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

J. Thees, R. Gotzhein,

University of Kaiserslautern, Germany

1. Introduction
2. The FDT Estelle
3. The ‘eXperimental Estelle Compiler’ XEC
4. A Case Study with XTP
5. Outlook

(formal) protocol specification

?

hand crafted implementation

automatically generated implementation

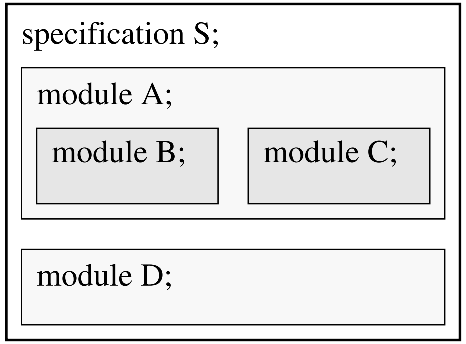
hand crafted implementation		automatically generated implementation
+	*efficiency*	-
+	*integration into pre-existing environments*	-
-	*correctness*	+
-	*development costs / times*	+

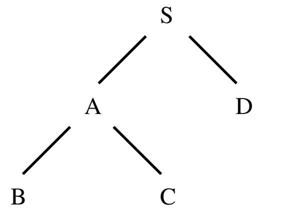
?

2. The FDT Estelle

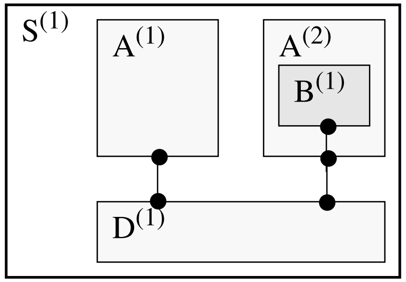
- ISO-standard (IS 9074) since 1989
- describes system of communicating EFSMs
- static module hierarchy

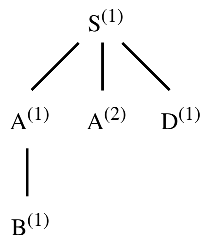
specification S; *{...}*
  body A *{...}*
    body B *{...}* **end**;
    body C *{...}* **end**;
  end; *{body A}*
  body D *{...}*
  end;
end. *{specification S}*





- dynamic module instance hierarchy and connection structure





- based on Pascal

  ⇒ directly implementable type- and execution-model

- behaviour described by *transitions*

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

- transition clauses: `FROM, TO, PROVIDED, WHEN, PRIORITY, DELAY, ANY`
  optional, default condition/effect if omitted

- existing automatic implementation generators

  * PET-DINGO *low performance*

  * EC *highly optimizing, high performance, **commercial***

Requirements on <u>platform for experiments</u> with efficient code gen.:

- *performance* oriented coding scheme
  (generated code / run time system)

- flexibility ➪ static & dynamic-adaptive *optimizations*

- small *conceptual distance* between specification and generated code
  (predictable influence of specification style on generated code / efficiency)

- support the *integration* of generated implementations into preexisting, non-Estelle environments

- differenciated *performance monitoring* and *execution statistics*
  (controllable overhead)

3. The ‘eXperimental Estelle Compiler’ XEC

- fully functional Estelle code generator
- platform for *testing, performance evaluation and optimization* of implementation methods
- uses PET as front-end, creates object oriented C++ code
- 1:1 encoding of Estelle specifications
  - example: preserves specification structure and name-spaces

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

$\xrightarrow{\text{XEC}}$

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

qualified C++ identifiers, e.g. ‘`SPEC_S::BODY_A::BODY_B`’

➪ no need for artificial extensions (e.g. numbers) for Estelle identifiers

- transfer of execution-mechanisms to the object oriented RTL
  * simplifies generated code and code generator
  * generated code highly abstracted (e.g. virtually no type casts)
  * dynamic optimizations implemented inside RTL
- represents transitions as objects

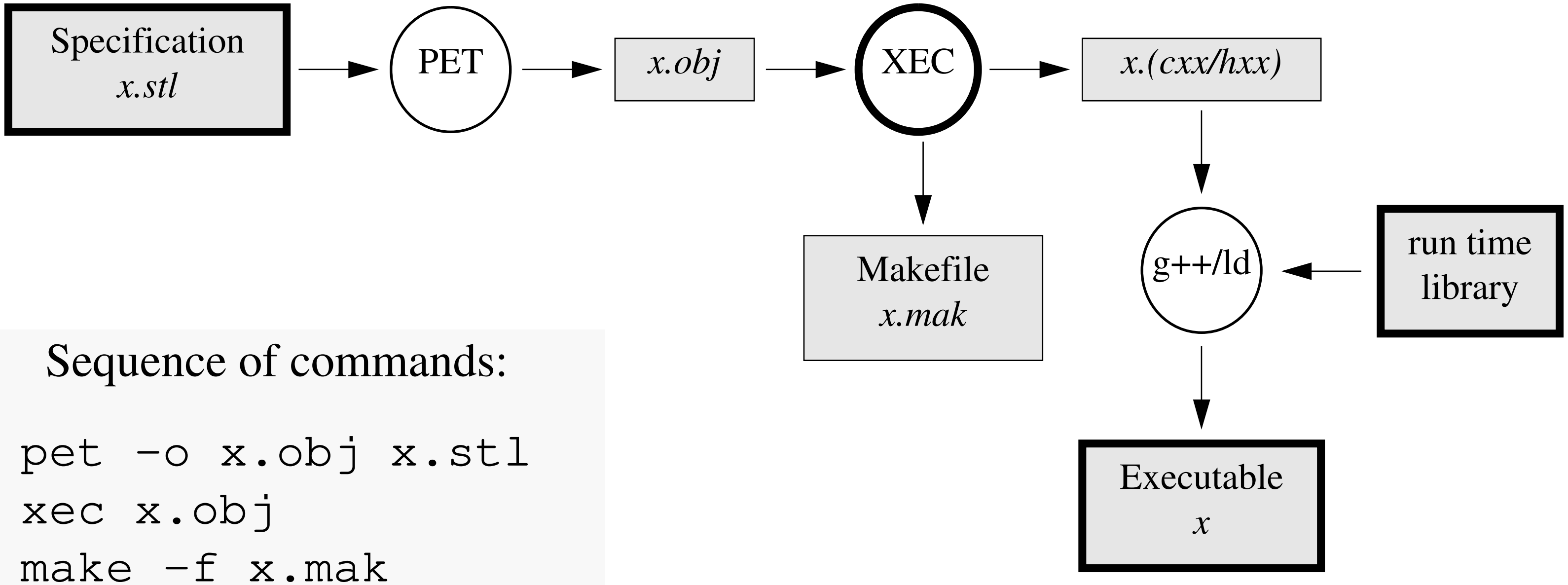
```
trans
    from s1
    provided n <= max
    name test:
        begin
        n := n+1;
        end;
```

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 */
```

- dynamic optimizations
  * Estelle's semantical model: alternating *selection* and *execution phases*
  * selection typically consumes 60% - 90% of execution time (XTP case study)
  * XEC supports *straight selection process* as reference

- practical application

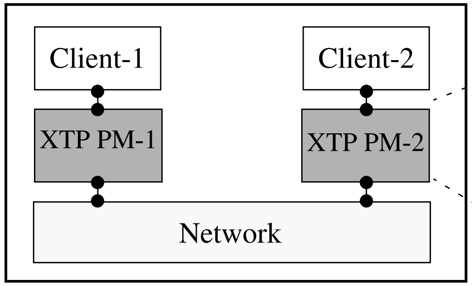


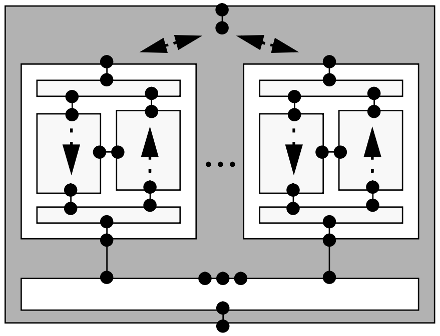
Sequence of commands:

```
% pet -o x.obj x.stl
% xec x.obj
% make -f x.mak
```

- performance monitoring
  - * *execution statistics* dumped at termination of generated implementation
  - * performance monitoring tool *PATO* with graphical front-end
- Estelle extension ‘Open Estelle’
  - * XEC supports independent compilation down to machine object files
  - * fusion of open systems and environments delayed to object-linking step (‘ld’)

4. A Case Study with XTP





- total times for end to end transmissions

	10 Bytes	10,000 Bytes
EC (full optimization)	4.0 ms	37.9 ms
DINGO (no optimization)	34.8 ms	108.6 ms
XEC (no optimization)	9.8 ms	40.7 ms
XEC ("*inactive-tree*" optimization)	5.1 ms	35.3 ms

- XEC's execution statistics

  * 18 module instances, 466 transition instances, 51 fired transitions per transmission
  * without optimizations: selection-time 87% (8.5 ms / 9.8 ms), test-fire ratio 161:1
  * 'inactive-tree'-opt.: selection-time 74% (3.8 ms / 5.1 ms), test-fire ratio 68:1

5. Summary / Outlook

- eXperimental Estelle Compiler (XEC)
  - * fully functional Estelle code generator
  - * platform for *testing, performance evaluation and optimization* of implementation methods

- optimization of automatically generated implementation
  - * transition selection has substantial share of time consumption (87%)
    ➪ high optimization potential
  - * e.g. event driven selection process

- influence of different specification styles

- improvement of Estelle w.r.t. the efficiency of implementations