A multi-processor Estelle to C compiler to experiment distributed algorithms on parallel machines

Claude Jard, Jean-Marc Jézéquel

A MULTI-PROCESSOR ESTELLE-TO-*C* COMPILER TO PROTOTYPE DISTRIBUTED ALGORITHMS ON PARALLEL MACHINES*

Claude JARD, Jean-Marc JEZEQUEL

I.R.I.S.A. Campus de Beaulieu
F-35042 Rennes Cedex, FRANCE
E-mail : jard@irisa.fr, jezequel@irisa.fr

Abstract. This paper presents a first attempt to generate parallel code from Estelle descriptions. We have dealt with a simple context in which only a static subset of Estelle and an homogeneous target machine are considered. We begin to present and justify the concept of experimentation on distributed algorithms for which our Estelle compiler has been designed. Then we discuss how the Estelle constructs are mapped onto *C* structures and how they are interpreted by a distributed runtime kernel. A technical annex gives an idea of the current version of the tool, named Echidna.

1 Introduction

1.1 The experimentation concept to analyse distributed algorithms

Everybody is aware that correct distributed algorithm design and implementation is still a very difficult exercise. So, as well in research laboratories as in software companies, people agree about the necessity of validating the specification before implementing the distributed algorithm.

Validation tools are quite different in their forms and their abilities, but they always accept in input a formal description of the considered distributed algorithm (or protocol). They output data, which can be viewed as confidence levels on some properties of the algorithm.

For short, we can say that the designer may attack his algorithm by three complementary techniques. We list here their advantages and major drawbacks :

- the formal properties verification : it gives a definite answer about validity, but existing methods only can actually analyse simplified models of the considered protocol. This forces the distributed algorithm to be described at an high abstraction level, so its formal verification leaves open the problem of properties preservation during its refinement course.
- the protocol simulation, using a simulated (and centralized) environment : it can deal with more refined models of the algorithm and efficiently detect errors on a subset of the possible algorithm behaviours. The main difficulty in such a case is to describe formally and simulate the execution environment. This one is generally very simplified, because it wouldn't be realistic (nor interesting) to take into account all the parameters of a real system, as for example, the exact influence of message size on transmission delays, or the action durations (which are not computable without execution).
- the observation and test of the protocol prototype implementation : here, the execution environment is a real one. But as there is a lack of tools to observe a distributed system as a whole, it will be difficult to validate actually the protocol. Moreover, the prototype behaviour may closely depend on implementation choices (e.g. non-determinism resolution). So, it will also be difficult to generalize from the observation to the possible protocol behaviours.

*This work has been partially supported by the French program on parallelism of the CNRS/C^3, and within the ADP research team of the IRISA laboratory.

It appears clearly that those approaches are much more complementary than in competition, and that an advised designer would try to use all of them. To help him, we propose to define and justify an intermediate technique, called experimentation, to fill the gap between simulation and prototype implementation.

Experimenting aims at observing the algorithm execution in a real environment (i.e. on a real distributed system) while allowing the larger subset of possible behaviours to occur.

Some parameters do not need to be simulated anymore as they are provided by the underlying system. If this distributed system is general enough, providing a controled environment, we will be able to transpose some aspects of the experimental behaviours to any other system, more or less modulus the experimentation machine. Some experimental results about particular algorithms, called synchronizers, running on the Intel iPSC hypercube confirm this point of view [3].

Another advantage of experimention on parallel machines is that we can fully use the power of such machines, to make it feasible to validate large distributed algorithms made of few thousand processes.

But, as we are on a real distributed system, we have to set up special techniques to perform distributed observation and measurement acquisition.

1.2 The Echidna project : tools for experimentation

The Echidna project aims at providing tools to prototype distributed algorithms on parallel machines. It provides an Estelle compiler for multi-processor machines and a set of software tools to observe the behaviours of the distributed algorithm under experimentation.

The formal description models the distributed system as a whole : we only consider closed systems. Since we are not interested in producing code for heterogeneous networks (as in [14] or [12] for example), we can work on the full algorithm, where remote communication belongs to the formal model. This allows us to produce executable parallel code directly.

Having a rather good experience in studying Formal Description Techniques for protocols, we chose the Estelle language, now an ISO standard [1], to describe our distributed algorithms. As communicating automata is the natural semantics of loosely coupled distributed systems, Estelle is particularly well suited for our purpose. On the other hand, Estelle is a superset of the ISO Pascal, with some ADA concepts (class-instance mechanism, kind of modularity ...). This allows efficient compilers producing efficient code.

Numerous works have been done around Estelle concerning automated verification, simulation or prototype implementation (see the survey [4]). Several Estelle tools have been developed in the U.S. (NBS compiler), Canada, Europe, where two EEC projects (Esprit Sedos and Sedos-demo [7]) aim at building an Estelle Work-Station.

So, with the very same formal description, we will be able to verify, simulate, or even implement a distributed algorithm : we are ensured to work on the very same object.

Our contribution to that research area is to be able to translate automatically Estelle descriptions into parallel executable code for distributed systems in order to perform experimentation. In the framework of experimentation, the target parallel machine must be general purpose, powerful and with an easily customizable environment. Our machine model consists in a set of sites (processors or machines) only communicating by message passing through an end to end reliable communication network. There is no loss of messages. They are exchanged within a finite but unpredictable delay, and the communication channels are FIFO queues.

Different machines may implement this abstract experimentation station model. We have focussed our development on the Intel iPSC hypercube supercomputer (64 80386-processors with 4-MBytes of memory each, see [8] for a more detailled presentation), since it has been available in our laboratory for two years. We have also been interested in networks of SUN-workstations (above TCP/IP) and PC's, and we are now investigating the Transputer world through the Floating Point Systems T-40 hypercube machine held in Grenoble (France) and the T-node supercomputer.

For portability reasons, we decided to generate *C* code, all the considered machines having an efficient *C* compiler (which was not the case for Pascal). Mapping Pascal constructs of Estelle into

C is a tedious task but this allows to generate a compact code which can be easily interfaced with other programs and libraries.

The purpose of this article is to present a first attempt to generate parallel code from Estelle descriptions. We have dealt with a simple context in which only a static subset of Estelle and an homogeneous target machine (same processors and compilers) are considered. We present how the Estelle constructs are mapped onto C structures and how they are interpreted by a Distributed RunTime Kernel (DRTK). Tools for distributed observation are under development and will be the core of an other presentation (some ideas were presented in [10]). We give in annex an example of a short Estelle program, its C translation and the current version manual of the compiler (which begins to be widely used by parallel programmers). A detailed version of this paper is available as a french technical report [11].

2 The computation model

2.1 Our Estelle model

We present here our considered Estelle subset. We assume that full Estelle is known to the readers (see [5] and [6] for presentations).

A system is either a set of parallel sub-systems running asynchronously, or a set of tasks. Asynchronous parallelism is required between physical sites. It is the Estelle programmer's responsability to define mapping of processes on physical sites : if the first parameter of a module is an integer, it is interpreted as the node identifier where the process must be run, the actual *placement* being done during task creation (initialisation)[1].

We do not consider the dynamic part of Estelle : the system architecture remains unchanged after initialisation. So there are two classes of modules : the structuring modules (refinements) which have no transition part and disappear after the configuration phase, and the real processes (leaves) which perform transitions but without dynamic statements like init, connect, detach ... Several Estelle concepts may then be simplified : sharing variables between children of a module becomes impossible, and distinction between processes and activities is irrelevant.

Such a static subset has already been considered in Veda [9] and Xesar [13] tools and seems to suit well for the context of experimentation. It considerably simplifies our parallel implementation. Implementing full Estelle would be an interesting project in using Estelle as a programming language for parallel machines.

The last semantic restriction is the delay clause, which is not implemented. But, we have built a global time service to timestamp events for observation. This global time may be used to implement the delay clause if necessary.

There are other (syntactic) restrictions : embedding procedures and transitions is not allowed. We reject Estelle particularities as "any" $<$ *type* $>$ and the famous "...", and we do not handle module variable manipulation through *all, exist, suchthat* constructs. This is not essential restrictions and they will disappear when integrating our compiler to an Estelle workstation.

The compiler translates the formal description into a set of data structures (expressed in the C programming language), which fully describe the specification constants, types, channel definitions, module specifications and their associated body definitions. Those data structures are then compiled by the local C compiler, and linked with the DRTK to be loaded and run on the target distributed system.

The DRTK allows the execution of an Estelle program on a distributed system (see figure 1). It consists of three parts : the scheduler, the Estelle runtime and the system interface. We detail their features in the following sections.

2.2 The Distributed Driver

The entry point of the executable code is on each node the *main()* function, whose text is:

[*] Each node is supposed to own a unique identity (a positive integer), and one node owns the identity 0. A centralised machine is assumed to be node 0 of a one node network.

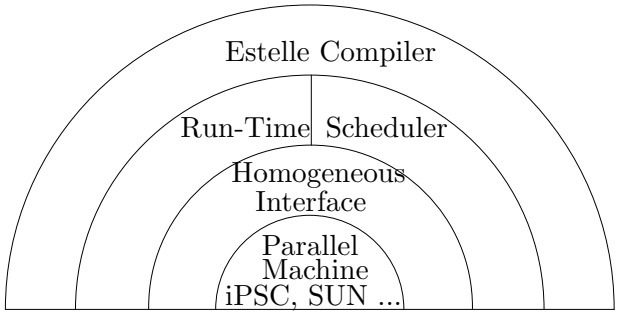

Figure 1: The Echidna compiler

```
main(){
  int seed,duration;
  getOptions();
  init(seed);
  configuration();
  startScheduler(duration);
  terminate(0);
}
```

getOptions() parses the command line to find options and parameters for the execution of the protocol, such as the duration of the execution, the seed for the random generator (in order to achieve reproductible experiments), or an alternate behaviour for the scheduler...

init() realises the general initialisation of the experiment.

configuration() is an external function, generated by the compiler. Upon invocation, it creates the root task of the Estelle specification and run its *initialize* part, which creates its child modules and recursively initialize them. When this resulting tree structure is achieved, only the leaf tasks that must be run on a node are actually set up in the active process list of this node to be scheduled by the local scheduler.

In the distributed execution of an Estelle specification, there are two levels of parallelism:

- inter-nodes parallelism which is the real parallelism of the machine (the distributed driver runs in parallel on each site)
- intra-node parallelism or pseudo-parallelism: the parallelism simulated by a local scheduler.

startScheduler() actually starts the task scheduling, according to the policy selected on the command line by the user, within deterministic or non-deterministic and synchronous or asynchronous. So the scheduler performs on each node the following simplified algorithm:

Repeat Until EndofExperiment

- Accept remote messages from outside (if some);
- For each process of this node, select first firable transition or choose at random one transition of all fireable transitions of maximal priority;
- For each process (or only for a subset of them if asynchronousity is simulated), fire the chosen transitions (if any);

end repeat

2.3 The Estelle Runtime

The aim of the Estelle Runtime is to build a virtual machine allowing the execution of an Estelle program, above the *C* runtime of the real machine. It provides the Estelle and Pascal primitive

functions and objects that do not exist in the C programming language: set manipulation, error detection and recovery, good random generator (for non-determinism), process manipulation ...

Furthermore, the runtime recognizes remote communications in order to route them through the network: when a message (interaction) is sent on an interaction point (ip), which is connected to an ip located on an other node, the message is routed by the underlying network (supposed to be FIFO without any loss nor alteration of messages) to the destination node. There, the function $AcceptRemoteMessages()$ invocated at each iteration of the scheduler, will accept the message and queue it at the destination ip.

This runtime is made of 1000 lines of C, and is easily portable on every machine where a classical C compiler is available.

2.4 The Interface with the System

The aim of this module is to provide an implementation of the parallel machine model (as defined in section 1.2), building an homogeneous interface with the underlying distributed system, whichever it is. It deals with remote communications, local clock, I/O and node identifiers.

If the real distributed system is close enough to the model, this module can be quite short (100 lines of C for the *iPSC/2*). But if not so close, this interface will be more complex, as for example with a SUN network (TCP/IP above *ethernet*), where we had to develop a software package from the *socket* concept.

Anyway only this part of the DRTK is to be adapted for each system, so the Echidna tool still remains easily portable.

When there is only one node, Echidna appears as an Estelle simulator which can be run on small computers (like PC).

3 The C representation of Estelle objets

3.1 The Estelle specification

The Estelle specification is a kind of black box without any external interaction.

If we look inside this black box during an execution, we can see some objects which interact each others:

- the tasks (or process), made of the association of a body to a module.
- the interactions points (ip), which are the interfaces between tasks and the outside world.
- the messages, or interactions, that tasks exchange on channels linking their ip's.
- the transitions, which are atomic guarded actions that can be performed by the tasks.

We will now describe in some details how those objects are represented with the C programming language.

3.2 The tasks

A task is made of a specification (*module*) and a *body*. The module is created upon declaration within its nesting module (its $father$). When the father module initialize its children, it provides them a *body* (see below the Estelle instruction *init*).

At run time, a task is represented by a control block which stores:

- some private data for the DRTK: process number, node identifier, activity state, transition descriptor list...
- the module ip's descriptors
- the child module descriptors (pointers to control blocks)
- the local variables of the process, including the main state of the automaton.
- the parameters transmited at initialisation time.

The C representation of this control block is implemented by the following generic type:

```
typedef struct context {
  ker_workspace kw; /*private space for the DRTK (invisible)*/
  union for all modules type of {
    list of (ip *)  lip;
    union for all the possible bodies of each module of {
       list of (struct context *) lt;
       list of (local variables) lv;
       list of (parameters) lpar;
    }
  }
} context;
```

3.3 The interaction points

The interaction points are the interfaces between tasks. They are used in a generic fashion and implemented as a DRTK private type. They are only known in the generated code as predefined descriptors.

An interaction point is made of:

- a unique ip identity over the whole system
- the node identity where the ip is located
- a pointer to its connected ip, if this ip is located on the same node.
- the unique identity of its corresponding ip
- the node identity where the corresponding ip is located
- a pointer to the task owner of the ip
- an unbounded queue storing messages received at this ip

The DRTK provides primitives to create ip's, to release them (when owner tasks are not initialised on this given node), to attach and connect them. The ip's which remain on a node after the configuration step are stored in a global list. Upon reception of an external message, this list will be searched in order to find out the destination ip. The size of an ip queue is only limitated by the available memory.

3.4 The messages

Messages are the interactions exchanged between tasks through their ip's. A message is made of:

- a static block holding:
  - the total size of the message
  - the destination ip identity
  - the message type
- a varying block, generated by the compiler to store the message parameters

The DRTK provides primitives to manage messages: create, delete, copy, local or remote send, add to an ip queue, and accept remote messages.

3.5 The transitions

An Estelle transition is a $< condition, action >$ pair. A *condition* is made of a (possibly empty) clause list, within *when* (message available on an ip), *provided* (boolean condition on the first message on the queue or on any local variable of the task), $from$ (condition on the main control state of the automaton), *delay* (specifying a time constraint for the firing of the transition) and *priority* (relative priority with respect to the other transitions of the *same task*).

An *action* is an optional change of the automaton main control state, followed by a compound Pascal statement, where specific Estelle instructions may take place (*e.g.* message sending...).

The Estelle langage makes it possible to gather different transitions under the same text (that we call meta-transition) with the *any* clause. An Estelle transition will be referenced by the number

of its meta-transition and a key identifying this transition within its meta-transition. This key looks like an access to a multi-dimensional array, and allows to set up the values for the indexes of the *any* clauses of the meta-transition.

We have chosen to generate only two C functions for each such meta-transition: evaluation and launch of the transition. Those functions take as parameters the task control block descriptor, and the transition identifying key.

Guard Evaluation The guard evaluation function starts setting up some temporary variables corresponding to the *any* clauses indexes, according to the parameter key. Then, it uses the functions provided by the DRTK to test the specified clauses. If they are all verified, it returns the priority of the transition, else it returns the value -1.

If we have a spontaneous transition without priority clause, the guard evaluation function is simply a NOP function which returns the minimal priority.

The *initialize* part of a module is represented by such a spontaneous transition, fired only once during the initialisation step of the task.

Transition firing The *action* part is translated in a C function, which, as the guard evaluation function, starts setting up *any* clauses indexes. Then it performs the associated compound Pascal statement, including the optional control state change.

4 The generated code

4.1 Non-Pascal instructions translation

First, let us see how instruction that appears within guards are translated:

When This clause is found in guards to test if a given message is available on an ip, and to open visibility on its parameters. It is translated within a guard evaluation function to:

```
{
# define ActualParameter_1 = m_in → FormalParameter_1
                    ...
# define ActualParameter_n = m_in → FormalParameter_n
  msg * m_in;
  if (((m_in = get_msg(ipDescriptor)) == NIL) ||
     (m_in → msg_kind != WantedType)) return(-1);
  /* tests on message parameters as specified */
}
```

From, Provided Those clauses allow to test the main control state of the automaton and any other local variable of the task. They are directly translated to proper C tests.

Priority As described above, the priority is computed and returned by the guard evaluation function.

Delay This clause is not currently implemented.

Any The unfactorisation of this kind of meta-transition is implemented as described in the previous paragraph.

Following instructions can only be found in the *action* parts of transitions.

Init This module initialisation instruction associates a body to the considered module instance. Here is its syntax:

$$init < ModuleInstance > with < body > (parameters)$$

It is translated to the following C block:

```
if (mk_proc(& context,Site,ContextSize,TransitionList,
                    SubModuleNumber,ipNumber) == OnThisNode) {
  context → FormalParameter_1 = ActualParameter_1;
                        ...
  context → FormalParameter_n = ActualParameter_n;
  init_proc(context);/*fire the task initialisation transition*/
}
```

Connect and Attach *Connect* allows a module to realise the connection between its sub-modules ip's, and *Attach* to link up a sub-module ip with a nesting module ip. Those instructions are implemented with direct invocations of corresponding DRTK primitives.

Output This instruction allows a task to send a message on one of its ip. Here is its syntax:

$$output < ipName > . < InteractionName > (MessageParameters)$$

It is translated to a *C* block which creates a message with the right tag selector, then assigns message parameters, and actually sends out the message on the selected ip:

```
{
  msg * m;
  m = mk_msg(MessageSize);
    m → FormalParameter_1 = ActualParameter_1;
                        ...
    m → FormalParameter_n = ActualParameter_n;
  send_msg(ipDescriptor,m);
}
```

To This instruction allows to modify the main state of the automaton. It is translated to a straight assignment on a local variable of the task called *State*, whose type is *enum*.

All The instruction *all* i : $LowBound..UpperBound$ *do* $< instruction >$ is a non-directivist loop, without previous declaration of the loop index. We implement it with a classical *C* loop.

Trace This instruction is our only extension to the Estelle language (which does not have any input/output instructions). Its syntax is the same as *writeln* in Pascal, but its semantic is specially designed within the experimentation context [2].

4.2 The generated code structure

The code generated by our compiler has the following form:

```
# include < echidnak.h > /* External declarations from the DRTK */
# define ( list of constants declared with Estelle keyword const )
typedef ( list of types declared with Estelle keyword type )
typedef struct {
  msg_type msg_kind;
  union {
      /* on the parameters of each possible message type */
       } u;
} msg;

typedef union s_context {
  ker_workspace kw; /* private space for the DRTK (invisible)*/
  union for all modules type of {
    list of (ip *) lip;
    union for all the possible bodies of each module of {
      list of (struct context *) lt;
```

```
        list of (local variables) lv;
        list of (parameters) lpar;
      }
    }
} context;


/* Here is the code for meta-transitions, for i := 1 to n */
int guard_i(a, p)
   int a; context * p; /* a is used to set up ANY-clause variables */
{
   /* test to perform guard evaluation */
   if firable return (priority) else return (-1);
}
int action_i(a, p)
   int a; context * p; /* a is used to set up ANY-clause variables */
{
   /* code to fire the transition */
}

/* Definition of variables to be exported to the DRTK */
/* The array transition[] gives the correspondance between a */
/* meta-transition text and its absolute number for the DRTK */
int (* transition[])() = {
          guard_1, action_1,
          ...
          guard_n, action_n
};
/* The array transany[] gives the variation bound for the */
/* any key for each transition */
int transany[n] = { nany_1, ..., nany_n };

void configuration()
{
   /* realise the nesting module creation (the specification) */
   /* and call its initialisation transition */
}
```

4.3 Pascal translation

On the syntactic level, the Pascal and *C* languages only differ in minor ways. The situation is worse on the semantic level.

Our translation is very similar to the public program PtoC written by Per Bergsten of the university of Gothenburg in Sweden.

Here is a list of the main problems we have encountered and their solutions :

- constants : number aliases are simply *define*d but string constants are converted to static character arrays, in order to avoid unnecessary duplication of strings in the object code.
- types and variables : integer subranges are mapped onto standard *C* arithmetic types according to a short table in the translator (scanned topdown until an enclosing subrange is found).

  C-arrays have peculiar semantics. Pascal arrays are encapsulated in a *struct* with a single member named *a*.

  Records and their variants are translated into C *struct* and *union* definitions. Artificial names (*u* and *v_xxx*) must be supplied for all record variants.

  The problem of recursive types as pointer types is solved by introducing a name (*s_xxx*) for the record type.

  Set types are translated in arrays of words. The first member gives the size of the set and the others hold the bits. The maximum handled size is 2048 elements.

- statements : the only parts that require special care are *with* and *for* statements. The with-statement is translated into nested compound statements, where pointer variables, referencing the corresponding records are declared and initialized. In order to evaluate once the record address, the accessible record fields are renamed within the scope of the with-statement.

  The for-statement requires that the loop boundaries are evaluated exactly once and must be exited when the upper bound has been reached. For that reason, the upper/lower bounds are held in local variables and a conditional break statement is added to the end of the loop.
- expressions : when the operands are sets, the expression is converted into a function call.

  The lower bound of the index type must be substracted when indexing.

  Pointer references and var parameters are handled by prefixing the expression with an asterisk. The special case of dereferencing followed by selection is also recognized.

4.4 Variables accesses and renaming

In order to generate reentrant code for transitions, the variables of Estelle bodies must be accesses through references to a process context. We have chosen a simple way to do this using the *C*-preprocessor : before generating the transitions of the given module body, the variables strings are *define*d to be references through a context pointer to the union structure coding all the Estelle bodies. They are *undef*ined when the transitions are generated.

Renaming is necessary since the block structure of the Estelle description is partially destroyed in the generated code. We chose to rename an user identifier only if its string is declared several times in the source text. In that case, it is prefixed by a unique number identifying its scope.

To prevent clashes of identifiers with the DRTK, the user identifiers are written with an initial uppercase letter, while the *C* identifiers and keywords begin by a lowercase character.

5 Implementation of the compiler

The compiler is written in Pascal and is seven thousand lines long. Parsing is performed using a left recursive descent of procedures. It consists of three major procedures that performs successively the conversion of the Estelle source program into a parse tree, some necessary transformations to prepare *C* generation, and the final traversal of the tree that prints the coresponding *C* constructs.

They are augmented by a set of procedures that maintain internal dynamical data structures (table for identifiers and strings, a multi-level symbol table, the parse tree...)

Full Estelle is parsed : the considered subset is defined by filtering. Most semantics checks are performed, either statically during parsing, or dynamically during execution. There is no other limitation on this subset: length of Estelle identifiers, number of messages, modules, interaction points (multiple dimension arrays of modules and ips are handled) and even the size of the specification are only limitated by the available memory. The compiler is quite fast (100 lines/s on a Sun), and can easily be run on a PC.

6 Conclusion

Validating a distributed algorithm is also experimenting it in order to better understand its real behaviour and performances, highlighting some phenomena generally non observable by verification nor by simulation.

The *ECHIDNA* tool implements this complementary approach of validation. *ECHIDNA* is now available for the Intel Hypercubes *iPSC/1* and *iPSC/2*, for Transputer based FPS-T40 Hypercube, for Sun Workstations Networks (with TCP/IP), Gould and PC. We are planning to make it also available for Apollo Workstations and other Transputer Networks (as for example the T-node machine).

As the input language for *ECHIDNA* is ISO Estelle, one can use a formal description of a distributed algorithm to run not only experimentation but also verification and simulation. Thus we are sure that the same distributed algorithm is studied during the three parts of the validation

process. This approach proved itself useful for some real problems, as for example the synchronizers study led at the IRISA, or the MAC/TR atomic fault resistant diffusion protocol validation (DELTA 4 Esprit Project): *ECHIDNA* allowed to get interesting measurements to compare real efficiency of different algorithm versions.

Furthermore the *ECHIDNA* ability to directly produce code for distributed systems makes Estelle a good choice as a high level parallel programming language for parallel machines (where there exist usually only *C* and Fortran environments). So, beyond validation activities, the main application fields considered for *ECHIDNA* are massive simulations, distributed systems and algorithms teaching, study and analyse, and even parallel prototype implementations.

If this interest for Estelle as a true parallel language is confirmed in the future, we think that it would be interesting to implement a full Estelle compiler for distributed systems.

References

[1] IS 9074. Estelle *: a Formal Description technique based on an Extented State Transition Model.* ISO TC97/SC21/WG6.1, 1989.

[2] M. Adam, Ph. Ingels, C. Jard, J.-M. Jézéquel, and M. Raynal. Experimentation on parallel machines is helpful to analyse distributed algorithms. In *Proceedings of the Workshop on Parallel and Distributed Algorithms, Bonas, France*, North Holland, September 1988.

[3] M. Adam, Ph. Ingels, and M. Raynal. *Algorithmes Distribués synchrones et systèmes répartis asynchrones : concepts, mises en œuvre et expérimentations.* Rapport de Recherche RR-0862, INRIA, Centre IRISA, Rennes, July 1988. 27 p.

[4] G.V. Bochmann. Usage of protocol development tools : the results of a survey. In 7^{th} *IFIP International Workshop on Protocol Specification, Testing, and Verification, Zurich, Suisse*, North Holland, May 1987.

[5] S. Budkowski and P. Dembinski. An introduction to Estelle: a specification language for distributed systems. *Computer Networks and ISDN Systems*, 14:3–23, 1987.

[6] J.-P. Courtiat, P. Dembinski, R. Groz, and C. Jard. Estelle : un langage ISO pour les algorithmes distribués et les protocoles. *Technique et Science Informatique*, 6(2), 1987.

[7] C. Diaz, M. Vissers and J.-P. Ansart. Sedos: software environment for the design of open distributed systems. In *Proceedings of the Esprit '85 week*, North Holland, 1985.

[8] M. Heath. The hypercube: a tutorial overview. In Michael T. Heath, editor, *Hypercube Multiprocessors 1986*, pages 7–11, Oak Ridge National Laboratory, 1986.

[9] C. Jard, R. Groz, and J.F. Monin. Development of VEDA : a prototyping tool for distributed algorithms. In *IEEE Trans. on Software Engin.*, March 1988.

[10] C. Jard and J.-M. Jézéquel. Outils pour l'expérimentation d'algorithmes distribués sur machines parallèles. In *Actes du Colloque C^3 d'Angoulème*, GRECO C^3/CNRS, December 1988.

[11] C. Jard and J.-M. Jézéquel. *Un compilateur Estelle multi-processeurs pour l'expérimentation d'algorithmes distribués sur machines parallèles.* Technical Report 453, IRISA University of Rennes, January 1989.

[12] J.L. Richard and T. Claes. A generator of C-code for Estelle. In M. Diaz, JP. Ansart, JP. Courtiat, P. Azéma, and V. Chari, editors, *The Formal Description Technique Estelle, results of the ESPRIT Sedos Project.*, North Holland, 1989.

[13] J.L. Richier, C. Rodriguez, J. Sifakis, and J. Voiron. Verification in XESAR of the sliding window protocol. In 7^{th} *IFIP International Workshop on Protocol Specification, Testing, and Verification, Zurich, Suisse*, North Holland, May 1987.

[14] S. Vuong, A. Lau, and R. Chan. Semi-automatic implementation of protocols using an Estelle-C compiler. *IEEE Transactions on Software Engineering*, SE-14(3):384–393, March 1988.