

Combined approaches, which start with high level knowledge ("real world" knowledge, domain knowledge, etc.), produce a model of this information and try to find model concepts instances in the system's implementation.

Bottom-up approaches are based on analysis and abstraction techniques of source code [1, 9, 10, 14]. These techniques, such as program slicing, run-time trace analysis, data-flows and control-flows analysis are closely related to human intervention. Automatic recovery of high level abstractions is difficult [3]. For example, automatic recognition of algorithms is complex due to the wide variety of coding techniques and the huge amount of code in which the algorithm may be embedded. In this case, human intervention is necessary to map code fragments to real world and domain concepts.

A combined approach is a two phase process. First, high level knowledge is analyzed and a model is produced. Second, program sources are explored to attempt to find instances of the previous model concepts. Code exploration is then driven by high level concepts recognition. Current research works [3, 5, 6, 12] focus on a particular application domain. A model of the domain is produced and model concepts are searched and located in the source code.

This paper presents a combined approach which makes use of the real world knowledge. This approach avoids arduous and costly domain models construction and can be applied to large legacy systems. It is supported by an information systems meta model which represents the main manipulated concepts.

The following section discusses the dilemma "real world" vs. "domain" model construction. The next two sections outline the meta model and the architectural recovery process. Finally, the main contributions and the perspectives are reported.

II. "Real world" vs. "domain" model construction :

Bottom-up approaches are mainly using source code and rarely make use of the system experts', the domain experts' and the users' knowledge. This information, which exist in human mind, is used informally; generally in the form of interactions with a reverse engineering tool. The user employs his knowledge to guide the reverse engineering process and construct a system's architecture.

Most combined approaches make use of human knowledge by building domain models. The target domain, to which the investigated system pertains, is studied and a model is produced. However, domain models building is difficult and costly [3]. It requires many system experts interviews and calls for "knowledge engineers" participation to organize raw knowledge into concepts and rules. The difficulty is also due to the models genericity and their ability to represent the application domain independently of a given system.

Moreover, a given system may belong to many application domains [3]. This yields that architectural recovery by a combined approach necessitates many domain models construction. For this reason, feasibility of combined approaches was demonstrated on small systems. Jean-Marc DeBaud [6] constructed a "Report Writing" domain model and showed that his method permits architectural recovery and evolution of Report Writing applications. Clements et al. [3] have developed a knowledge-based software assistant prototype called "Gadfly" to support development and comprehension of command, control and communication systems (C3) from a security perspective. They constructed C3 and information security domains, which were used in the reverse/forward engineering task.

Our approach, which aims at architectural recovery of large legacy systems, proposes a rigorous human knowledge exploitation and doesn't advocate domains modeling. This knowledge is unreliable but is in turn of high level of abstraction. In fact, a person may remember a system's function (even obsolete) but doesn't remember easily an implementation detail. This knowledge may correspond to information that appears anywhere or hasn't been updated in the system's documentation. The method

we are presenting consists in knowledge collection and real world concepts identification to make use of this knowledge in the architectural recovery.

The method is supported by an information systems meta model that represents the major manipulated concepts. The meta model is layered into two levels : a real world level and an architectural level. The following section describes the meta model and its concepts.

III. Meta-model presentation

The proposed meta model is described at figure 1 in an object oriented notation. It uses the concepts of relationship, inheritance and aggregation. The real world level summarizes the global system's context. The architectural level describes the main concepts necessary to architectural description.

III.1. Real world level

Real world level corresponds to the information system's view of the real world. In particular, it identifies the organization structure (internal and external **organizational units**²), the organization's **goals** and the **activities** that contribute to their achievement. An organizational unit references an activity when it uses its results. An activity is the responsibility of an internal organizational unit. Activities may be manual or automated. They use **resources** (forms, statements, ...) and generate **products** (processed or annotated documents, printed reports, ...). An activity has **constraints** : a **pre-condition** to trigger the activity and a **post-condition** to control its good execution.

III.2. Architectural level

Jerding [9] defines a Software Architecture as "*a high level program model that describes a system's major pieces (its Components) and how they interact (its Connectors)*". Intuitively, components correspond to boxes, and connectors accord to lines in box-and-line descriptions of software architectures. It is now admitted that architectural descriptions are based on components and connectors [3, 13, 14]. Architectural Description Languages (ADLs) are based, among other concepts, on the description of components and connectors [8].

The meta model reflects this comprehension by including the system's constituents : "**components**" and their interaction modes "**connectors**". It has been indicated that a component may be a **program sequence** or a **data store**. A connector may be uni/bi-directional and concerns a data or a control flow. An **automated activity** is implemented by ordered sequences of program chunks (executable procedures, batch files, etc.). A **product** is implemented by a component.

IV. The architectural recovery method

The method is composed of three main steps. The first step consists in gathering the real world knowledge of the information system. The second, uses the source code to automatically generate a preliminary system's architecture. The generated architecture corresponds to a macroscopic view of the system. The final third step scatters the initial architecture and makes links between the real world knowledge and the initial architecture components. This step allows verification of the information obtained during the first step and dictates the architecture granularity.

² Internal organisational units : departments, services, etc. External organisational units : customers, suppliers, etc.

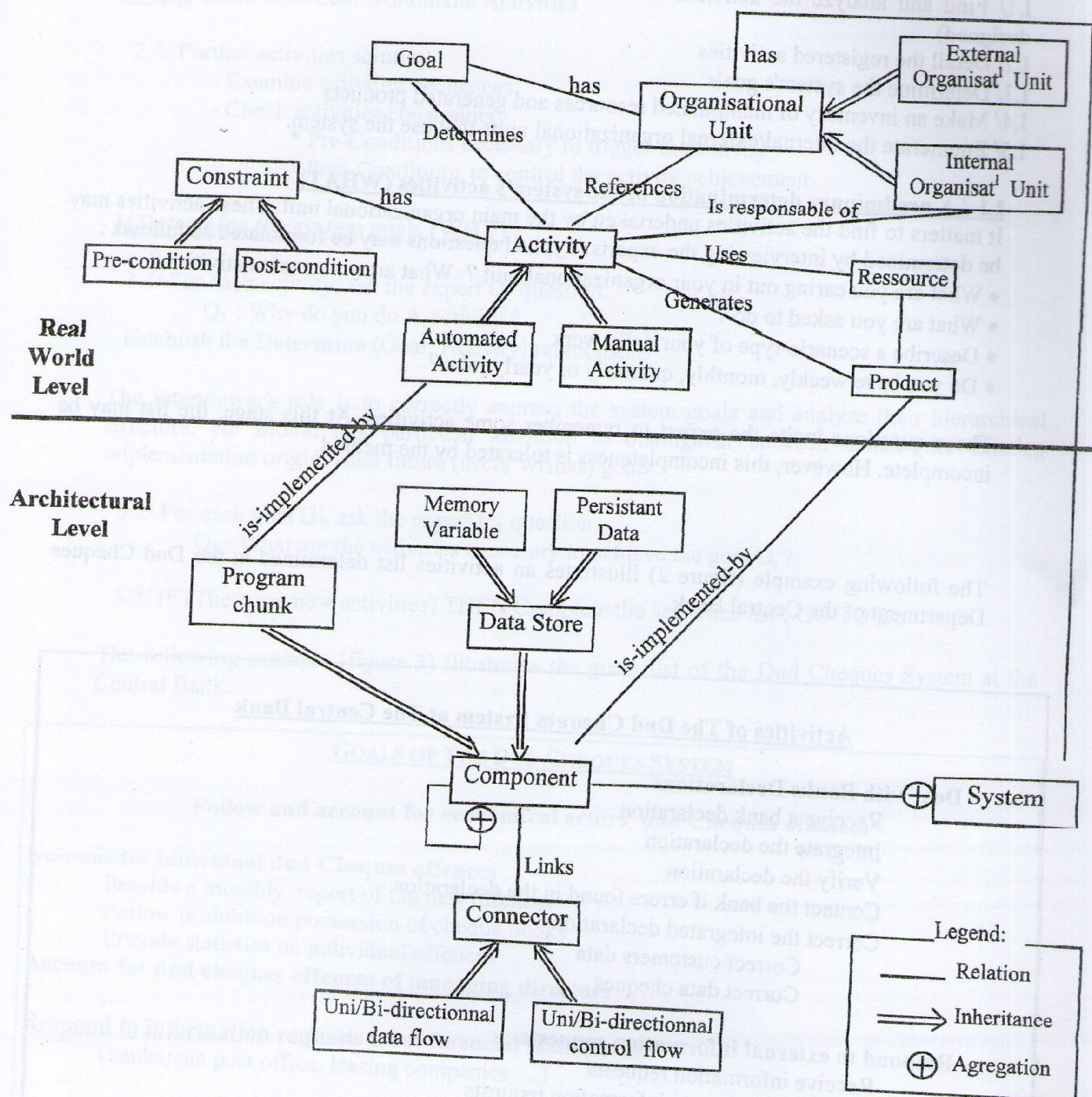


Figure 1 : The proposed Meta-Model

IV.1. First step : Real world model construction

This step is driven by interviews with the system experts, the users and the domain experts. A system expert corresponds to a person who has participated in the system design, development or maintenance. A user is a person that frequently makes use of the system to carry out his job. A domain expert refers to a person that masters the system's business rules. For paper's need, we refer to one of the persons above by the "expert" qualifier.

This first step is composed of five sub-steps as follows :

- I.1/ Find and analyze the activities of the main organizational unit (for whose the system is designed)
- I.2/ Detail the registered activities
- I.3/ Determine the system's goals
- I.4/ Make an inventory of manipulated resources and generated products
- I.5/ Enumerate the internal/external organizational units that use the system.

I.1-/ A preliminary determination of the system's activities (WHAT)

It matters to find the activities undertaken by the main organizational unit. These activities may be determined by interviewing the experts. Typical questions may be formulated as follows :

- What are you caring out in your organizational unit ? What are your main activities ?
- What are you asked to do ?
- Describe a scenario-type of your daily work
- Do you have weekly, monthly, quarterly or yearly jobs ?

These questions incite the expert to remember some activities. At this stage, the list may be incomplete. However, this incompleteness is tolerated by the method.

The following example (figure 2) illustrates an activities list determined at the Dud Cheques Department of the Central Bank.

Activities of The Dud Cheques System at The Central Bank

Deal with Banks Declarations

- Receive a bank declaration
- Integrate the declaration
- Verify the declaration
- Contact the bank if errors found in the declaration
- Correct the integrated declaration
 - Correct customers data
 - Correct data cheques

Respond to external information requests

- Receive information requests
- Prepare responses to information requests
- Prepare the mail

Figure 2 : Activities of the Dud Cheques System at the Central Bank

2-/ Detail the determined activities (HOW) :

REPEAT

{for each activity}

- 2.1/ Make an abstract³ description of the activity

³ The activity description should not reflect implementation details such as file, database, report or program/module names. Example: "Receive the floppy disk containing file FDECLAR.DAT" should be replaced by "Receive declarations"

2.2/ Determine the activity nature : Manual, Automatic, Semi-Automatic⁴

2.3/ IF (Activity IS Semi-Automatic) THEN Decompose the activity

UNTIL There is no Semi-Automatic Activities

2.4/ Further activities scrutiny

- Examine activities sequences
- Check activities constraints
 - * Pre-Conditions necessary to trigger the activity
 - * Post-Conditions to control the activity achievement

3/ Determine the system goals (WHY) :

3.1/ For each activity, ask the expert Q₁ question.

Q₁ : Why do you do A_i activity ?

Establish the **Determine** (Goal, Activity) relationship

The interviewer's role is to correctly express the system goals and analyze their hierarchical structure. He should pay particular attention to distinguish between actual (the existing implementation origins) and future (users' wishes) goals.

3.2/ For each goal G_i, ask the expert Q₂ question.

Q₂ : What are the activities necessary to achieve the goal G_i ?

3.3/ IF (There are new activities) THEN Complete the activities list ; GO TO 2/.

The following example (figure 3) illustrates the goals list of the Dud Cheques System at the Central Bank.

GOALS OF THE DUD CHEQUES SYSTEM

Follow and account for economical actors' dud Cheques offences

Account for individual dud Cheques offences

- Provide a monthly report of the new offences
- Follow prohibition possession of cheque books
- Provide statistics on individual offences

Account for dud cheques offences of managing directors

....

Respond to information requests from financial institutions
(banks, the post office, leasing companies, ...)

Support new cheque books agreement

Figure 3 : Goals of the Dud Cheques System at the Central Bank

4/ Initial system manipulated resources determination

This step consists in gathering (in bulk) the resources manipulated by the main organizational unit. Each resource is linked to the activity that uses or generates it. At this point, some activities may be detected as they were previously forgotten.

⁴ A semi-automatic activity is composed of one or many manual activities and one or many automated activities

- 4.1/ Determine the manipulated resource list
- 4.2/ For each resource
 - 4.2.1/ find the activities that uses it (**Uses**(Activity, Resource) relationship)
 - 4.2.2/ find the activity that generates it (**Generate**(Activity, Resource) relationship)
- 4.3/ IF (There are new activities) THEN Complete the activities list ; GO TO 2/.

5/ System organizational units determination

Responses to the following questions allow the determination of organizational units that have a relationship with the prospected system. An example is presented at figure 4.

- 5.1/ For each activity, ask the expert Q_1 question.
 Q_1 : Who is driving/carrying A_i activity ?
- 5.2/ For each "product" resource, ask the expert Q_2 question.
 Q_2 : Who needs P_i product ?
- 5.3/ For each organizational unit, identify its nature : Internal or external.

Q_1 question establishes the **is-responsible-of** (organizational unit, activity) relationship
 Q_2 question establishes the **references** (organizational unit, activity) relationship

The knowledge collected during this step will be validated later by confrontation to source code and mapping links to the architectural components.

ORGANIZATIONAL UNITS REQUESTING THE DUD CHEQUES SYSTEM OF THE CENTRAL BANK	
Financial institutions (banks, the post office, leasing companies, ...)	External
Justice ministry (Law courts, ...)	External
The treasury	External
Dud cheques department	Internal
Central information department	Internal
Economical information department	Internal
Bank Inspections department	Internal
Banks department	Internal
Refinancing department	Internal

Figure 4 : Organizational Units requesting the Dud Cheques System of the Central Bank

IV.2. Second step : Preliminary architecture of the system

This step aims at generating automatically, an initial architecture of the system (components, connectors). Automatic decomposition is done according to the following rules :

Initial components are :

- Program sources : Separate program files

• **Data files** : Their list is determined by an automatic examination of program sources. Generally, this analysis concerns declarative sections. For example, in COBOL programs, examination concerns the INPUT-OUTPUT SECTION. For PASCAL language, the analysis deals with TYPE and VAR declarations in all programs and sub-programs. However, in some cases, analysis of all the source code is needed, especially with languages that do not require preliminary declarations or when external calls (intrinsic) are used.

Initial connectors are :

Connector	Linked Components
program call	Program-Program
Declaration	Program-File
Manipulate	Program-File

At the end of this step, the component list has a low level granularity. The final third step allow component refinement.

IV.3. Third step : Architecture refinement

This step constructs the matrix of the figure 5. Lines contain components that correspond to program sequences and columns list the automated activities. A checked cell means that the component C_i is needed to achieve the A_i activity.

Is implemented by Implement	Activity 1	Activity 2	Activity i	Activity n
Component 1	X	X		
Component 2		X		X
Component i				
Component n	X			

Figure 5 : Components/activities matrix

Matrix analysis allows determination of :

- Components that implement many real world activities
- Components that do not correspond to any real world activity
- Real world activities implemented by many components
- Real world activities that are not implemented by any component

The refinement stage consists in :

- Decomposing components that implement many activities
- Examining components that do not correspond to any of the real world activities
- Check if they do not implement a non listed activity(ies). If so, update the activities list
- Check if they do not correspond to obsolete functions or dead code. If so, eliminate the component.
- Inspecting real world activities that are implemented by zero components
- Check the activity nature (Manual, Automatic)

The refinement process is stopped when :

- A real world activity is implemented by one or many components
- A component does implement only one real world activity
- There doesn't exist any component corresponding to zero activities
- There doesn't exist any activity implemented by zero components

V. The method contributions

The main contribution of this architectural reconstruction is to allow reasoning about the system at a high level of abstraction. The resulting architecture permits change impact analysis and can be used to better apprehend the system's evolution.

The relationships ("determines", "generates", etc.) between real world concepts, and mapping links ("is-implemented-by" relationships) between real world concepts and system's components are the key features that permit to respond to questions such as :

- Which activities contribute to goal G_i satisfaction ?
- If goal G_i evolve to goal G_j , what should be updated in the system's implementation ? What are the involved automated activities ?
- If the content and/or the quality of the product P_i change, what are the required code source modifications ?

In these cases, mapping links provide the components and connectors lists associated with the implicated activities.

Moreover, it is possible to reason about the manual activities and plan their automation. Besides, the recovered architecture may be used to better apprehend the implementation of Business Process Reengineering (BPR). BPR solutions [4] influence considerably the activities natures and sequences, and affects the content and quality of the resources/products. This yields that some activities, resources or products may be created, deleted or redesigned. Relationships and mapping links, resulting from the application of our method, can support this reorganization effort. Architectural evolution is then done according to organisational needs.

VI. Conclusion :

In this paper, we have presented a reverse engineering method that enables architectural recovery of large legacy systems. This method makes a rigorous use of human knowledge by real world model construction and avoid the arduous and costly domain model building. The method is based on an information systems meta model, layered into two levels and which summarizes the main manipulated concepts. The method and its contributions were presented.

A first experimentation of our method was driven manually at the Central Bank of Tunisia, and had proven the method usefulness. The results of the experimentation were as follows :

- An architectural description of the prospected system
- a list of old programs which were no more used
- many versioning problems identification
- and many dead code elimination

We plan to develop a tool that proceeds to source code analysis, automatic generation of the initial architecture, and graphical visualisation. We are investigating techniques to aid in program slicing, to be used during the third step, when a component do implement many real world activities.

References :

- [1] G. Abowd, A. Goel, D. F. Jerding, M. M. McCracken, M. Moore, J. William Murdock, C. Potts, S. Rugaber, L. Wills, "MORALE Mission Oriented Architectural Legacy Evolution", Proceedings of the International Conference on Software Maintenance'97, Bari, Italy, September 29-October 3, 1997.

- [2] John K. Bergey, Linda M. Noethrop, Dennis B. Smith, "Entreprise Framework for the Disciplined Evolution of Legacy Systems", Technical Report (CMU/SEI-97-TR-007), Pittsburg, PA: Software Engineering Institute, Carnegie Mellon University, October 1997.
- [3] Paul Clements, Robert Krut, Ed Morris, Kurt Wallnau, "The Gadfly : An Approach to Architectural-Level System Comprehension", Proceeding of IEEE 4th International Conference on Program Comprehension, Berlin 1996.
- [4] T. H. Davenport, J. E. Short, "The New Industrial Engineering: Information Technology and Business Process Redesign", Sloan Management Review, Summer 1990, pp. 11-27.
- [5] Jean-Marc DeBaud and Spencer Rugaber, "A Software Re-Engineering Method using Domain Models", Proceedings of the International Conference on Software Maintenance, Nice, France, October 1995, pp. 204-213.
- [6] Jean-Marc DeBaud, "Lessons from a Domain-Based Reengineering Effort", Proceedings of the Working Conference on Reverse Engineering, WCRE'96, IEEE, Monterey, California, November 1996.
- [7] M. J. Freeman, P. J. Layzell, "A Meta-Model of Information Systems to Support Reverse Engineering", Information and Software Technology, Vol. 36, N°5, May 1994, pp. 283-294.
- [8] D. Garlan, R. T. Monroe, D. Wile, "ACME: An Architecture Description Interchange Language", Proceedings of CASCON'97, November 1997.
- [9] Dean Jerding and Spencer Rugaber, "Using Visualization for Architectural Localization and Extraction", Proceedings of the fourth Working Conference on Reverse Engineering, IEEE Computer Society, Amsterdam, the Netherlands, October 6-8, 1997, pp. 56-65.
- [10] Rick Kazman, Jeromy S. Carrière, "Playing Detective: Reconstructing Software Architecture from Available Evidence", Technical Report CMU/SEI-97-TR-010, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, October 1997.
- [11] Rick Kazman, Mark Klein, Mario Barbacci, Tom Longstaff, Howard Lipson, and Jeromy Carrière, "The Architecture Tradeoff Analysis Method", Technical Report (CMU/SEI-98-TR-008, ESC-TR-98-008), Pittsburg, PA: Software Engineering Institute, Carnegie Mellon University, July 1998.
- [12] Melody M. Moore, Spencer Rugaber, "Domain Analysis for Transformational Reuse", Proceedings of the fourth Working Conference on Reverse Engineering, WCRE'97, IEEE, 1997.
- [13] Mary Shaw and David Garlan, "Software Architecture/Perspectives on an Emerging Discipline", Prentice Hall, 1996.
- [14] P. Tonella, R. Fiutem and G. Antoniol, "Augmenting Pattern-Based Architectural Recovery with Flow Analysis : Mosaic - A Case Study", Proceedings of the Working Conference on Reverse Engineering, WCRE'96, IEEE, 1996.

Using LOTOS for High Level Specification of Large Scale System

Noura Alloui
Université de Batna
Institut d'Informatique
Batna 05000 Algérie

ABSTRACT: The last decade has known considerable progress in the application of formal methods for hardware design. Formal methods are based on the use of formal specification language to describe a hardware.

In this paper, LOTOS (Language of Temporal Ordering Specification) is used for the formal specification of a large scale system. As application, we specify the RRM (Rewrite Rule Machine) at high level of description.

KEY-WORDS: Hardware design, formal specification, formal specification language, LOTOS.

1. Introduction

Digital circuit design is an area where debugging hardware is difficult and hardware errors are very costly to fix. To achieve error-free designs, hardware designers provide first a description of circuit then proceed to correct its functionality behaviour before its production. Two approaches, simulation and formal methods of specification, have been used in the hardware design. Simulation has been extensively used in the design of several digital circuits. However, with the VLSI technologic evolution, the hardware systems become more complex. The verification of descriptions of complex systems using the simulation becomes impractical. Formal methods are based on the use of formal specification language to describe a hardware. A language is formal if it has precise and an unambiguous semantics. The advantage of this approach is to provide correctness of the specification even in the case of complex systems.

The last decade has known considerable progress in the application of formal methods for hardware design. According to the language used to describe hardware system, these methods are classified in three classes: approach using VHDL language [1], approach based on functional programming [11] and approach using LOTOS language.

LOTOS is recently applied to specifying digital circuits [6, 13, 14]. The use of LOTOS in design hardware system have gained more attention. Its investigation in the hardware domain is a novel approach that might offer new insights or benefits compared to other approaches of hardware specification [14]. The advantages of this approach are [6]:

- LOTOS has a formally defined syntax, static semantics, and dynamic semantics which are described operationally in terms of inference rules.
 - it is based on a set of sound mathematical theories which allow designers to prove a rich set of algebraic equivalence properties, based on several types of equivalence relations between a component which drives many other components, a very common characteristic of hardware systems.
- In spite of LOTOS's benefits, few of works have applied it in hardware design. The only work known is the specification of logic circuits at logic level made by Sinott [13].

In this paper, we use LOTOS to specify large-scale hardware at level of transfert register. As an application, we specify a massively parallel machine namely RRM (Rewrite Rule Machine). The main interest of this specification is to provide a correctness criterion for the RRM architectural design.

The most basic level of RRM is a processing element called *cell*. Four cells, which share local communication buses, make up a *tile*. In section 2 we present a general architecture of RRM followed by tile and cell descriptions. The specification of the cell and the tile in LOTOS are presented in section 3. In section 4, we present related work and in section 5 we give our evaluation and indicate future work.

2. RRM description

To describe how the RRM machine executes programs on data, we first suppose that we have a term to rewrite with applying a collection of rewrite rules. An RRM computation starts with this term as data and the collection of rewrite rules as a program. The rules are applied until the term is reduced, in the sense that no rule is applicable. The resulting term is considered as the final result of the computation. The RRM is MIMD/SIMD machine. In SIMD (Single Instruction Multiple Data) mode, single rule is applied at different sub-terms of the same rewriting term. In MIMD (Multiple Instruction Multiple Data) mode, many different rules can be applied simultaneously at many different sub-terms of rewriting term.

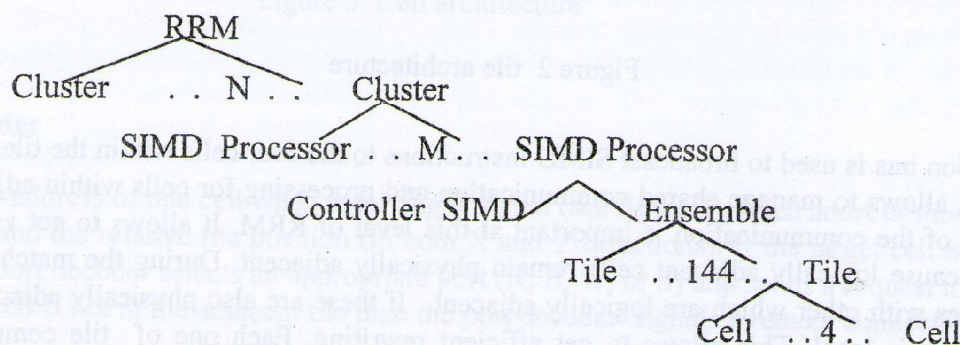


Figure 1 RRM architecture

The RRM architecture is a multi-grain hierarchy, with several different levels of organization, each having a different structure (figure 1). These levels may be described as follows:

- The most basic level is a processing element called a *cell*. It contains registers which store one item of a node of a graph representing a rewriting term and pointers to two other cells.
- Four cells, which share local communication buses, make up a *tile*.
- An *ensemble* and a *controller* make up a *SIMD processor*. The ensemble consists of a number of 144 tiles interconnected with a regular mesh of fixed degree. All cells of the same ensemble execute instructions broadcast from the common central controller.
- A *cluster* interconnects many *SIMD processors* to communicate in a large computation. Two *SIMD processors* can communicate if the rewriting term is partitioned through these two processors.
- A *network* interconnects several clusters to give a complete RRM.

In a single ensemble, instructions representing one rewriting rule are broadcast from a controller to its array of cells. Each cell executes one instruction; hence it executes in SIMD (Single Instruction Multiple Data) mode. A complete RRM consists of many ensembles, each executing its own independent instruction stream; hence it operates in MIMD (Multiple Instruction Multiple Data) mode.

For the purpose of this case-study, only the second level (a tile) will be described.

2.1 Description of a tile

As shown in figure 2, a tile contains four cells and a resource management unit. Particular to this architecture are the buses (for simplicity called N, E, W and S) connecting the tile with adjacent ones[3].

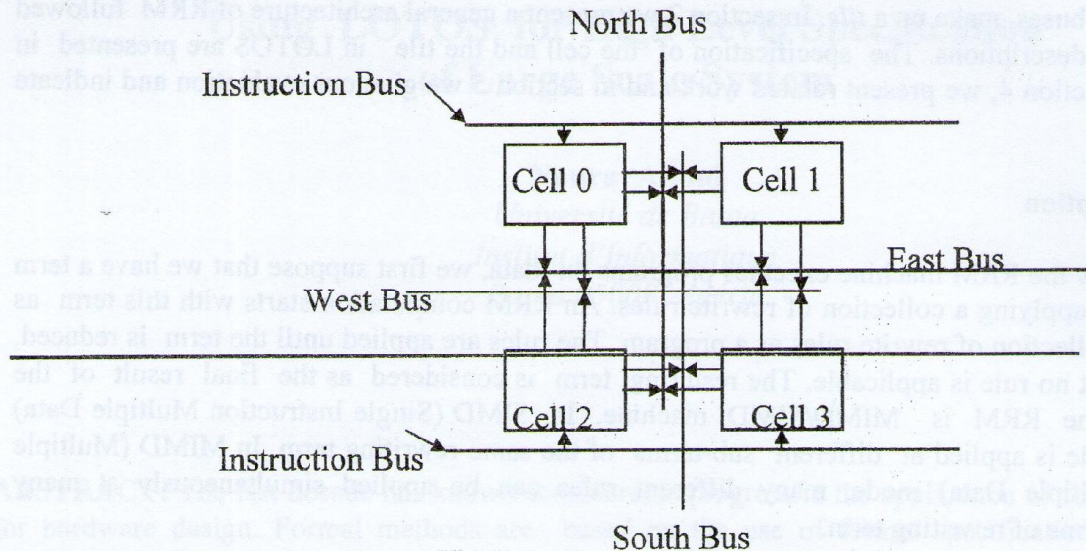


Figure 2 tile architecture

The instruction bus is used to broadcast SIMD instructions to the four cells within the tile. A resource management allows to manage shared communication and processing for cells within adjacent tiles. The locality of the communication is important at this level of RRM. It allows to get very efficient rewriting because logically adjacent cells remain physically adjacent. During the matching one cell communicates with other which are logically adjacent. If these are also physically adjacent then the communication is local. This allows to get efficient rewriting. Each one of tile components will describe bellow.

2.1.1 Description of a cell

Cell structure is determined by the fact that cells provide physical representations for the abstract nodes in a graph, as well as the basic processing power for rewriting.

Figure 3 shows the organization of a cell and its different components are described bellow.

A control unit

It allows the control of executing SIMD instructions sequence, the local communication and the executing of autonomous process in the case of distant communication. These two last operations are not described in our specification. The control unit uses the registers marks, which holds volatile information (similar to condition codes) and a reference counter namely cref.

A 12 bit arithmetic and logic unit (ALU)

It uses the accumulator as one of the operands, and any other register as the second operand. The ALU is capable to move data between registers, compare the content of a register against another register, perform addition, subtraction or bitwise boolean operations. Only move and compare operations are specified in our description.

A register file

It contains the following registers:

- token, which encodes the function or constant symbol of data node,
- left and right, which point to the descendant nodes,
- flags, which holds less volatile information, such as type and reduction status, and
- four general purpose temporary registers, ntoken, nleft, nright and nflags. These registers are not described in our specification.

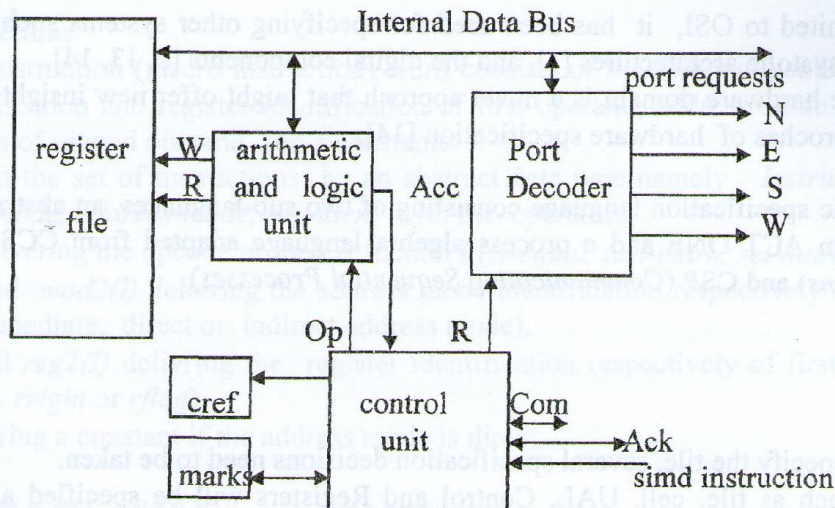


Figure 3 Cell architecture

A port decoder

It decodes an address of one cell when receiving R signal (see figure 3). Such address shows the target cell number and the relative tile position (in both X and Y directions). If the target cell is in adjacent tile then the port decoder selects an appropriate port (N, E, W, or S) and sends a request to select port. If the target cell is not in the adjacent tile then the port decoder signals a remote connection.

2.1.2 Cell Operations and the Instruction Set

Each cell in an *ensemble* executes the SIMD instructions broadcasted by the controller. However, *only active cells* will perform the operation. Here is a short description of only SIMD instructions used in our specification:

- **Arithmetic/logic instructions** compares the content of a register against another register or simply move data between registers.
Examples: (is-equal token 'value) compare the contents of token against value; the cell remains active only when they are equal.
(move 'value token) move value to token register.

- **Marks instructions** test or modify a bit in the marks register.
Examples: (test-mark 'a) tests a bit called 'a' in the marks register; the cell remains active only when it is set.
(set-mark 'a) set a bit called 'a' in the marks register.

- **Init instruction** activates all cells.

2.1.3 Description of a Resource Management Unit

A resource management unit allows the allocation of free cells and dispatching of requests to cells of a tile.

Remark : In this paper we have not specified components responsables of communication between cells such as a port decoder and resource management unit.

3. Formal Specification of RRM in LOTOS

LOTOS (*Language of Temporal Ordering Specification*) has been widely and successfully used to specify communications systems such as standards for OSI (Open Systems Interconnection).

However, it is not limited to OSI, it has been used for specifying other systems such as the CIM architectures [4], the systolic architectures [7], and the digital components [6, 13, 14]. Its investigation in the hardware domain is a novel approach that might offer new insights or benefits compared to other approaches of hardware specification [14].

LOTOS is an algebraic specification language consisting of two sub-languages, an abstract data type language adapted from ACT ONE and a process algebra language adapted from CCS (*Calculus of Communicating Systems*) and CSP (*Communicating Sequential Processes*).

3.1 Design Decision

Before attempting to specify the tile, several specification decisions need to be taken.

1- All components such as tile, cell, UAL, Control and Registers will be specified as process as opposed to abstract data type (ADT). Such decisions have been taken by Sinott [13] for the specification of basic components. The problem with specifying a component, such as a cell, as an ADT is that it would lose the inherently dynamic characteristics of its operations. Another argument arises in the idea of the interconnection of components [13]. By having the cells specified as processes, the idea behind the 'wiring-together' of four cells into a tile is easier to achieve.

2- Sinott [13] has specified a signal by LOTOS event and has used sort Boolean to represent it. At a level of transfert of registers, we think that a signal can be also specified by LOTOS event. However, at this high level of description, we will prefer to use other sorts than Boolean to represent it.

3- The question now arises as to how exactly should the buses be modelled? The bus could be considered as component as well, but to do so would make specifications very unwieldy [14]. In virtually all logic designs the wires can be ignored. Ignoring the wires makes connection of components very easy in LOTOS: events at connected output and input interaction point are synchronised by giving them the same LOTOS interaction point.

3.2 Data types

Associated to a tile, several data sets have to be specified (e.g., the set instructions). We give here an overview of the abstract data types used in the specification to represent these sets.

3.2.1 Registers

The registers used by our specification are token, left, right, flag, marks and cref.

Excepted cref that is defined by a sort *Nat*, all the other registers are defined by a sort *Bit_array*.

type BIT_ARRAY is ARRAY actualized by Bit using
sortnames bit for Item, Bit_array for Array
endtype

type ARRAY is NAT

formalsorts Item, Array

opns put : Item, Nat, Array → Array

-[: Array, Nat → Item

initial : → Array

eqns for all V: Item, I, I': Nat, A: Array

of sort Item

I eq I' ⇒

put (V, I, A) [I'] = V

I ne I' ⇒

put (V, I, A) [I'] = A[I']

end type

3.2.2 Instructions

The SIMD instruction (macro instruction) word consists of the following subfields: opcode, address mode identification and register identification of first operand, address mode identification, register identification of second operand and a constante.

We represent the set of instructions by an abstract data type namely *Instruction* that exports the sorts *inst*, *opcode*, *address-mode*, *reg-ident* with the operators,

- *code(I)* delivering the opcode of the instruction *I* (*is-equal*, *test-mark*, *set-mark*, *move* and *init*),
- *mod1(I)* and *mod2(I)* delivering the address mode identification respectively of the first and second operand (immediate, direct or indirect address mode),
- *reg1(I)* and *reg2(I)* delivering the register identification respectively of first and second operand (*rtoken*, *rleft*, *rright* or *rflag*).
- *cst(I)* delivering a constant if the address mode is direct.

type Instruction is BIT_ARRAY

sorts *inst*, *opcode*, *address_mode*, *reg_ident*

opns *init*, *is-equal*, *test-mark*, *set-mark*, *move* : → *opcode*

rtoken, *rflag*, *rleft*, *rright* : → *reg-ident*

mod1, *mod2* : *inst* → *address_mode*

code : *inst* → *opcode*

immediate, *direct*, *indirect* → *address_mode*

reg1, *reg2* : *inst* → *reg-ident*

cst : *inst* → Bit_array

endtype

3.3 Formal specification of a tile

We represent a tile by a process called *Tile*. The instruction bus is specified by the interaction point *Ins*, the data buses (north, east, south and west) connecting a tile with other adjacent tiles are specified respectively by the interaction points *DN*, *DE*, *DS* and *DW* and both request and acknowledgement buses are represented by the interaction points *RN*, *RE*, *RS* and *RW*. The address of tile in the ensemble is represented by the variables *line* and *column*. The term *hide* is used to indicate that *Con0*, *Con1*, *Con2* and *Con3* are specified as hidden interactions points. These interaction points are not observable from the environment of *Tile* process. The tile structure is expressed in term of four cells and a management resource unit (as it is described in section 2.1). Then, it is specified in term of 4 instances of *Cell* process and an instance of *Management-Resource* process. The process *Cell* has the same interaction points that process *Tile* which are: *Ins*, *DN*, *DE*, *DS*, *DW*, *RN*, *RE*, *RS* and *RW*. In the first instance of *Cell* process, *Con0* modelizes the input of connecting signal received from *Management-Resource* process. The number 0 represent the address of the first instance of *Cell*. All cells of a tile work in parallel and can interchange data through data buses. This behaviour is expressed by using the selective parallel composition operator */[DN, DE, DS, DW]/* between all instances of *Cell* process. The management resource can send to each cell a connecting signal (*Con0*,..*Con3*). The operator */[Con0, Con1, Con2, Con3]/* is used to synchronize each instance of process *Cell* on the appropriate interaction point.

The formal specification of tile in LOTOS is the following:

process Tile[*Ins*, *DN*, *DE*, *DS*, *DW*, *RN*, *RE*, *RS*, *RW*] (*line*, *column*: Nat): noexit:=

hide *Con0*, *Con1*, *Con2*, *Con3* in

(Cell[*Ins*, *DN*, *DE*, *DS*, *DW*, *RN*, *RE*, *RS*, *RW*, *Con0*] (0)

| [DN, DE, DS, DW]

Cell[*Ins*, *DN*, *DE*, *DS*, *DW*, *RN*, *RE*, *RS*, *RW*, *Con1*] (1)

| [DN, DE, DS, DW]

Cell[*Ins*, *DN*, *DE*, *DS*, *DW*, *RN*, *RE*, *RS*, *RW*, *Con2*] (2)

| [DN, DE, DS, DW]

Cell[*Ins*, *DN*, *DE*, *DS*, *DW*, *RN*, *RE*, *RS*, *RW*, *Con3*]) (3)

| [Con0, Con1, Con2, Con3]

Management-Resource [*RN*, *RE*, *RS*, *RW*, *Con0*, *Con1*, *Con2*, *Con3*]

endproc (*Tile*)

3.4 Formal specification of a cell

The structure of a cell, as shown in figure 3, is expressed in terms of a Control Unit, a Port Decoder, an UAL and a File of Registers. We specify these components respectively by processes *Control*, *Port-Decoder*, *UAL* and *Registers*. The signals *Op* and *R* are specified respectively by the interaction points *Op* and *R*. *Acc* represents an accumulator. Both *R* and *W* signals (see figure 3) are specified by only an interaction point namely *R/W*. Data bus is modeled by the interaction point *Data*. Note that *Op*, *Acc*, *R*, *R/W*, *Data* and *Com/Acq* are specified as hidden interaction points (*hide Op, Acc, R, R/W, Data, Com/Acq*). These interaction points are not observable from the environment of *Cell* process. Process *Control* has five interaction points *Ins*, *Com/Acq*, *Op*, *R*, and *Acc*.

This instance takes the initial values (*initial*, 0) to the variables representing registers marks and cref. The *Control* process shares *Op* and *Acc* with *UAL* processes and *R* with *Port-Decoder* processes. This implies that these three processes must synchronize on these three interaction point. This behaviour is expressed by using the selective parallel composition operator $/[Op, R, Acc]/$. The process *Port-Decoder* uses *R* and *Data* as inputs points and *RN*, *BE*, *BS*, *RW*, *RN*, *RE* and *RS* as outputs points. It shares *Data* with *UAL* process. This behaviour is expressed by using the selective parallel composition operator $/[Data]/$ between the two processes.

The *UAL* process has four interaction points: *Op*, *Acc*, *R/W* and *Data*. The instance of *Register* process takes the initial values (*initial*, *initial*, *initial*, *initial*) to all variables representing registers token, flag, left and right. This process shares *R/W* and *Data* with *UAL* process and only *Data* with *Port-Decoder* process. This is expressed by using the operator $/[R/W, Data]/$ between *Registers* process and the two other processes.

Specification of a cell is the following:

```
process Cell [Ins, BN, BE, BS, BW, RN, RE, RS, RW, Con] (Ad: Nat) : noexit:=
hide Op, Acc, R, R/W, Data, Com/Acq in
Control [Ins, Com/Acq, Op, R, Acc] (initial, 0)
|[Op, Acc, R]|
( Port_Decoder [R, Data, BN, BE, BS, BW, RN, RE, RS, RW]
|[Data]|
UAL [Op, Acc, R/W, Data] )
|[R/W, Data]|
Registers [R/W, Data] (initial, initial, initial, initial)
endproc (*Cell*)
```

3.4.1 Formal specification of Control

When a cell is not active (bit 1 of register marks is 0), it can receive the instruction of activation *init* and executes it. This behaviour is specified by taking an input *I* and calling *Control* process with the value 1 in bit 1 of register marks. When a cell is active (bit 1 of register marks is 1), it can receive any other instruction. Note that we specify only the execution of local instructions (ie. when the addressage mode is not indirect).

In this case, if the input code instruction is *test-mark* or *set-mark*, it is the process *Control* that executes it. Else, The instruction *I* is send to *UAL* process through the interaction point *Op*. If the instruction code is *is-equal* then, the process *Control* receives the result of test through the interaction point *Acc*. In the case where the result of test is negative the cell must be inactive. This behaviour is expressed by calling *Control* process with the value 0 in bit 1 of register marks.

```

Process Control [Ins, Com/Acq, Op, R, Acc] (marks: Bit_array, cref: Nat) : noexit :=
[marks[1] eq 0] → (* bit 1 of marks is 0 : cell is not active*)
(Ins ? I: inst [code(I) eq init]; Control[Ins, Com/Acq, Op, R, Acc](put(1, 1, marks), Cref)
  [])
[marks[1] eq 1] → Ins ? I: inst; (*bit 1 of marks is 1: cell is active*)
[(mod1(I) ne indirect) and (mod2(I) ne indirect)] → (*case of local instruction*)
([code(I) eq test-mark] → (* execution of test-mark instruction*)
  [marks[cst(I)] eq 1] → Control [Ins, Com/Acq, Op, R, Acc] (marks, cref)
  [])
[marks[cst(I)] ne 1] → Control [Ins, Com/Acq, Op, R, Acc] (put(0, 1, marks), cref)
  [])
[code(I) eq set-mark] → (* execution of set-mark instruction*)
  Control [Ins, Com/Acq, Op, R, Acc] (put(1, cst(I), marks), cref)
  [])
[code(I) eq move] → Op ! I; Control [Ins, Com/Acq, Op, R, Acc] (marks, cref)
  [])
[code(I) eq is-equal] → Op ! I; Acc ? b: boolean [b eq true]; Control [Ins, Com/Acq, Op, R, Acc] (marks, cref)
  [])
[code(I) eq is-equal] → Op ! I; Acc ? b: boolean [b eq false]; Control [Ins, Com/Acq, Op, R, Acc] (put(0, 1,
marks), cref) )
endproc (*Control*)

```

3.4.2 Formal specification of UAL

The UAL performs all arithmetic and logic instructions. It is specified by a process *UAL* that accepts an instruction *I* from the interaction point *Op* (*Op ? I: inst*).

If the instruction *I* is a move between two registers, the execution of this instruction is specified by the following behaviour :

W/R ! read ! reg1(I); Data ? d: Bit_array; W/R ! write ! reg2(I); Data ! d.

This means that *UAL* sends, through the interaction point *W/R*, a signal *read* and an identification of register *reg1(I)* to the process *Registers*. It waits from the interaction point *Data* the register contain *d*. Then, it sends firstly a signal *write* and an identification of target register *reg2(I)* through the interaction point *W/R*. and secondly the constant *d* through the interaction point *Data*.

If the instruction *I* is a test of a register contain against another, the execution of this instruction is specified by the following behaviour :

W/R ! read ! reg1(I); Data ? d1: Bit_array; W/R ! read ! reg2(I); Data ? d2: Bit_array;

This means that *UAL* sends, through the interaction point *W/R*, a signal *read* and an identification of register *reg1(I)* and it waits from the interaction point *Data* the contain *d1* of this register. Next, it sends a signal *read* and an identification of register *reg2(I)* through the interaction point *W/R*, and it waits from the interaction point *Data* the contain *d2* of this second register. The process *test-equal* is called to verify if *d1* is equal to *d2*. If they are equal, the process *UAL* produces through the interaction point *Acc* an output with true as value (*Acc ! True*), else it produces false.

process UAL [Op, W/R, Data, Acc] : **noexit** :=

```

Op ? I: inst;
[code(I) eq move] → (* execution of a move instruction between two registers*)
( [(mod1(I) eq direct) and (mod2(I) eq direct)] → W/R ! read ! reg1(I); Data ? d: Bit_array; W/R ! write !
reg2(I); Data ! d
  [])
  (* execution of move a constant to a register*)
[(mod1(I) eq immediate) and (mod2(I) eq direct)] → W/R ! write ! reg2(I); Data ! cst(I)
)
[code(I) eq is-equal] → (* execution of test of the contain of a register against another*)

```

```

( [(mod1(I) eq direct) and (mod2(I) eq direct)] → W/R ! read ! reg1(I); Data? d1: Bit_array; W/R ! read !
reg2(I); Data ? d2: Bit_array;
test-equal (d1, d2) >> accept equal: Boolean in
( [equal eq true] → Acc ! true (* result of test is positive*)
  []
[equal ne true] → Acc ! false ) (*result of test is negative*)
)
[] (* execution of test of a contain register against a constant*)
( [(mod1(I) eq immediate) and (mod2(I) eq direct)] → W/R ! read ! reg2(I); Data ?d:array bit;
test-equal (d, cst(I)) >> accept equal: Boolean in
([equal eq true] → Acc ! true (* result of test is positive*)
  []
[equal ne true] → Acc ! false ) (* result of test is negative*)
) >> UAL [Op, W/R, Data, Acc]
where process test-equal (a1, a2: Bit_array): exit (equal: Boolean) :=
[a1[1] eq a2[1]] →
[a1[2] eq a2[2]] →
. . .
[a1[12] eq a2[12]] → exit (true)
[]
choice I: Nat [ ]
[ I le 12] →
[a1[I] ne a2[I]] → exit (false)
endproc (*test-equal*)
endproc (* UAL *)

```

3.4.3 Formal Specification of Registers

We represent the file of registers by a process *Registers* which interacts with its environment through two interaction points *W/R* and *Data*. All the registers that we have specified are presented by the variables *token*, *flag*, *left* and *right* of type *bit_array*. The process Register takes as inputs *c* and *r*. The first input represents a signal indicating either *read* or *write* request, and the second represents register identification (of value *rtoken* for *token*, *rflag* for *flag*, *rleft* for *left* and *rright* for *right* registers). In the case of read request, it produces the value of concerned register as output. For example, the behaviour $[r \text{ eq } rtoken] \rightarrow \text{Data} ! token$ means that concerning the register *token*, the process *Register* sends the value of the variable *token* through the interaction point *Data*. If the request is *write*, it takes *d* as an input and calls the instance of process *Registers* with *d* as value of the concerning register. Example, in the behaviour $[r \text{ eq } rtoken] \rightarrow \text{Data} ? d: \text{bit_array}$; *Registers*[*W/R*, *Data*] (*d*, *flag*, *left*, *right*), it calls *Registers* with a new value *d* for the register *token*. Specification of Registers is the following:

```

Process Registers [W/R, Data] (token, flag, left, right: bit_array) : noexit :=
W/R ? c: command ? r: reg_ident;
[c eq read] →
( [r eq rtoken] → Data ! token
  []
[r eq rflag] → Data ! flag
  []
[r eq rleft] → Data ! left
  []
[r eq rright] → Data ! right ) >> Registers[W/R, Data] (token, flag, left, right)
[]
[c eq write] →

```

```

([r eq rtoken] → Data ? d:bit_array ; Registers[W/R, Data] (d, flag, left, right)
  []
[r eq rflag] → Data ! flag ; Registers[W/R, Data] (token, d, left, right)
  []
[r eq rleft] → Data ! left ; Registers[W/R, Data] (token, flag, d, right)
  []
[r eq rright] → Data ! right ; Registers[W/R, Data] (token, flag, left, d) )
endproc (*Registers*)

```

4. Related work

Different formalisms have been proposed and used in conjunction with digital system specification. A complete overview of these formalisms is outside the scope of the present paper, however, in this section we will only highlight the ones that we will compare with our approach.

Different researchs have used the fonctional programming languages for the specification of digital systems. Among these researchs, Hydra [11] which allows the specification of the behaviour and the structure of digital circuits using fonctional language. It contains an ensemble of fonctions for combining circuits considered as black box to compose more complex circuits. Before combining two circuits, the arguments type of every circuit must match with the argument of the other else the result is an illegal circuit. These types determine what ports must be connected and what direction the wires must get an information. All this tests are useless in a language as LOTOS. It allows specified circuits to be combined simply by synchronising at the appropriate gates. Other remark is that Hydra can not specify bidirectional wires. This would be possible in LOTOS.

Other works have used languages that are based on interval temporal logic for formally specifying a hardware system. Among these languages, Tempura [5], which has been used for large-scale specification of the processor EP/3. In this language only integers, booleans and lists are available as data-types. An other problem is the use of shared variables for the modelisation of communication wires. In LOTOS, using gates for communication is more appropriate.

Computer hardware description languages were the first textual descriptive techniques to be used in the design of hardware. Examples include VHDL [1]. This language is more complex and it has not a general semantics for all its constructs [12].

5. Conclusion

In this paper, we have shown that LOTOS language allows an easy way to describe a complex hardware at the level of registers transfert. With this language, we can combine easily specified structures simply by synchronising at the appropriate interaction points for get other more complex. For example we can combine many instances of *Tile* process for get the *ensemble* specification (the ensemble consists of a number of 144 tiles). LOTOS allows also to represent concurrent systems, giving operations to synchronise parallel processes or have them running independently. As it has been shown in the specification of a *Tile*, the selective parallel composition operator has been used to synchronise four instances of *Cell* process.

Because LOTOS semantics are defined operationally, the *Tile* specification can be directly simulated on an interpreter or translated into an executable program (eg. TOPO [6] which is a translator from LOTOS to C). An other advantage for using LOTOS in the specifying of a tile is that it is based on a set of sound mathematical theories [6] which allow to prove a rich set of algebraic equivalence properties.