*.

4.6 Performance Monitoring

An important aspect for the testing and the quantitative evaluation of implementation and optimization methods is the ability to gain precise *performance monitoring data* with minimized overhead. Therefore, we have adapted the performance monitoring tool *PATO* [6] to XEC.

PATO, which originally had been designed for the Estelle compiler DINGO, can gather execution traces of Estelle implementations and interactively evaluate these data with a graphical front-end. It is possible to monitor global activities like the processing of data packets on their way from end to end, as well as the time consumption of local events (down to the level of single statements or transition clause evaluations). Despite this subtle tracing resolution, the overhead of the monitoring can be kept under control by the possibility to determine the generation of traces precisely.

5. A Case Study with XTP

Beside extensive tests with small and medium sized Estelle specifications (among other things as part of several practical courses in our research group), we also performed a case study [1, 10] on a relatively large specification (about 7900 lines of specification text) of the Xpress Transport Protocol (XTP, [11]). This specification describes a complete end-to-end communication scenario with two user module instances, two XTP protocol machine instances and a network simulator (see Figure 7). The protocol machine instances contain of a dynamic set of *context modules*, which represent the end-points of the current XTP connections. The complex structure of the specified system leads to a total of 466 transition "instances" (see Section 4.3) in 18 module instances for only one end-to-end connection. The transmission of one user data packet from one end to the other one requires the firing of 51 transitions.

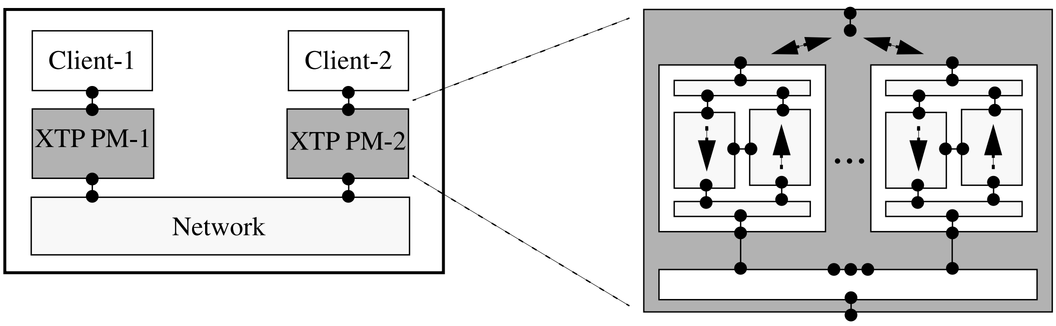

Figure 7. Module Instance Hierarchy and Connection Structure of the Estelle Specification of XTP

To compare the performance of XEC with EC and DINGO, we now have a look at the time it takes to transmit small (10 Bytes payload) and large (10,000 Bytes payload) packets over an existing XTP connection from one end to the other one. In this experiment, the network module transmits every packet without loss, modification, reordering or explicit delay. Table 2 shows the transmission times for the implementation created by XEC (applying the *straight selection mode*, i.e. without active optimization) in comparison to EC and DINGO.

Compiler	Optimization	10 Bytes	10,000 Bytes
DINGO	**none**	34.8 ms	108.6 ms
EC	**full**	4.0 ms	37.9 ms
XEC	**none**	9.8 ms	40.7 ms

Table 2: Total Times for End to End Transmission

In this comparison, DINGO and EC mark out the performance spectrum: from DINGO's simple, straight coding with low performance to EC's sophisticated optimization methods, which render a nearly 10 times higher performance (for small packets). In this context, XEC's basic mode (no optimization) demonstrates the effectiveness of performance oriented and specification related coding, even without active optimizations. Especially for large packets, where the transmission of data is much more time consuming than the selection phase (which is independent of the data packet size), the non-optimized XEC implementation can meet the performance of EC's optimized implementation.

XEC's monitoring data show that for small data packets, the transition selection phase consumes about 87% of the execution time, i.e. about 8.5 ms of 9.8 ms total transmission time. The main cause for this overhead is the fact that the straight selection method on the average tests about 161 transitions to select one for firing. Obviously, there is a high unused optimization potential.

Compiler	Optimization	10 Bytes	10,000 Bytes
XEC	***"inactive-tree"***	5.1 ms	35.3 ms

Table 3: Total Times for End to End Transmission

As a demonstration for the performance increment by dynamic optimizations, we now add the rather simple "*inactive-tree*" optimization. This optimization tries to determine sub-trees of the module instance tree, which had no fireable transitions in the last selection cycle. The modules in such an inactive tree can be excluded from further selection steps, until either one of them interacts with its environment (e.g. receives an interaction) or a delay-timer inside the tree expires. This rather simple optimization (just a few additions to the run time library) reduces the ratio between tested and fired transitions in our case study from 161:1 to 68:1. This reduces the amount of selection time during a transmission from 8.5 ms to 3.8 ms and therefore allows XEC to catch up with EC (see Table 3).

6. Outlook

We have introduced the *eXperimental **E**stelle **C**ompiler (XEC)*, a new implementation generator for the specification language Estelle. XEC is a fully operational Estelle compiler and is experimental in the sense that it has been developed as a platform for the performance-evaluation, optimization, and testing of implementation methods. Future work will focus on the effects of *dynamic optimizations* and *specification style* on run time efficiency.

We have seen that the selection of transitions has a substantial share of the time consumption of Estelle implementations (e.g. 87% with small packets in our case study). Consequently the transition selection is an important starting point for optimizations. Building on XEC, we are now going to implement and evaluate more sophisticated optimization methods. A promising approach is to reduce the number of transitions that have to be tested by extracting currently not fireable transitions until an event that may enable them (e.g. the receipt of a message) occurs. Another approach is the use of certain elements of the state space (e.g. control state or the type of the next interaction to be received) as an array index which directly pre-selects the set of possibly fireable transitions.

As a consequence of the small conceptual distance between code generated by XEC and the Estelle specification, it can be expected that the specification style directly influences the run time efficiency. We will also quantify this influence by means of further case studies.

7. Acknowledgements

This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) under grant Go 503/4-1.

The case study discussed in Section 5 was based on an Estelle specification of XTP 4.0 that was kindly made available by Stanislaw Budkowski (INT, France) and Octavian Catrina (PU Bucharest, Romania).

8. References

[1] Dahl, S.: *Performance Comparison of Protocol Implementations* (in german), Master thesis, Univ. of Kaiserslautern, Dept. of Comp. Sci., 1996

[2] Dembinski, P., Budkowski, S.: *Specification Language Estelle*, in: M. Diaz et al. (eds.), *The Formal Description Technique, Estelle*, North-Holland, 1989, pp. 35-75

[3] Gotzhein, R., Rößler, F., Thees, J.: *Towards "Open Estelle"*, in: 6. GI/ITG Workshop "Formal Description Techniques for Distributed Systems", Erlangen-Nürnberg, Germany, May 1996

[4] Gotzhein, R., Bredereke, J., Effelsberg, W., Fischer, S., Held, T., König, H.: *Improving the Efficiency of Auto-*

mated Protocol Implementation Using Estelle, Computer Communications, 19 (1996), pp. 1226-1235

[5] ISO/TC97/SC21: *Estelle - A Formal Description Technique Based on an Extended State Transition Model*, ISO/TC97/SC21, IS 9074, 1989

[6] Penner, H.: *Implementation of a Software Monitor for the Analysis of Automatically Generated Protocol Implementations* (in german), Master thesis, Univ. of Kaiserslautern, Dept. of Comp. Sci., 1996

[7] Richard, J. L., Claes, T.: *A Generator for C-Code for Estelle*, in: M. Diaz et al (eds.), The Formal Description Technique Estelle, North-Holland, 1989, pp. 397-420

[8] Sijelmassi, R., Strausser, B.: *The PET and DINGO tools for deriving distributed implementations from Estelle*, Computer Networks and ISDN Systems 25, 1993, pp. 841-851

[9] Thees, J., Gotzhein, R.: *A Formal Syntax and a Formal Semantics for Open Estelle*, Internal Report 292/97, Univ. of Kaiserslautern, Germany, Dept. of Comp. Sci., 1997

[10] Thees, J., Gotzhein, R.: *Performance Evaluation of Automatically Generated Protocol Implementations* (in german), 9. ITG/GI-Workshop “Messen, Modellieren und Bewerten von Rechen- und Kommunikationssystemen”, Freiberg, Germany, 1997

[11] XTP Forum, *Xpress Transport Protocol Specification*, XTP Rev. 4.0, XTP Forum, Santa Barbara, USA, 1995