SPECIFICATION OF A FUNCTIONAL SYNCHRONOUS DATAFLOW LANGUAGE FOR PARALLEL IMPLEMENTATIONS WITH THE DENOTATIONAL SEMANTICS

Guilhem de WAILLY Fernand BOÉRI, Senior Member IEEE
Thème Architectures Logicielles et Materielles
Laboratoire d'Informatique, Signaux et Systèmes
URA 1376 du CNRS et de l'Université de Nice - Sophia Antipolis
41, bd Napolélon III - 06041 - Nice CEDEX - France
http://alto.unice.fr/~gdw boeri@unice.fr

KEYWORDS

denotational semantics, data-flow, parallelism

ABSTRACT

In this paper, a semantic description of an abstract language using a functional synchronous data-flow (FSDF) paradigm is defined. This work is intended to implement signal processing applications in parallel architectures. This language is as simple as possible.

Some properties are defined, such as time and memory determinisms. In order to establish these properties, criterion functions are established. Relations between properties and criterion functions are proven (these proofs are not in this paper).

Systems are solved with a fully functional abstract machine which allows a great parallelism exploitation. Parallel accesses to the common memory can be statically shared among processors. It results in a cheap and simple parallel architecture without any conflicts management.

1 MOTIVATIONS

A signal processing CAD tool must have an accurate specification language and allow fast implementations. Implementations must be time and memory deterministic.

In order to prove programs and behaviors, the specification language must be functional [2, 15]. Its semantics has to be as transparent as possible to the designer. A graphical representation of applications is welcome: signal processing applications are often thought in terms of boxes (operators) and lines (data paths).

Parallel architectures are expected to be fast. Data-flow modeling preserves inner parallelism of applications. The data-flow language LUCID [1] shows that iterative programs can efficiently be translated into data-flow.

LEE introduced the Synchronous Data-Flow model [16], which focuses on the communications of different clocked processes in a memory-deterministic way, but it is not functional. The three following languages are all data-flow like, functional and time/memory deterministic. SISAL [11] was especially intented for efficient parallel implementations. Later, LUSTRE [13] has been defined to deal with real-time and reactive systems. SIGNAL [3] was designed for signal processing. It has a powerful modularity tool. Both LUSTRE and SIGNAL are built upon clocks defined into the core language.

λ-matrices, the concept developed here, can be seen as a primitive functional [5] abstract language of the languages above. It is specified with the denotational semantics [17]. **Followed-by** is the only temporal operator, which allows all other temporal constructions. Note that its semantics differs from the one used in the languages above. Hence, clocks can be implemented with **alternatives** and **streams**, so they are not defined.

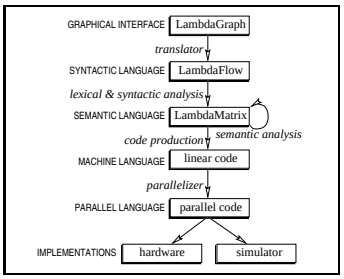

Figure 1: A CAD tool chain for parallel implementations of signal processing applications.

Due to the functional feature and the accurate semantics, programs and behaviors can be proven. Hence, some properties of the λ-matrices are defined, such as **closure**, **calculability** and **stability**. These properties are established in a proven way by the use of **criterion functions** [10]. So, the solving process is time-memory deterministic. λ-matrices are expected to be efficiently implemented onto a specific parallel architecture model, due to the particular solving method [8]. They can be statically compiled [12], and the sharing of the common memory is fixed at compile-time. Parallel architecture becomes very simple without any conflicts management.

This article gives the full semantics description of λ-matrices. This work is the semantics aspect of a CAD tool chain shown in figure 1. This paper only focuses on the denotational description of the language, its solving machine and the criterion functions.

First, an example is given (§ 2): it implements a simple second-order recursive filter with λ-matrices language, and shows concisely that the solving method is suitable for parallelism exploitation (§ 2.5). The abstract language is introduced in an informal way (§ 3), followed by its denotational definitions (§ 4). Then, the functional abstract solving machine is described (§ 5), followed by the crite-

rion functions (§ 6). Then, we explain briefly why the solving process is time/memory deterministic. At last, some extensions of the λ-matrices (§ 8) are described.

2 EXAMPLE

This simple example shows how an application can be translated into the λ-matrices language. It has two parts: the theoretical description of a simple signal processing filter with Z-equations [14], and its translation into our formalism.

2.1 Theoretical description

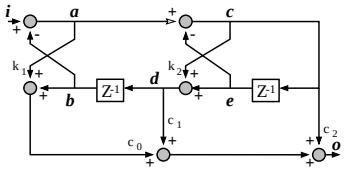

Figure 2: *Second order recursive filter.*

A diagram of a second order recursive filter is shown in figure 2. Its Z-equations are:

$$\begin{cases} o = a + b + c + d \\ a = i - b \\ b = Z^{-1} d \\ c = a - e \\ d = c + e \\ e = Z^{-1} c \end{cases} \quad (1)$$

The simplification of these equations (1) gives the transfer function:

$$H(Z) = \frac{O(Z)}{I(Z)} = \frac{3+3Z^{-1}+Z^{-2}}{1+2Z^{-1}+Z^{-2}}. \quad (2)$$

Now, the recurrent time function can be deduced:

$$o(t) = 3i(t)+3i(t-1)+i(t-2)-2o(t-1)-o(t-2). \quad (3)$$

The first values are calculated with input values $i = \langle i_1, i_2, \ldots, i_n \rangle$:

$$\begin{array}{l} o_1 = 3i_1 \\ o_2 = 3i_2 - 3i_1 \\ o_3 = 3i_3 - 3i_2 + 4i_1 \\ o_4 = 3i_4 - 3i_3 + 4i_2 - 5i_1 \\ \cdots \end{array} \quad (4)$$

2.2 Translation into a λ-matrix

The corresponding λ-matrix can be written from the diagram in figure 2. The method is to give a name to each node and to write its value from the diagram. The obtained λ-matrix is:

$$\left[\begin{array}{l} O := +(+(+(A,\ B),\ C),\ D) \\ A := -(i,\ B) \\ B := 0\ f_{by}\ D \\ C := -(A,\ E) \\ D := +(C,\ E) \\ E := 0\ f_{by}\ C \end{array} \right]. \quad (5)$$

There is a great similarity between these equations (5) and Z-equations (1): λ-matrices data-flow semantics is well-adapted to write signal processing equations. In addition, they allow a full modular design of applications.

2.3 Resolution of this λ-matrix

Let us suppose that the input i has the followed values i_0, i_1, ..., i_n. The system (5) is solved with a functional abstract machine. This machine defines a fixed-point equation that alternates **evaluations** and **regenerations** of the system until a stopping condition is reached. In our example, this condition does not exist and the solving process has no end.

The abstract machine can be modeled with the diagram in figure 3. In this figure, the state vector contains all the initial values of the stream states. The new state values are computed by the evaluation operator, and a new system is built with these values by the regeneration operator. This step is infinitely repeated by the recursive resolution operator.

Note that obtained results are proven because the solving expression is fully functional.

2.4 High-level language λ-flow

```
filter is lambda in:real.begin
  out  ! A + B + C + D;
  A    := in - B;
  B    := 0.0 followed-by D;
  C    := A - E;
  D    := C + E;
  E    := 0.0 followed-by C;
end
```

λ-matrices are an abstract language. They allow the modularity of applications at a low level. So, a high level data-flow language has been defined, named λ-FLOW. It especially deals with modularity based on the modularized version of the λ-matrices [7] and gives a convenient interface to the designer. It is possible to program the example in figure 2 with this language:

2.5 Parallelism considerations

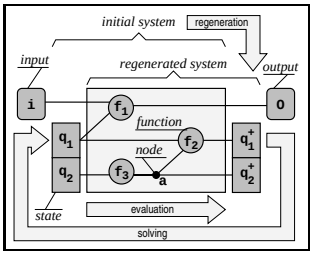

Figure 3: Solving of a system. f_1, f_2 and f_3 are functions.

The resolution of a λ-matrix has three steps: input sampling, evaluation-time and stream regeneration-time. The evaluation operator **gives a value** to each actor, while the regeneration operator **regenerates** stream states.

In figure 3, the solving process computes each component of the target state vector[1]. The order of these evaluations is unimportant, but they must be all terminated when the system is regenerated. The resolution method clearly separate the parallel code from the sequential code: function f_1, f_2 and f_3 can be evaluated in parallel (f_3 is evaluated twice, without any optimization), and then, the target state vector is copied into the source state vector.

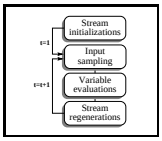

Figure 4: Scheduling.

But this mechanism is well adapted with the assumption of an infinite number of processors. Indeed, as shown in figure 3, the node a is evaluated twice. A temporary variable can be put in place of the node. The value of this variable is evaluated, and the resulting value is used twice. But the introduction of temporary variables implies some functional dependencies: the node a has to be evaluated before f_2 and q_2^+.

The solving scheduling is shown in figure 4. Streams are first initialized. Then, inputs are sampled, and all

[1]Note that in a real implementation, there is only one state vector: the memory.

the temporary variables are evaluated depending upon their functional dependencies. Stream states are evaluated, and the process returns to the inputs sampling.

Now, classical methods can be used to schedule the tasks [4]. Note that the implementation is static: we have proposed a simple parallel architecture that plainly exploit this fact (§ 8.2).

3 λ-MATRICES DATA-FLOW

The systems are described with an abstract language. This language is composed of a set of **actors** suitable for a data-flow programming. We have defined **atoms**, **basic objects** (alternative, application, definition and stream) and primitive **modularity objects** (extraction and vector). The behavior of these objects is given by the **dynamic operators** (regeneration, evaluation and reduction) (§ 5).

Atoms form the alphabet of the language. Natural integers and their associated operators, user data and their specific operators[2], identifiers, some comparators, $\bot$ and $\top$ are atoms. Syntax of atoms is usual.

Alternative is a choice between two expressions depending on a condition. It is written : $condition \rightarrow then, else$. If the evaluation of $condition$ is not 0 then the result is the evaluation of $then$, else it is the evaluation of $else$.

Application acts as a filter of its arguments according to the operator semantics. It is written: $op(arg_1, arg_2, \ldots, arg_n)$, which applies the operator op to the arguments arg_i.

Definition allows identifier-value associations. It is written : $ident := value$. This writing denotes a definition and not an affectation. The identifier becomes a synonym of the value in the current environment or in its sub-environments. When an identifier does not match a definition in the current environment, it is considered as an input. The inputs at the top-level system are reserved identifier such as $input_i$.

Stream allows to write in a functional way a recurrent equation. Such an equation can be seen in different ways: it is a **delay** operator in automation control, and it is a **state variable** in software engineering. Stream is written: $state\ f_{by}\ contract$. It has two parts : a **state** which contains the initial value and a **contract** for computing the following state values[3].

Vector is a basic structuring object that gathers expressions into the same structure. A vector also defines a frame of environment where identifier-value associations are created by definition objects. A vector is written: $\left[\begin{smallmatrix} c_1 \\ \vdots \\ c_n \end{smallmatrix}\right]$, where $c_1 \ldots c_n$ are actors.

[2] User data and their operators define an algebra. Integer algebra is available by default. Several algebra can coexist in the same system.

[3] Note that this stream semantics differs from the one used in LUCID,... Indeed, if a denotes the stream $a = \langle a_1, a_2, \ldots, a_n \rangle$ and b the stream $b = \langle b_1, b_2, \ldots, b_n \rangle$, then the stream $c = \langle a_1, b_2, b_3, \ldots, b_n \rangle$ can be obtained with the expression $c := (1\ f_{by}\ 0) \rightarrow a, b$ and the stream $c = \langle a_1, b_1, b_2, \ldots, b_n \rangle$ with $c := a\ f_{by}\ b$.

Vectors are a primitive tool for a modularized design, greatly enhanced by the modularized λ-matrices [7].

Extraction can read an indexed value inside a vector. Its writing is : $module \cdot index$. Extractions are an explicit functional mechanism for defining multi-output functions.

4 DENOTATIONAL DEFINITIONS

The following sections provide a formal denotational semantics for the primitive expressions of the λ-matrices. A convenient introduction can be found in LLOYD [17].

4.1 Standard expressions

This section describes some syntactic forms of useful functions in the semantics.

$\langle \ldots \rangle$	:	sequence formation
$s \downarrow k$	:	kth member of the sequence s (1-based)
$\#s$	:	length of sequence s
$s \S t$	:	concatenation of sequences s and t
$s \dagger k$	:	drop the first k^{th} members of sequence s
$t \rightarrow a \oplus b$	:	McCarthy conditional "if t then a else b"
$\rho[x/i]$	:	substitution "ρ with x for i"
$x\ in\ D$	:	injection of x in domain D
$x \vert D$	:	projection of x in domain D

4.2 Abstract syntax

The elements of the language can be given:

A	$\in Alg$	:	user's algebra dependent
N	$\in Int$	:	integers
I	$\in Ide$	:	identifiers
F	$\in Flag$	:	flags
O	$\in Ope$	:	operators
$\underline{E}$	$\in \underline{Exp}$	:	expressions
$\overline{E}$	$\in \overline{Exp}$	:	frozen expressions

With these elements, valid expressions of the language can be defined:

Exp	$\longrightarrow$	A	: algebra
		I	: identifier
		N	: integer
		F	: flag
		$E \mapsto E, E$	: alternative
		$O(E^*)$	: application
		$E\ f_{by}\ E$	: stream
		$I = E$	: definition
		$E.E$	: extraction
		$[E^*]$	: vector

In addition, **frozen expressions** are:

$\overline{Exp}$	$\longrightarrow$	A	: algebra
		F	: flag
		N	: integer
		$[\overline{E}^*]$	: vector

They correspond to the result of the evaluations operator (§ 5.3).

4.3 Semantic domains

Symbol = is used for testing the equality and symbol ≡ indicates the equivalence. Semantic domains are:

$$
\begin{array}{rcll}
t \in & \mathcal{D} & & : \text{data} \\
\xi, n \in & \mathcal{N} & & : \text{integer} \\
o \in & \mathcal{O} & & : \text{operator} \\
i \in & \mathcal{I} & & : \text{identifier} \\
f \in & \mathcal{F} & \equiv \{\perp\} & : \text{flag} \\
a \in & \mathcal{T} & \equiv \mathcal{X} \times \mathcal{X} \times \mathcal{X} & : \text{alternative} \\
p \in & \mathcal{P} & \equiv \mathcal{O} \times (\{\mathcal{D} + \mathcal{N}\}^*) & : \text{application} \\
d \in & \mathcal{F} & \equiv \mathcal{I} \times \mathcal{X} & : \text{definition} \\
e \in & \mathcal{E} & \equiv \mathcal{X} \times \mathcal{X} & : \text{extraction} \\
s \in & \mathcal{S} & \equiv \mathcal{X} \times \mathcal{X} & : \text{stream} \\
v \in & \mathcal{V} & \equiv \mathcal{X} \times \mathcal{X}^* & : \text{vector} \\
\epsilon, \pi \in & \mathcal{X} & \equiv \mathcal{D} + \mathcal{N} + \mathcal{I} + \mathcal{T} + \mathcal{P} & \\
 & & \quad + \mathcal{F} + \mathcal{E} + \mathcal{S} + \mathcal{V} & : \text{expression} \\
\overline{\epsilon}, \overline{\pi} \in & \overline{\mathcal{X}} & \mathcal{D} + \mathcal{N} + \mathcal{F} + \mathcal{V} & : \text{frozen expression} \\
\kappa \in & \mathcal{K} & \equiv \mathcal{X} \to \mathcal{X} & : \text{continuation} \\
\rho \in & \mathcal{R} & \equiv \mathcal{V}^* & : \text{environment}
\end{array}
$$

The environment structure is not the classical one: here, an environment is a list of vectors. In these vectors, only the definitions will be considered. In semantics equations below, environment parameters are named ρ. The function *lookup* i ρ retrieves the value associated to the identifier i, function *lexical* i ρ retrieves the environment where the identifier i is defined and the function *extend* i π ρ adds the definition $i = \pi$ to the topmost vector of the given environment ρ.

Because the state of the system is globally modified at regeneration-time and because there is no side-effect operator in the language, the usual indirect data access (store) between a name and a value is not needed.

4.4 Semantic equations

Semantic equations operate on the expressions of the abstract language and produce the expressions of the semantic domains. Their signatures are:

$$
\begin{array}{rll}
\triangleright & : Exp \to \mathcal{N} \to \mathcal{R} \to \mathcal{K} & : \text{solving} \\
\bigtriangledown & : Exp \to \mathcal{R} \to \mathcal{K} & : \text{regeneration} \\
\bigtriangleup & : Exp \to \mathcal{R} \to \mathcal{K} & : \text{evaluation} \\
\triangleleft & : Exp \to \mathcal{K} & : \text{reduction} \\
\Delta_l & : Exp \to \mathcal{R} \to \mathcal{K} & : \text{closure} \\
\Delta_c & : Exp \to \mathcal{R} \to \mathcal{K} & : \text{calculability} \\
\Delta_s & : Exp \to \mathcal{R} \to \mathcal{K} & : \text{stability} \\
\sigma & : Exp \to \mathcal{R} \to \mathcal{K} & : \text{norm}
\end{array}
$$

5 ABSTRACT MACHINE

5.1 Resolution $\triangleright$

Only closed (§ 6.1), calculable (§ 6.2) and stable (§ 6.3) systems can be solved. The solving operator is a tail-recursive function. Its main parameters are an environment and a system. In addition, it has a stopping condition that allows the recursion to be ended. This stopping condition is an integer that indexes a value inside the system: when the extracted value is 0 the resolution is terminated and the continuation is invoked.

The solving operator alternates **evaluations** (§ 5.3) and **regenerations** (§ 5.2) of the system until the stopping condition is reached. Its expression is:

$$
\begin{array}{l}
\triangleright[\![\,[\pi^*]\,]\!]\alpha\rho\kappa \equiv \\
\qquad \bigtriangleup^*[\![\,\pi^*\,]\!]\rho\S\langle[\pi^*]\rangle\,\lambda\epsilon^*.\epsilon^* \downarrow \alpha \neq 0 \quad \to \bigtriangledown[\![\,[\pi^*]\,]\!]\rho\lambda\epsilon.\ \triangleright[\![\,\epsilon\,]\!]\alpha\rho\kappa \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \oplus\kappa\ [\epsilon^*]
\end{array}
$$

where $[\pi^*]$ is the system to be solved, α the stopping condition integer, ρ the current environment and κ the continuation. Note that alternative objects of the language are written $c \mapsto t, e$ and denotational alternatives are written $c \to t \oplus e$. In the equation, $\rho\S\langle[\pi^*]\rangle$ is the extended environment, ϵ^* the sequence of the evaluated components of the system.

5.2 Regeneration $\bigtriangledown$

This operator **regenerates all the stream states** of a system in a synchronous way. The new stream states are the evaluation result of their contract ($\bigtriangledown[\![\,\pi f_{by} \pi'\,]\!]$). All the components of other objects are regenerated one by one. The regeneration of a vector is the vector of the regenerated components in the extended environment (a vector plays the role of a frame of an environment). The expression of the regeneration operator is:

$$
\begin{array}{rl}
\bigtriangledown^*[\![\,\pi\pi^*\,]\!]\rho\kappa & \equiv \bigtriangledown[\![\,\pi\,]\!]\rho\lambda\epsilon.\ \bigtriangledown^*[\![\,\pi^*\,]\!]\rho\lambda\epsilon^*.\kappa\langle\epsilon\rangle\S\epsilon^* \\
\bigtriangledown^*[\![\,\,]\!]\rho\kappa & \equiv \kappa\ \langle\rangle \\
\hline
\bigtriangledown[\![\,\pi \mapsto \pi', \pi''\,]\!]\rho\kappa & \equiv \bigtriangledown[\![\,\pi\,]\!]\rho\lambda\epsilon.\ \underset{\kappa\ \epsilon \mapsto \epsilon', \epsilon''}{\bigtriangledown}[\![\,\pi'\,]\!]\rho\lambda\epsilon'.\ \bigtriangledown[\![\,\pi''\,]\!]\rho\lambda\epsilon''. \\
\bigtriangledown[\![\,o(\pi^*)\,]\!]\rho\kappa & \equiv \bigtriangledown^*[\![\,\pi^*\,]\!]\rho\lambda\epsilon^*.\kappa\ o(\epsilon^*) \\
\bigtriangledown[\![\,f, i, t, n\,]\!]\rho\kappa & \equiv \kappa\ f, i, t, n \\
\bigtriangledown[\![\,i = \pi\,]\!]\rho\kappa & \equiv \bigtriangledown[\![\,\pi\,]\!]\rho\lambda\epsilon.\kappa\ i = \epsilon \\
\bigtriangledown[\![\,\pi.\pi'\,]\!]\rho\kappa & \equiv \bigtriangledown[\![\,\pi\,]\!]\rho\lambda\epsilon.\ \bigtriangledown[\![\,\pi'\,]\!]\rho\lambda\epsilon'.\kappa\ \epsilon.\epsilon' \\
\bigtriangledown[\![\,\pi f_{by} \pi'\,]\!]\rho\kappa & \equiv \bigtriangleup[\![\,\pi'\,]\!]\rho\lambda\epsilon.\ \bigtriangledown[\![\,\pi'\,]\!]\rho\lambda\epsilon'.\kappa\ \epsilon\ f_{by}\ \epsilon' \\
\bigtriangledown[\![\,[\pi^*]\,]\!]\rho\kappa & \equiv \bigtriangledown^*[\![\,\pi^*\,]\!]\,\rho\S\langle[\pi^*]\rangle\,\lambda\epsilon^*.\kappa\ [\epsilon^*]
\end{array}
$$

5.3 Evaluation $\bigtriangleup$

This operator **gives a value** to the expressions of the language. The application evaluation ($\bigtriangleup[\![\,o(\pi^*)\,]\!]$) is the reduction (§ 5.4) of the application of the operator to the evaluated arguments (applicative mode).

An extraction ($\bigtriangleup[\![\,[\pi^*].\pi\,]\!]$) extracts a component of a vector with an integer. This selection is performed with a projection into the tested domain ($\epsilon|_{integer}$).

The evaluation of an identifier ($\bigtriangleup[\![\,i\,]\!]$) is the evaluation of its associated value (*lookup* i ρ) in the environment where the identifier was defined (*lexical* i ρ) (lexical binding). This evaluation may not have an end if the expression defines a fixed-point equation with the form $x = f(x)$. Such expressions are detected with the calculability criterion (§ 6.2). The expression of the evaluation operator is:

$$
\begin{array}{rl}
\bigtriangleup^*[\![\,\pi\pi^*\,]\!]\rho\kappa & \equiv \bigtriangleup[\![\,\pi\,]\!]\rho\lambda\epsilon.\ \bigtriangleup^*[\![\,\pi^*\,]\!]\rho\lambda\epsilon^*.\kappa\ \langle\epsilon\rangle\S\epsilon^* \\
\bigtriangleup^*[\![\,\,]\!]\rho\kappa & \equiv \kappa\ \langle\rangle \\
\hline
\bigtriangleup[\![\,\pi \mapsto \pi', \pi''\,]\!]\rho\kappa & \equiv \bigtriangleup[\![\,\pi\,]\!]\rho\lambda\epsilon. \\
 & \qquad \epsilon \neq 0 \to \bigtriangleup[\![\,\pi'\,]\!]\rho\kappa \oplus \bigtriangleup[\![\,\pi''\,]\!]\rho\kappa \\
\bigtriangleup[\![\,o(\pi^*)\,]\!]\rho\kappa & \equiv \bigtriangleup^*[\![\,\pi^*\,]\!]\rho\lambda\epsilon^*.\triangleleft[\![\,o(\epsilon^*)\,]\!]\kappa \\
\bigtriangleup[\![\,f, t, n\,]\!]\rho\kappa & \equiv \kappa\ f, t, n \\
\bigtriangleup[\![\,i = \pi\,]\!]\rho\kappa & \equiv \kappa\ \perp \\
\bigtriangleup[\![\,[\pi^*].\pi'\,]\!]\rho\kappa & \equiv \bigtriangleup^*[\![\,\pi^*\,]\!]\,\rho\S\langle[\pi^*]\rangle\,\lambda\epsilon^*.\ \bigtriangleup[\![\,\pi'\,]\!]\rho\lambda\epsilon. \\
 & \qquad \epsilon|_{integer} = \perp \to \kappa\ \perp \oplus \kappa\ \epsilon^* \downarrow \epsilon \\
\bigtriangleup[\![\,\pi.\pi'\,]\!]\rho\kappa & \equiv \bigtriangleup[\![\,\pi\,]\!]\rho\lambda\epsilon. \\
 & \qquad \epsilon|_{vector} = \perp \to \kappa\ \perp \oplus \bigtriangleup[\![\,\epsilon.\pi'\,]\!]\rho\kappa \\
\bigtriangleup[\![\,i\,]\!]\rho\kappa & \equiv \bigtriangleup[\![\,lookup\ i\ \rho\,]\!](lexical\ i\ \rho)\kappa \\
\bigtriangleup[\![\,\pi f_{by} \pi'\,]\!]\rho\kappa & \equiv \bigtriangleup[\![\,\pi\,]\!]\rho\kappa \\
\bigtriangleup[\![\,[\pi^*]\,]\!]\rho\kappa & \equiv \bigtriangleup^*[\![\,\pi^*\,]\!]\,\rho\S\langle[\pi^*]\rangle\,\lambda\epsilon^*.\kappa\ [\epsilon^*]
\end{array}
$$

Note that the evaluation results belong to frozen expressions, a subset of the expressions of the language (§ 4.2).

5.4 Reduction $\triangleleft$

The reduction operator simply applies the operator to the arguments and gives the result to the continuation. This operator was separated from the evaluation process because its activity depends on the algebra used. In this way, algebra dependent computations are clearly separated from the λ-matrices dependent computations.

$$
\begin{array}{rl}
\triangleleft[\![\,o(\pi^* \perp \pi'^*)\,]\!]\kappa & \equiv \kappa\ \perp \\
\triangleleft[\![\,o(\pi^*)\,]\!]\kappa & \equiv \kappa\ o\ \pi^*
\end{array}
$$

At least, if one argument is $\perp$, the result is $\perp$.

6 CRITERION FUNCTIONS

6.1 Closure Δ_l

This operator allows to check if a system is closed. In a closed system, each encountered identifier matches a definition in the current environment. This function uses the *lookup* environment function that returns the associated value to an identifier into a hierarchical environment, or $\perp$, if there is no corresponding definition ($\Delta_l[\![\, i \,]\!]$).

$$
\begin{array}{rl}
\Delta_l^*[\![\, \pi\pi^* \,]\!]\rho\kappa & \equiv \Delta_l[\![\, \pi \,]\!]\rho\lambda\epsilon.\epsilon \to \Delta_l^*[\![\, \pi^* \,]\!]\rho\kappa, \kappa\ 0 \\
\Delta_l^*[\![\ \ ]\!]\rho\kappa & \equiv \kappa\ 1 \\
\hline
\Delta_l[\![\, \pi \mapsto \pi', \pi'' \,]\!]\rho\kappa & \equiv \Delta_l^*[\![\, \langle \pi, \pi', \pi'' \rangle \,]\!]\rho\kappa \\
\Delta_l[\![\, o(\pi^*) \,]\!]\rho\kappa & \equiv \Delta_l^*[\![\, \pi^* \,]\!]\rho\kappa \\
\Delta_l[\![\, f, t, n \,]\!]\rho\kappa & \equiv \kappa\ 1 \\
\Delta_l[\![\, i = \pi \,]\!]\rho\kappa & \equiv \kappa\ 1 \\
\Delta_l[\![\, \pi.\pi' \,]\!]\rho\kappa & \equiv \Delta_l^*[\![\, \langle \pi, \pi' \rangle \,]\!]\rho\kappa \\
\Delta_l[\![\, i \,]\!]\rho\kappa & \equiv \kappa\ (lookup\ i\ \rho) = \perp \to 0 \oplus 1 \\
\Delta_l[\![\, \pi f_{by} \pi' \,]\!]\rho\kappa & \equiv \Delta_l^*[\![\, \langle \pi, \pi' \rangle \,]\!]\rho\kappa \\
\Delta_l[\![\, [\pi^*] \,]\!]\rho\kappa & \equiv \Delta_l^*[\![\, \pi^* \,]\!]\rho\S\langle [\pi^*] \rangle\ \kappa
\end{array}
$$

It can be proven that the system provided by the regeneration of a closed system is also closed: **closure is conservative** [10].

6.2 Calculability Δ_c

When a closed system is calculable, it does not contain any fixed-point equation. A fixed-point equation occurs when the associated value of a definition uses the identifier of the definition, such as $x = f(x)$[4].

Each component of objects is checked with a simple tree-marking method: encountered symbols are over-defined in the environment with a special value ($(lexical\ i\ \rho)\S\langle [i = \perp] \rangle$); then, if a symbol is found to be associated with this special value, a cycle is created.

$$
\begin{array}{rl}
\Delta_c^*[\![\, \pi\pi^* \,]\!]\rho\kappa & \equiv \Delta_c[\![\, \pi \,]\!]\rho\lambda\epsilon.\epsilon \to \Delta_c^*[\![\, \pi^* \,]\!]\rho\kappa, \kappa\ 0 \\
\Delta_c^*[\![\ \ ]\!]\rho\kappa & \equiv \kappa\ 1 \\
\hline
\Delta_c[\![\, \pi \mapsto \pi', \pi'' \,]\!]\rho\kappa & \equiv \Delta_c^*[\![\, \langle \pi, \pi', \pi'' \rangle \,]\!]\rho\kappa \\
\Delta_c[\![\, o(\pi^*) \,]\!]\rho\kappa & \equiv \Delta_c^*[\![\, \pi^* \,]\!]\rho\kappa \\
\Delta_c[\![\, f, t, n \,]\!]\rho\kappa & \equiv \kappa\ 1 \\
\Delta_c[\![\, i = \pi \,]\!]\rho\kappa & \equiv \kappa\ 1 \\
\Delta_c[\![\, \pi.\pi' \,]\!]\rho\kappa & \equiv \Delta_c^*[\![\, \langle \pi, \pi' \rangle \,]\!]\rho\kappa \\
\Delta_c[\![\, \pi f_{by} \pi' \,]\!]\rho\kappa & \equiv \Delta_c[\![\, \pi \,]\!]\rho\lambda\epsilon.\epsilon = 0 \to \kappa\ 0 \\
 & \quad \oplus \Delta_c[\![\, \pi' \,]\!]\langle\rangle\kappa \\
\Delta_c[\![\, i \,]\!]\rho\kappa & \equiv \lambda\epsilon.\epsilon = \top \to \kappa\ 0 \\
 & \quad \oplus \epsilon = \perp \to \kappa\ 1 \\
 & \quad \oplus \Delta_c[\![\, lookup\ i\ \rho \,]\!] \\
 & \quad\ (extend\ i\ \top\ (lexical\ i\ \rho))\kappa \\
\Delta_c[\![\, [\pi^*] \,]\!]\rho\kappa & \equiv \Delta_c^*[\![\, \pi^* \,]\!]\rho\S\langle [\pi^*] \rangle\ \kappa
\end{array}
$$

It can be proven that the system provided by the regeneration of a calculable system is also calculable: **calculability is conservative** [10].

6.3 Stability Δ_s

The dimension[5] of stable systems is identical to the dimension of its regenerated system.

An alternative must have the norm (§ 6.4) of its *then* equal to the norm of its *else* ($\Delta_s[\![\, \pi \mapsto \pi', \pi'' \,]\!]$). An extraction must have a vector or an identifier for indexed value, and an integer for index ($\Delta_s[\![\, \pi.\pi' \,]\!]$). A stream must have a frozen expression (§ 4.2) as initial state, and the norm of its state must be equal to the norm of its contract ($\Delta_s[\![\, \overline{\pi} f_{by} \pi \,]\!]$). With these additional conditions, the dimension of a stable system is constant [10]. The expression is:

$$
\begin{array}{rl}
\Delta_s^*[\![\, \pi\pi^* \,]\!]\rho\kappa & \equiv \Delta_s[\![\, \pi \,]\!]\rho\lambda\epsilon.\epsilon \to \Delta_s^*[\![\, \pi^* \,]\!]\rho\kappa, \kappa\ 0 \\
\Delta_s^*[\![\ \ ]\!]\rho\kappa & \equiv \kappa\ 1 \\
\hline
\Delta_s[\![\, \pi \mapsto \pi', \pi'' \,]\!]\rho\kappa & \equiv \sigma[\![\, \pi' \,]\!]\rho\lambda\xi.\sigma[\![\, \pi'' \,]\!]\rho\lambda\xi'.\xi = \xi' \\
 & \quad \to \Delta_s^*[\![\, \langle \pi, \pi', \pi'' \rangle \,]\!]\rho\kappa \oplus \kappa\ 0 \\
\Delta_s[\![\, o(\pi^*) \,]\!]\rho\kappa & \equiv \Delta_s^*[\![\, \pi^* \,]\!]\rho\kappa \\
\Delta_s[\![\, f, i, t, n \,]\!]\rho\kappa & \equiv \kappa\ 1 \\
\Delta_s[\![\, i = \pi \,]\!]\rho\kappa & \equiv \kappa\ 1 \\
\Delta_s[\![\, [\pi^*].n \,]\!]\rho\kappa & \equiv \Delta_s[\![\, [\pi^*] \,]\!]\rho\kappa \\
\Delta_s[\![\, i.n \,]\!]\rho\kappa & \equiv \kappa\ 1 \\
\Delta_s[\![\, \pi.\pi' \,]\!]\rho\kappa & \equiv \kappa\ 0 \\
\Delta_s[\![\, \overline{\pi} f_{by} \pi \,]\!]\rho\kappa & \equiv \sigma[\![\, \overline{\pi} \,]\!]\rho\lambda\xi.\sigma[\![\, \pi \,]\!]\rho\lambda\xi'. \\
 & \quad \xi = \xi' \to \Delta_s[\![\, \pi \,]\!]\rho\kappa \oplus \kappa\ 0 \\
\Delta_s[\![\, \pi f_{by} \pi \,]\!]\rho\kappa & \equiv \kappa\ 0 \\
\Delta_s[\![\, [\pi^*] \,]\!]\rho\kappa & \equiv \Delta_s^*[\![\, \pi^* \,]\!]\rho\S\langle [\pi^*] \rangle\ \kappa
\end{array}
$$

It can be proven that the system provided by the regeneration of a stable system is also stable: **stability is conservative** [10].

6.4 Norm σ

The norm of an expression is the biggest dimension of all the possible results obtained by evaluation. Several dimensions can be obtained with the use of alternatives, according to the condition evaluation result.

So, the norm of an alternative ($\sigma[\![\, c \mapsto \pi, \pi', \pi'' \,]\!]$) is the biggest norm of either its *then* or its *else* clauses. The norm of applications ($\sigma[\![\, o(\pi^*) \,]\!]$) is 1 because an application returns either a data or $\perp$ (the dimension of data is supposed to be 1). The norm of identifier ($\sigma[\![\, i \,]\!]$) is the norm of the corresponding value ($lookup\ i\ \rho$) in the corresponding environment ($lexical\ i\ \rho$). The norm expression is:

$$
\begin{array}{rl}
\sigma^*[\![\, \pi\pi^* \,]\!]\alpha\rho\kappa' & \equiv \sigma[\![\, \pi \,]\!]\rho\lambda\epsilon.\sigma^*[\![\, \pi^* \,]\!]\alpha\rho\lambda\epsilon^*.\kappa\ (\alpha\ \epsilon\ \epsilon^*) \\
\sigma^*[\![\ \ ]\!]\alpha\rho\kappa & \equiv \kappa\ (\alpha\ 0\ 0) \\
\hline
\sigma[\![\, c \mapsto \pi, \pi', \pi'' \,]\!]\rho\kappa & \equiv \sigma[\![\, \pi' \,]\!]\rho\lambda\epsilon'.\sigma[\![\, \pi'' \,]\!]\rho\lambda\epsilon''. \\
 & \quad \kappa\ \epsilon' > \epsilon'' \to \epsilon' \oplus \epsilon'' \\
\sigma[\![\, o(\pi^*) \,]\!]\rho\kappa & \equiv \kappa\ 1 \\
\sigma[\![\, f, t, n \,]\!]\rho\kappa & \equiv \kappa\ 1 \\
\sigma[\![\, i = \pi \,]\!]\rho\kappa & \equiv \kappa\ 1 \\
\sigma[\![\, [\pi^*].n \,]\!]\rho\kappa & \equiv \sigma[\![\, \pi^* \downarrow n \,]\!]\rho\kappa \\
\sigma[\![\, [\pi^*].\pi \,]\!]\rho\kappa & \equiv \sigma^*[\![\, \pi^* \,]\!] > \rho\S\langle [\pi^*] \rangle\ \kappa \\
\sigma[\![\, i.\pi \,]\!]\rho\kappa & \equiv \sigma[\![\, lookup\ i\ \rho.\pi \,]\!](lexical\ i\ \rho)\kappa \\
\sigma[\![\, \pi.\pi' \,]\!]\rho\kappa & \equiv \kappa\ 1 \\
\sigma[\![\, i \,]\!]\rho\kappa & \equiv \sigma[\![\, lookup\ i\ \rho \,]\!](lexical\ i\ \rho)\kappa \\
\sigma[\![\, \pi f_{by} \pi' \,]\!]\rho\kappa & \equiv \sigma[\![\, \pi \,]\!]\rho\kappa \\
\sigma[\![\, [\pi^*] \,]\!]\rho\kappa & \equiv \sigma^*[\![\, \pi^* \,]\!] + \rho\S\langle [\pi^*] \rangle\ \kappa
\end{array}
$$

7 DETERMINISMS

The expression of the resolution operator is tail-recursive: it defines a fixed-point equation (§ 5.1). Its main parameter is the system to be solved [π^*]. If the stopping condition is not reached, the result of the current resolution is the resolution of the regenerated system $\nabla[\![\, [\pi^*] \,]\!]\rho$.

We have defined some properties such as closure, calculability and stability. A system can be solved only if it has these properties. The question is: if a system S_0 has these properties, does the regenerated system $S_1 = \nabla S_0$ have them, and more generally, does all the next regenerated systems $S_t = \nabla S_{t-1}$ have them ? We proved that the answer is yes with some induction proofs [10].

The most important criterion is stability: it deals with the dimension of systems. The dimension can be viewed as the amount of information necessary to describe the system. We have proven that the regenerated system of a

[4] But recurrent equations (built with a stream) are allowed, such as $x = 0\ f_{by}\ +(1, x)$ that defines the stream of natural integers.

[5] The dimension of an atom is 1, and the dimension of a compounded object is the sum of each its component dimension.

stable system is also stable, which implies that the dimension remains constant. We do not study computation-time: we only say that if a system is stable, the global computation time has a limit, without wanting to know it. With fine time-studies, λ-matrices could efficiently be used to describe real-time systems.

8 OUR FURTHER PLANS

8.1 User friendly interface

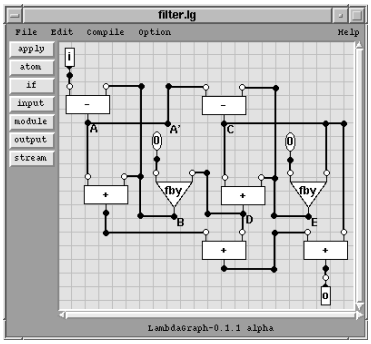

Figure 5: *Graphical interface of the λ-matrices.*

A graphic object-oriented interface of a λ-matrix named λ-graph [6] can be seen in figure 5. A syntactic data-flow compilable language named λ-FLOW was designed. This is the upper level of this work, which provides the great convenience of user friendly interfaces. These languages are independent of the used algebra: "things" that manipulates programs are not known in advance.

8.2 Parallel architecture

The parallelism of the model will be exploited in a specific architecture. Concurrent problems are avoided because the λ-matrix model clearly separates the sequential and parallel program blocks (§ 2.5).

In addition, due to its functional feature, the order of computations is not important. Functional dependencies only exist with intermediate variables, which can be seen as optimizations. Already, the parallel compiler principles are designed [8]. An example of a specific architecture is shown in figure 6. The scheduling is obtained at compile-time, merely with a pseudo-execution. Due to the stability criterion, this scheduling is always valid. So the memory management is avoided, merely replaced by 3-states chips. A simulator of this parallel architecture has been built [9].

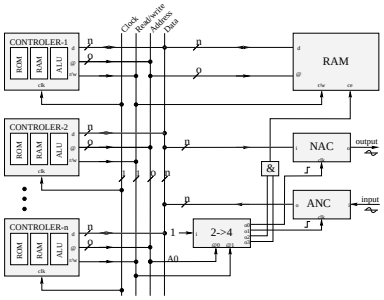

Figure 6: *Model of the λ-matrix* MIMD *architecture.*

9 CONCLUSION

This paper describes the functional semantics of an abstract language named λ-matrix. It is a functional synchronous data-flow-like language (FSDF). This language is intended to implement signal processing applications on parallel architectures. It is the semantics aspect of a CAD tools chain built by the authors. It contains a graphical data-flow editor, a syntactic language and its sequential compiler, a parallel compiler and a parallel architecture simulator.

This functional semantics description has three parts: the description of the elements of the abstract language, the abstract solving machine and the criterion functions. The description of elements of the language is based on a simple abstract grammar. The solving process is composed of two activities: evaluation that gives a value to expressions, and regeneration that recycles stream states. This particular method increases parallelism exploitation of the programs. Three criterion functions are defined, each associated to its property. These functions are relative to time-memory determinisms of implementations.

References

[1] E. A. Ashcroft and W. W. Wadge. Lucid, a Nonprocedural Language with Iteration. *j-CACM*, 20(7):519–526, July 1977.

[2] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *j-CACM*, 21(8):613–641, aug 1978.

[3] A. Benveniste and P. LeGuernic. Hybrid dynamical systems theory and the SIGNAL language. *IEEE Transactions*, 35:535–546, 1990.

[4] M. Cosnard and D. Trystram. *Algorithmes et architectures parallèles.* InterÉdition, 1993.

[5] G. de Wailly. Implémentation des λ-matrices à l'aide du λ-calcul. Technical Report 95-34, I3S, july 1995.

[6] G. de Wailly. User manual of lambda graph, the graphical interface of the functional synchronous data flow language lambda flow. Technical Report 95-33, I3S, july 1995.

[7] G. de Wailly and F. Boéri. A formal specification of modularized λ-matrices. Technical Report 95-57, I3S, october 1995.

[8] G. de Wailly and F. Boéri. A parallel architecture for lambda matrices, a functional data-flow abstract language. Technical Report 95-42, I3S, july 1995.

[9] G. de Wailly and F. Boéri. A parallel architecture simulator for the lambda matrices. In *Association of Lisp Users Meeting and Workshop Proceedings.* LUV'95, august 1995.

[10] G. de Wailly and F. Boéri. Proofs upon basic and modularized λ-matrices. Technical Report 95-69, I3S, december 1995.

[11] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the SISAL language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, 1990.

[12] P. Fradet and D. Le Métayer. Compilation of functional languages by program transformation. In *Transaction on Programming Languages and Systems*, volume 13,1, pages 21–51. ACM, 1991.

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. Programmation et vérification des systèmes réactifs : le langage LUSTRE. *Techniques et Sciences Informatiques*, 10(2):139–158, 1991.

[14] M. Kunt. *Traité d'Électricité-Traitements numérique des signaux.* Edition Georgi, 1980.

[15] P.J. Landin. The next 700 programming languages. *Communication of ACM*, 9:157–166, march 1966.

[16] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[17] A. Lloyd. *A pratical introduction to denotational semantics.* Cambridge Computer Science Texts 23, 1986.