

Automatic Refinement of Distributed Systems Specifications Using Program Transformations

Antônio Carlos Lima de Santana,
Antônio Francisco do Prado,
Wanderley Lopes de Souza

Departamento de Computação (DC)
Universidade Federal de São Carlos (UFSCar)
Via Washington Luiz, km 235
São Carlos - SP - CEP 04499-610 - Brazil
{santana, prado, desouza}@dc.ufscar.br

Marcelo Sant'Anna

Departamento de Informática
Pontifícia Universidade Católica do Rio de
Janeiro
Rua Marquês de São Vicente, 225 - Gávea
Rio de Janeiro - RJ - CEP 22453-900 - Brazil
santanna@inf.puc-rio.br

Abstract

Formal specification techniques and automatic refinement tools for distributed systems have become key issues in current computing technology. This paper reports the development of a refinement tool based on the Extended State Transition Language (Estelle). Estelle is a Formal Description Technique (FDT) for distributed systems and communication protocols standardized by ISO. The refinement approach addresses an OO execution metamodel which is instantiated using C++. Program transformations are used as the main enabling technology behind the construction of this tool.

Keywords: Distributed Systems, Formal Description Techniques, Estelle, Transformation Systems, Draco

1. Introduction

The increasing demand for complex distributed systems made evident the critical need for a good development process for such class of systems. The use of informal specifications can lead to errors that spread over all development phases and so, it is

desirable to work with specifications that are concise, clear and free of ambiguities. To address such a need, several Formal Description Techniques (FDTs) have been proposed. Estelle is one of these techniques, standardized by ISO and worldwide accepted [10].

Since the middle of the last decade, some tools have been developed for Estelle like syntax-oriented editors, verifiers, simulators, testers and compilers. In 1995 [1] proposed an Object-Oriented Model to transform an Estelle specification into a running implementation. Following this model, several strategies may be applied to accomplish the automatic refinement of Estelle specifications. In this work, an strategy based on program transformations is presented.

The paper is organized as follows: Section two gives an overview of FDTs with special emphasis on Estelle. The third section presents the Object Oriented Execution Metamodel for the implementation process. The fourth section presents the refinement environment. The fifth section provides information on how the generated modules are deployed as a running distributed system. The sixth section shows a case study. The seventh section finishes this paper making an overall conclusion and providing pointers to related work.

2. Specifying Distributed Systems

Nowadays Estelle and LOTOS (Language of Temporal Ordering Specification) [12] are the standard FDTs recognized by ISO, whereas SDL (Specification and Description Language) [16] is the standard FDT recognized by ITU-T (International Telecommunication Union - Telecommunication).

Estelle was developed for the formal specification of Distributed Systems and communication protocols, mainly those related to the Open Systems Interconnection (OSI) reference model [8]. Estelle is based on an extended state transition model combining two kinds of notations. States, interactions, and transitions are described by a Finite State Machine (FSM). Variables and parameters are described by the ISO Pascal programming language [9]. Further details on Estelle is given in the next subsection. An Estelle tutorial is presented in [13].

LOTOS is an international FDT able to produce implementation independent protocol specifications. LOTOS is in addition a rich design language which is able to formally support design by stepwise refinement, as well as a variety of design styles (constraint oriented, state/EFSM oriented, monolithic, etc.).

LOTOS is intended for specifying concurrent and distributed systems. It consists of a language for specifying processes (similar to CCS and CSP) and an algebraic specification language called ACT ONE.

SDL is a general purpose description language for communicating systems. Behaviour is described by communicating Extended State Machines represented as concurrent processes. Communication is represented by signals and can take place between processes or between processes and the environment of the system.

2.1 The Estelle FDT

An Estelle specification is composed of a set of *module* instances that communicate by means of *interactions*. A module may be refined into sub-modules for the definition of an hierarchical structure between the components of the specification. An example of such a structure is given in Figure 2.1 where the system and its modules are represented as rectangles. Interactions are represented as arrows. The little circles mean interaction points.

Each module is represented by a black box with input/output *Interaction Points* (IPs, for short). Unbounded FIFO queues are associated to the IPs for

the storage of incoming interactions. A queue may be *individual*, when it exclusively belongs to one IP or *common* when it is shared by more than one IP.

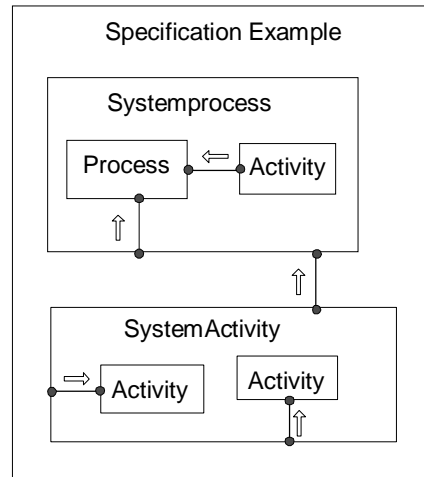


Figure 2.1 - The architecture for a typical Estelle specification

Two IPs of different modules can be connected by a full-duplex communication *channel*. A channel defines the interactions that can go out or come in through its IPs.

A module behavior is described by an ExtendedFinite State Machine (EFSM). A module is *active* if its definition includes at least one transition; otherwise it is *inactive*. Active modules must have one of these class attributes: *systemprocess*, *process*, *systemactivity*, *activity*. *Systemprocess* or *systemactivity* modules are called *subsystem* modules and cannot be enclosed within active modules.

The transitions of a module have priority over its children transition. This behavior, known as the parent/children priority principle and the class attributes define parallelism and non-determinism characteristics for a module. Non-deterministic firing of transitions happen within subsystems attributed as *systemactivity*. Both asynchronous and synchronous parallelism can be expressed in Estelle.

The internal structure of each subsystem and the bindings between IPs of their submodules may vary (i.e., they are dynamic) because actions of a module instance within a subsystem may include statements creating and destroying its children or bindings between children IPs or bindings between the module instance and its children IPs.

3. An OO Execution Metamodel for Estelle

Estelle is intended to allow developers to specify systems in a high level of abstraction. Some of these abstractions are similar to Object Orientation (OO) abstractions. The Estelle modules can be seen as black boxes that encapsulate data and behavior, resembling the objects in OO. The interactions in Estelle may be associated to the messages in OO. Many module bodies may share the same module header, suggesting the inheritance in OO. The Object Oriented Model proposed in [1] was improved in [2], becoming the starting point for this work.

Using the Object-Oriented Model mentioned above, an Estelle architectural concept (e.g., *module*, *interaction*, *interaction point*) or construction (e.g., *transition*, *channel*, *communication link*) can be implemented as a hierarchical structure of classes. A generic class (base class) captures the common structure or behavior of a Estelle concept or construction and derived classes (subclasses) capture their specializations. A component of this hierarchy can be static or dynamic: in the first case it is common

to all implementations while in the second case it is specification dependent. The static components form a class library (shelf-classes) to be referred to by the dynamic ones which must be created for each new implementation.

Figure 3.1 shows the use of the inheritance mechanism to derive the subclasses *InactiveModule*, *SystemModule*, *ProcessModule* and *ActivityModule* from the base class *Module* and to derive the subclasses *SystemProcessModule* and *SystemActivityModule* from the subclass *SystemModule*. All these classes are shelf-classes.

The object-oriented models shown in Figures 3.1 and 3.2 use the FUSION method notation [7]. In this notation the boxes represent classes, with their names on the top, attributes in the middle and methods at the bottom. The triangles denote the inheritance mechanism, which implements the generalization/specialization abstraction.

This figure also shows the main attributes and methods for the represented classes. There are other classes in this model; the complete description is presented in [2].

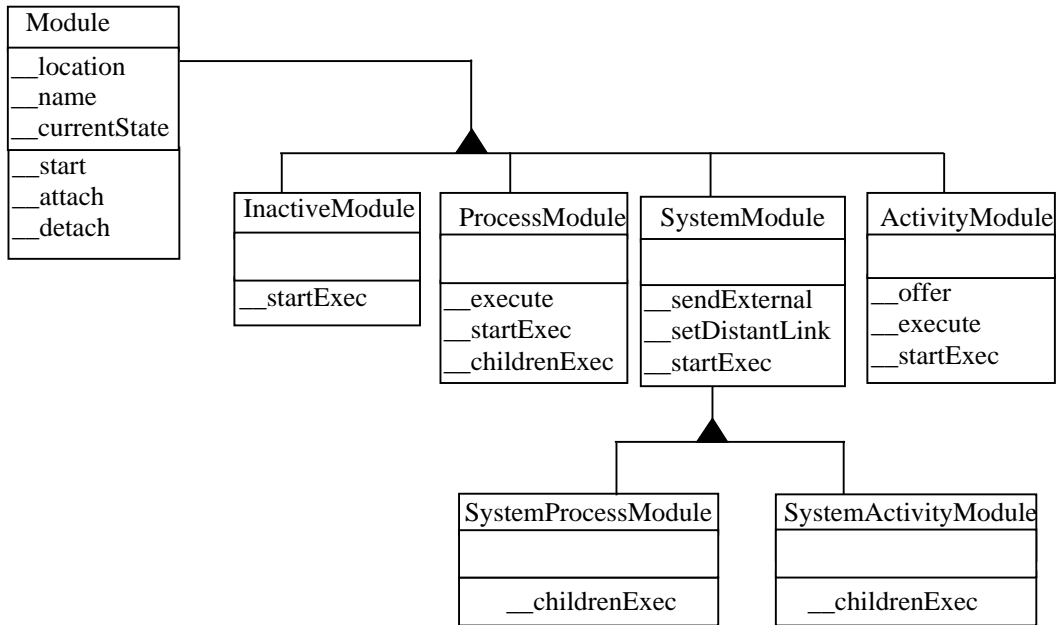


Figure 3.1 - Model for the Module architectural concept

Dynamic classes resulting from the *module* architectural concept have also been foreseen in [2]. The description of a module in a Estelle specification is composed of a header and at least one body. The header describes the module external visibility in terms of exported variables and Interaction Points. The

body describes the behavior of the module by declaring the conditions and actions of its transitions.

An object-oriented metamodel was built to generate the dynamic classes. The attributes of the metamodel classes are filled with parameters obtained from the Estelle specification. The newly generated

classes are instances of the object-oriented model for this particular Estelle specification. As shown in Figure 3.2, these instances inherit the static aspects of the shelf-classes (**ProcessModule** in this example) and

aggregate the dynamic ones. This higher level of abstraction is called a metamodel because it contains metaclasses, i.e., classes that create other classes.

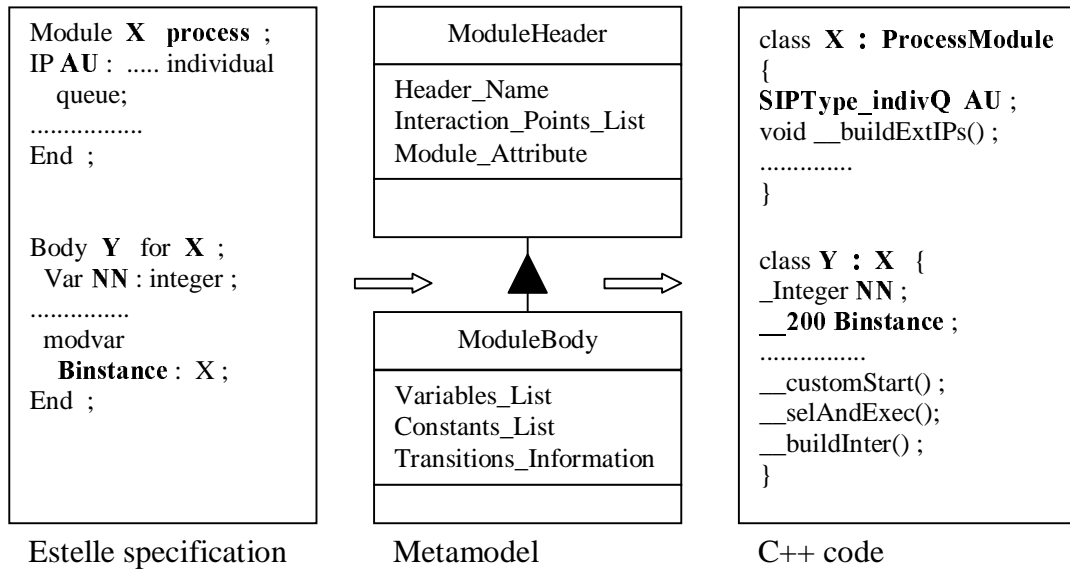


Figure 3.2 - MetaModel for the dynamic classes

According to the metamodel, a C++ class named **X** was generated for the module header **X** in the Estelle specification given as example. **Module_Attribute** in this example stores the information that the module is a **ProcessModule**. Similarly, a C++ class named **Y** was generated for its corresponding body. The Interaction Point **AU** in the Estelle specification appears as an attribute of the class **X**. **SIPType_indivQ** is a class of the object-oriented model for IPs that use individual queues.

Within the module body, **NN** and **Binstance** are turned to class attributes. The type **__200** used is another dynamically generated class. The metaclasses should also generate the methods (e.g. **__customStart**, **__selAndExec** and **__BuildInter**) of the generated classes.

4. Refining Distributed Systems Specifications Using Program Transformations

The Draco-PUC is a domain-oriented transformational system [3], [11], [14], [18] where system or software descriptions in high level specification or programming languages may be automatically transformed into executable code. In

Draco-PUC, a domain is supported by a **language**, **transformation libraries** and a **prettyprinter**.

4.1 The Refinement Environment

Figure 4.1 shows the integrated environment for Distributed Systems development using Draco domains in the lifecycle of Distributed System specified in Estelle. Starting from the user requirements, the Estelle programmer can write an Estelle specification.

Given the Estelle specification, the transformation process begins with the syntax verification by the Estelle parser in the **Parse** activity, which also generates the Estelle DAST. Then the **Transform** activity applies the transformation libraries on this DAST to improve the specification within the same domain or to transform it into another Draco domain. The output of the **Transform** activity, a C++ DAST, is oriented by the object-oriented metamodel shown in Figure 3.2.

The target domain DAST is the input for the **Unparse** activity which outputs a project composed by multiple C++ files implementing Estelle modules. The generated target project is stored in a work area and then compiled. The generated executable files are stored in a Network FileSystem (NFS).

Following that, the executable modules can be loaded and **Executed** on their corresponding CPUs. In the **Validate** activity the system behavior (**Result A** and **Result B**) is checked by the Estelle programmer against the requirements, providing feedback in order

to improve the initial specification. Based on this activity new requirements can be added and corrections can be made. The final project is obtained when there is no more corrections or requirements to be added.

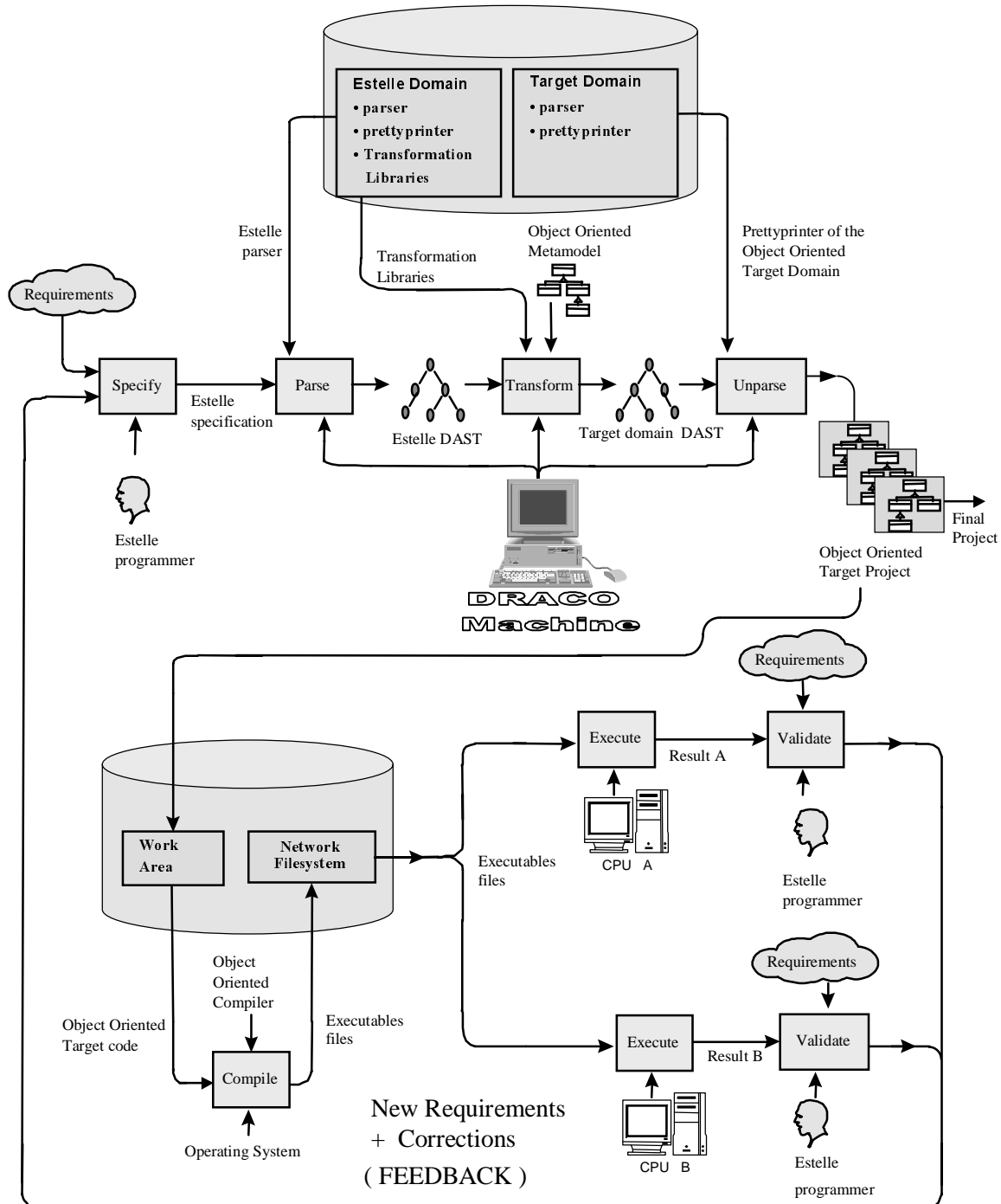


Figure 4.1 - The Integrated Environment for Distributed Systems development

The object-oriented metamodel for Estelle specifications guides the building of the transformation libraries. This guarantees that the C++ code generated in the transformation process is conforming to the object-oriented model shown in Figure 3.1. The transformation library built comprises more than 6.500 lines of code in 143 complete *Transforms* and 132 *Templates*, organized in 28 *Sets of Transforms*. Some supporting libraries were also built to hold shared memory and temporary objects, like stacks and lists.

5. Distributing Modules across the Network

The generated modules communicate with each other using a Network Interface no matter if they are in the same CPU or not. The Network Interface services use low-level communication facilities such as the Unix *sockets* in the INTERNET domain. A Communication Server program (implemented as an UNIX *daemon*) must be run on every machine where module instances may be started. So, a module can ask the *daemon* to create, load and execute the program that implements its child module using the C/C++ commands *fork* and *execv*. A protocol to synchronize the running modules is presented in [2].

For the sake of portability, the communication features used by the Communication Server and by the Network Interface were organized in a layered structure, putting the Platform Dependent Functions (low level) aside from the high level abstractions.

The Communication Server must be running in each machine to permit the interaction between distant modules. Even if the modules run on a single machine the Communication Server is needed.

The application must know the location (machine name and directory) of the executable files of the system. This information is asked at implementation time in order to build internal tables.

The Communication Server provides a convenient way of loading and running applications, with features like windows interface, step-by-step execution and variables examination. It writes the hostname and port number to the standard output. The user must redirect the output to a special file in append mode and then distribute (using ftp) this file to all the machines running parts of the system. This distribution step is not necessary if NFS (Network File System) is being used. The application reads this special file at load time to be able to communicate with the local

Communication Server. Every communication between modules is made via Communication Server.

Additionally, the modules have to know at run time where to start their children modules. In Estelle, the developer is unable to inform the module locations. One approach could be to extend the Estelle grammar to support it. In this work, the chosen solution was to allow the developer to inform these locations in each *init* statement at implementation time.

6. A Case Study

Once the domains are built, any Estelle specification can be transformed into C++ code. Some cases studies [15] were used to validate the approach and to give feedback to the development process. For the sake of illustration, this section will describe the implementation of the distributed algorithm published in [4] hereafter referred as MUTEX. This algorithm is related to a service which grants access to a shared resource (e.g. printer, tape unit) using a mutual exclusion mechanism. In this application, several users in different locations may request the same resource but only one can use it at a time. Each user is connected to a local controller and all controllers are linked to a Virtual Ring. This architecture is shown in Figure 6.1.

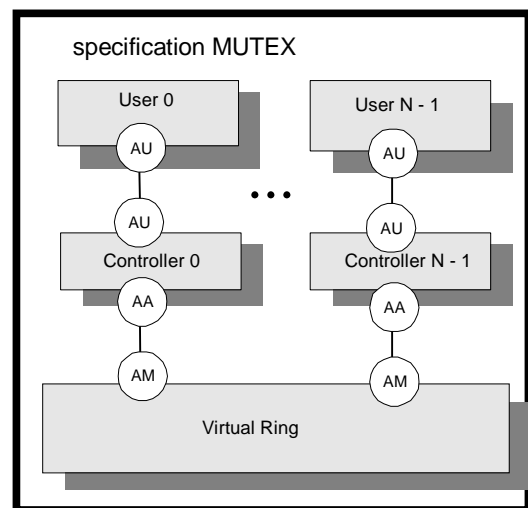


Figure 6.1 - Architecture for the MUTEX specification

The privilege is negotiated by the controllers through the exchange of informations on situations. Each controller decides whether its user can have the access depending on the situation of his left neighbor

and on its own situation. Only one controller holds the privilege at a time. After the use (or not) of the resource, the controller that holds the privilege changes its situation to allow the circulation of the privilege. A temporization mechanism prevents the resource waste when the privilege is available but the user does not use it.

In this implementation, the shared resource was an output file, simulating a physical printer. This output file was stored in a Network File System to guarantee its uniqueness.

An overview of the specification for the Distributed Algorithm is given in Figure 6.2, showing

the channels, modules, interactions and interaction points. In this figure, the constant named **Numb_Stations** (1) means the number of stations that may request the service. The constant named **Time_Privilege** (2) means the time a controller holds the privilege. The Virtual Ring module of Figure 6.1 was specified by a header (**TYPE_RING**) and a body (**BODY_RING**) which are shaded in Figure 6.2. All the IPs in this module act as Providers in the **Access_Ring** channel. This means that they can emit the **S_Req** and **S_Resp** interactions and that they can receive the interactions **S_Conf** and **S_Ind**.

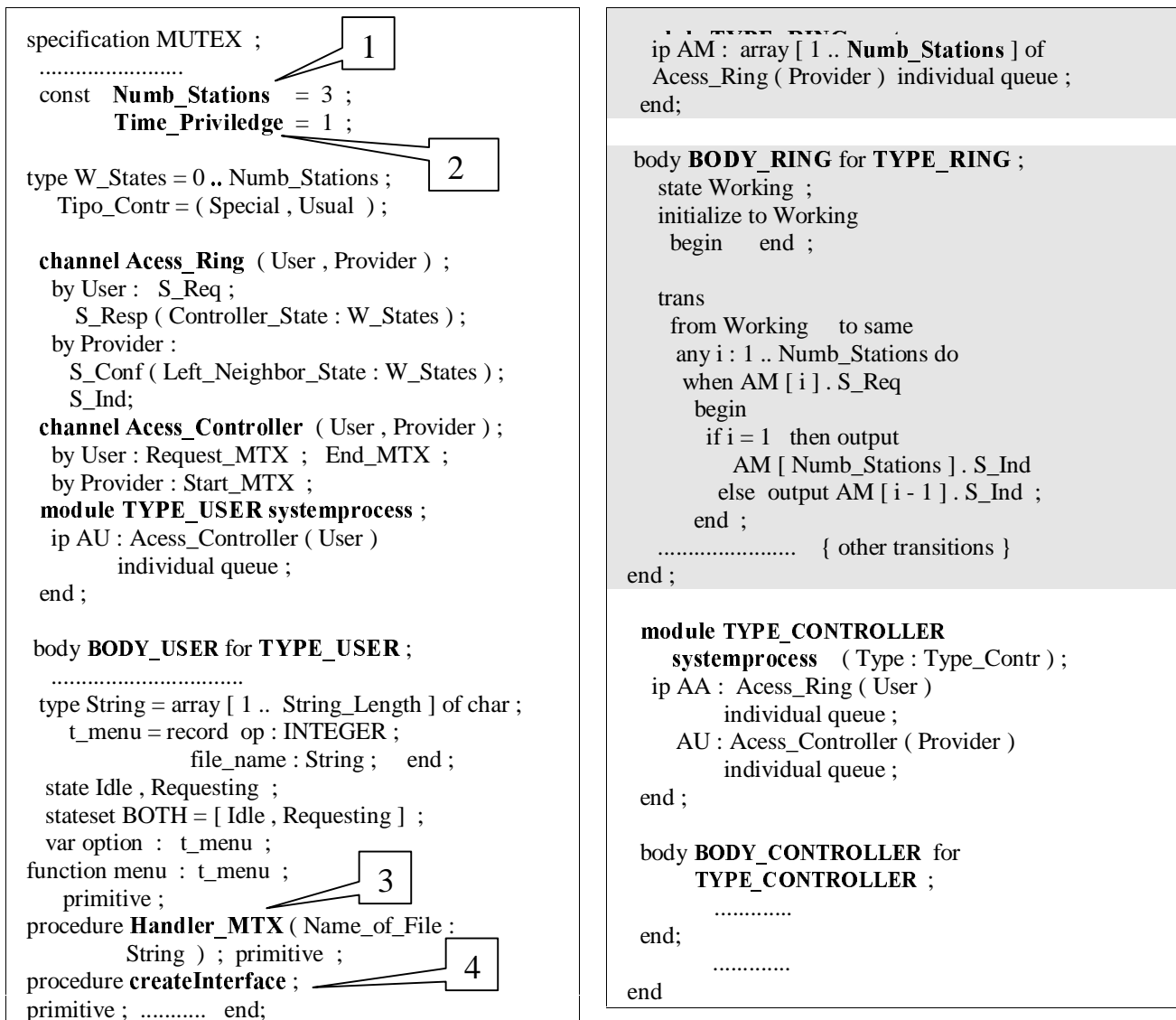


Figure 6.2 - Overview of the Estelle specification for the MUTEX

The MUTEX specification contains Estelle procedures and functions, the *record* data type, the Estelle *any* clause, arrays and many modules. Its implementation lead to 2,200 lines of C++ code, not taking the shelf classes library and the Communication libraries into consideration.

Figure 6.3 presents the mapping between the MUTEX modules and the classes generated for them.

All the modules in the MUTEX specification are of type *systemprocess*, thus requiring the creation of classes derived from the *SystemProcessModule* shelf-class.

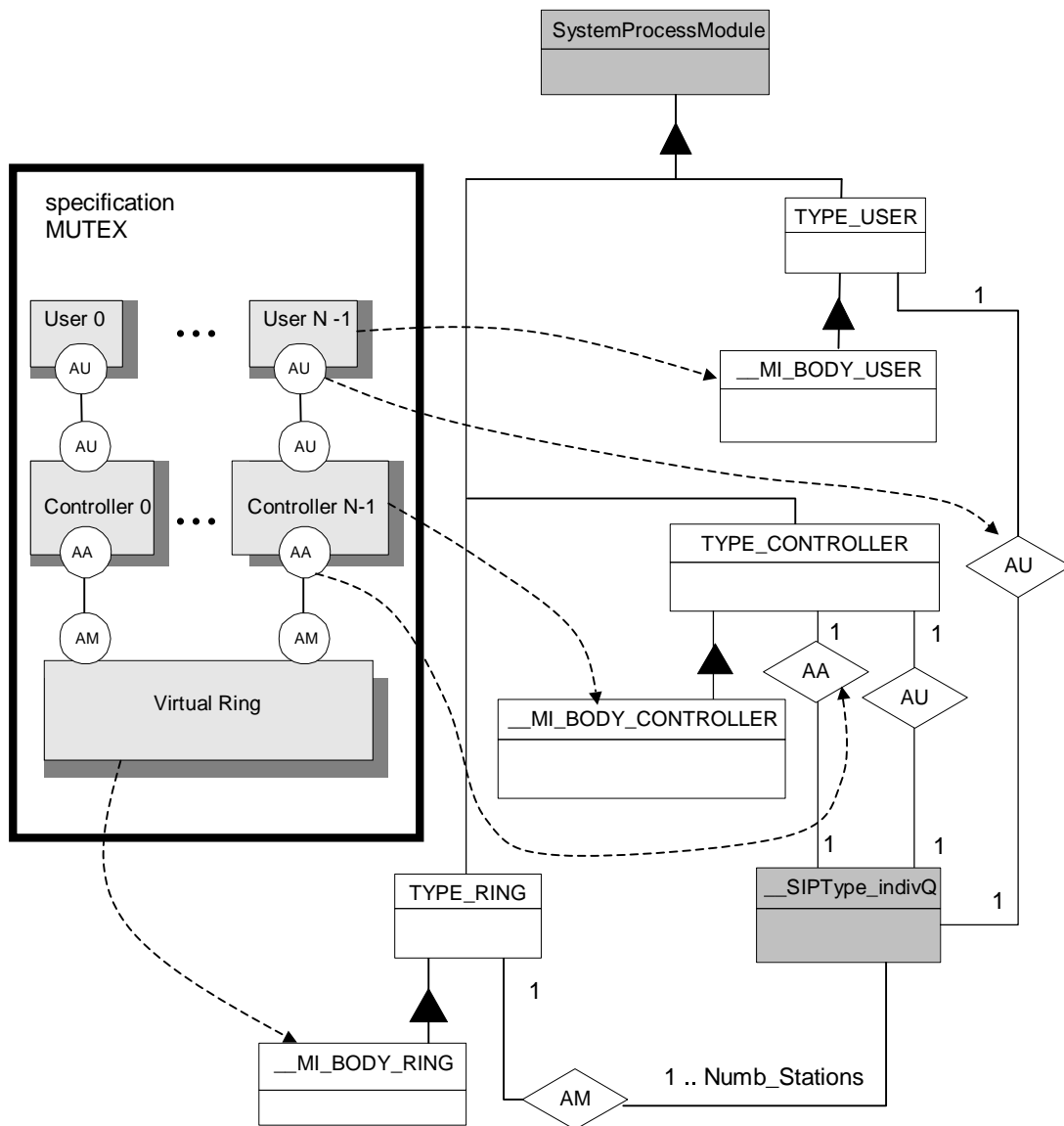


Figure 6.3 - MUTEX classes derived from SystemProcessModule

In Figure 6.3, the classes prefixed with TYPE represent the module headers while the classes prefixed with __MI represent bodies for these module headers. The shelf-classes shown earlier in Figure 3.1 are shaded. The module header

TYPE_CONTROLLER presents two interaction points (IPs), one named AA and other named AU, both associated to individual queues. The module header TYPE_RING may have several interaction points. The

exact number is dictated by the constant `Numb_Stations`.

It should be noted that instances of classes for the module bodies do have interaction points by means of inheritance.

`__MI_BODY_CONTROLLER`, for instance, has one occurrence of the IP named AA and one occurrence of the IP named AU.

The C++ implementation was driven by the Metamodel depicted in Figure 3.2, and it preserved the semantics of the composition of an Estelle specification, that is, a set of modules that communicate by interaction points. So, the not shaded object-oriented classes shown in Figure 6.5 are instances of the classes generated by the metaclasses shown in 3.2 for the MUTEX implementation.

Two Pentium machines running Linux were used to test the implementation. One of these machines has 32 Mbytes of RAM and the other has 16 Mbytes. The physical connection was performed by means of Ethernet adapters, UTP (Unshielded Twisted Pair) cables and a hub. The two machines make part of a lab network comprised of Windows NT, Windows 95 and Netware environments, sharing TCP/IP and IPX/SPX protocols. Regarding to the MUTEX specification, the success of the experience was confirmed by the fact that the input files were correctly appended to the output file without mixing.

7. Conclusions and results

The Computing Department at São Carlos University intended to produce a tool that automatically generated implementations from Estelle specifications. To accomplish this goal an object-oriented model for Estelle specifications was proposed, suggesting its implementation as a future work. There is no restriction on the target computer language to be used.

This work is the first draft of a Master thesis on Software Engineering applied to the development of distributed systems and it aims to produce the desired tool, according to the given model. C++ was chosen as the target language because of its conformance to the object-oriented paradigm but other object-oriented languages could be used as well. The major contributions are:

- The validation of the proposed OO execution model and the adoption of the Draco approach to transform Estelle specifications into C++ code. This

approach allows the porting of Estelle specifications to other platforms with a high degree of reuse.

- The construction of the Estelle domain composed of parser, prettyprinter and transformation libraries.

- The extension of the recommendations given in [2]. Despite the valuable help of the OO model for Estelle specifications, a great effort was made to solve practical programming issues. Several constructs in Estelle do not have similar in C++ and thus had to be simulated. Procedure nesting is one of these constructs, where a procedure may handle two variables with the same name but different scopes. Even so, all the recommendations of the OO Model were respected.

- Both the code generation and the execution may now take place in low cost machines, available in many laboratories.

Since the Estelle semantics is embedded within the Draco transformations it is possible to state that the generated implementation conforms to its specification. The experiments performed have confirmed such a statement.

Draco runs in a variety of platforms; the generated code was compiled with the GNU C++ compiler and ran on Pentium machines under Linux operating system. Centralized and distributed applications were successfully tested and presented the same behavior as the observed in Sparc/Sun implementations with no perceptible loss of performance. As long as the platform dependent functions be ported, the implementations will run on other distributed platforms. Nowadays, all the communication based on *sockets* is available on Windows 95 and NT with few modifications comparing to Linux.

A survey on Estelle tools is presented in [2] pointing PETDINGO [17] as the tool with the highest degree of automation. Practical experiments with PETDINGO are reported in [5]. PETDINGO is composed of two products: PET, a front-end component that generates an intermediate representation of the objects found in the specification and DINGO, that recovers these objects and generates the C++ code. With the Draco approach, the transformation is carried out in only one passage.

PETDINGO has no interactivity: undefined words have to be informed by editing files before the final compilation. Another advantage of Draco over PETDINGO is that PETDINGO generates only C++ code whereas Draco may generate code in other languages. The target language in Draco is also grammar-oriented while PETDINGO has only the

source grammar, i.e. Estelle. Both the Draco machine and PETDINGO use Bison as syntax analyser, but the former has a backtracking mechanism to solve grammar conflicts, as stated in [6].

9. Acknowledgements

This work was partially supported by CNPq, the Brazilian financial agency for scientific projects. The author Antônio Santana would like to thank the help of Fundação Paulista de Tecnologia e Educação, in special Faculdade de Informática de Lins, where he teaches.

10. References

- [1] Barbosa, C.B. & Lopes de Souza, W. *Modelagem Orientada ao Objeto de Especificações Escritas em Estelle*. Proceedings of the 13th Brazilian Symposium on Computer Networks, Belo Horizonte (MG), 22-26/05/1995, pp. 23-39.
- [2] Barbosa, C.B. *Modelagem orientada a objetos de especificações Estelle*. Master's thesis, Postgraduate Program in Computer Science (PPG-CC), Federal University of São Carlos (UFSCar), 1995.
- [3] Leite, J.C.S. & Sant'Anna, M. & Freitas, F. *Draco-PUC: A Technology Assembly For Domain Oriented Software Development*, Proceedings of ICSR94 (International Conference on Software Reuse), IEEE Press, 1994.
- [4] Farias, C.R.G. & Lopes de Souza, W. *Especificação Formal e Validação de um Sistema de Acesso por Exclusão Mútua*. Technical Report RT-DC 004/95, Department of Computer Science (DC), Federal University of São Carlos (UFSCar), 1996.
- [5] Farias, C.R.G. & Lopes de Souza, W. & Barbosa, C.B. *Design and Implementation of Distributed Databases Using Estelle*. Proceedings of the 11th Brazilian Symposium on Database, São Carlos(SP), 14-16/10/1996, pp.158-171.
- [6] Freitas, F.G & Leite, J.C.S. & Sant'Anna, M. *Aspectos Implementacionais de um Gerador de Analisadores Sintáticos para o Suporte a Sistemas Transformacionais*. 1st Brazilian Symposium on Programming Languages, B.Horizonte(MG), Set/1996.
- [7] Coleman, D. et al. *Object-Oriented Development - The Fusion Method*. Prentice Hall, 1994.
- [8] ISO IS 7489. *Information Processing Systems - Basic Reference Model of Open Systems Interconnection*. 1983.
- [9] ISO IS 7185. *Pascal Programming Language*. 1983.
- [10] ISO IS 9074. *Information Processing Systems - Open System Interconnection - Estelle - A Formal Description Technique based on an Extended State Transition Model*. 1988.
- [11] Sant'Anna, M. & Leite, J.C.S. & Prado, A.F., *Draco-PUC: A Workbench for Developing Transformation-Based Software Generators*, Proceedings of ICSE98, ACM Press, 1998.
- [12] ISO IS 8807 *Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique based on the temporal ordering of observational behaviour*. 1989.
- [13] Lopes de Souza, W. *Estelle: uma técnica para a descrição formal de serviços e protocolos de comunicação*. Journal of the Brazilian Computer Society, Vol.5 N.1, Jul/Sep 1989, pp. 33-44.
- [14] Neighbors, J. *Software Construction Using Components*. PhD thesis, University of California at Irvine (EUA), 1980.
- [15] Santana, A.C.L. & Prado, A. F. & Lopes de Souza, W. *Utilização do Paradigma Draco para implementar especificações Estelle na linguagem C++* Proceedings of the 15th Brazilian Symposium on Computer Networks, São Carlos(SP), 19-22/05/1997, pp.118-134.
- [16] ITU-T *Recommendation Z.100*, 1993.
- [17] Sijelmassi, R. & Strausser, B. *The Distributed Implementation Generator: an overview and user guide*. Technical Report NCSL/SNA, Gaithersburg (EUA), 1991.
- [18] Sant'Anna, M. & Leite, J.C.S, Prado, A.F., *A Generative Approach to Componentware*. Proceedings of CBSE98 (Workshop on Component-Based Software Engineering), 1998.