A CAD Tool Chain for Signal Processing, with Parallel Implementation Issues

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
{gdw|boeri}@unice.fr

Abstract

In this paper, a CAD tool chain is globally presented. These tools allow graphical descriptions of signal processing applications to be implemented in a specific parallel architectures, via several transformation steps.

The user interfaces of this tool chain is a graphical editor and a syntactic language, both based on a semantic functional synchronous data flow formalism. They allow a modular design of application.

This CAD tool chain is built upon an accurate semantic data flow language named λ-matrix. The language is independent with the user's algebra: it is "a thing that handles things". Several criterion functions are defined to determine some properties such as time and memory determinisms.

λ-matrices can be compiled for several targets. Programs are firstly flattened with a functional operator that keeps the semantics. Then a sequencer produces the code, according to the selected target.

Due to the particular solving method which allows parallelism exploitation, we have defined a cheap parallel architecture built with common controllers.

1 Introduction

A **signal processing** application differs from the other engineering softwares in two ways: **design** and **implementation**.

The design depends on the mathematic tools used to model the system [16]. So, the model will have a simple and an accurate **semantics**. This semantics will be as transparent as possible and will not interact with the user's thought.

In addition, systems are often written in a **graphic** way. In such a graph, boxes are operators and lines are data paths. Because of the modularity, boxes could be either basic operators or complex modules.

A **modular** design of systems is essential: it is the basis of imperative languages such as C, and it allows separate compilation. A module is only known by its interface and its inner implementation remains hidden to the final user.

Implementation of signal processing systems require both safety and speed.

Whatever the input values, the **safety** protects a system against any deadlock. In addition, safety attests the existence of an upper bound of the needed memory and the computation times. The safety may be incrementally established module after module at design time. Because of all these requirements, this model has to be built with a formal proof system. **Functional** languages [1, 4, 19] are built upon the simple and powerful λ-calculus [20], itself built on mathematics. This tool allows proofs, but it is too expressive and not specific enough for signal processing and its implementation will remain inefficient.

Parallel design of architectures [6] is expected to increase the speed of applications.

Implementation of **data-flow** graphs are expected to give good results in the parallelism exploitation [3, 5, 17]. Indeed, their semantics keeps the parallelism of the applications, unlike the imperative languages which forces the programmer to think in terms of sequences. Only synchronous data-flow are suitable for signal processing because of time constraints. Of course, graphic representations of applications are possible.

The main idea in this work is to use the **best features** of both functional and data-flow concepts. The first one allows proofs of programs while the second is suitable for a graphic representation of signal processing applications while preserving their inner parallelism.

In this paper, a **CAD Tool chain** for signal processing implementation is globally described. This tool chain includes a graphical editor of programs (λ-graph), a syntactic language (λ-FLOW), a formal language (λ-matrix), a parallelizer compiler and a parallel architecture simulator. This chain is shown in figure 1.

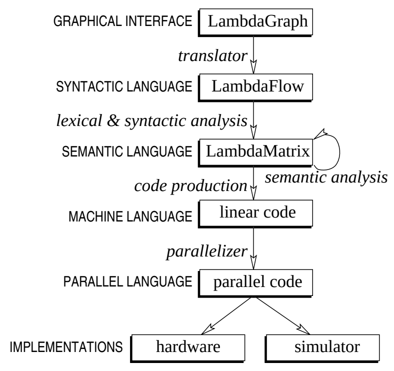


Figure 1: *A CAD tool chain for parallel implementation of signal processing.*

The whole tool chain is build upon a main abstract data flow language named λ-matrix[1]. Proofs of programs and their behaviors are possible.

Firstly, the graphical editor is describe (§ 2). Then, the syntactic language is shown (§ 3). It is built upon an abstract functional synchronous data flow language (§ 4). Then the parallel architecture and its simulator are described (§ 5). At least, the parallel compiler can be seen (§ 6).

[1]The "λ" of "λ-matrix" denotes the ability to write this model with λ-expressions [7] and "matrix" is because it is a kind of vectorial calculus.

2 Graphical Editor

The graphical editor λ-graph [8] allows graphical programming in the **data flow** style. The used semantics is functional synchronous data flow (FSDF), based on an algebraic abstract language, the λ-matrix (§ 4). The layout of this editor can be seen in figure 2.

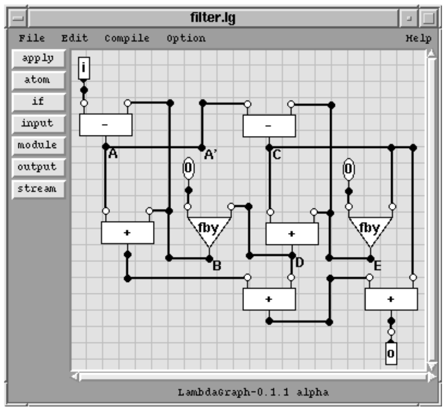


Figure 2: *Graphical editor.*

On the left side of the window, the buttons in the **command** panel allow the user to create actors with a click in the **canvas** area. Actors can be moved, configured, selected, linked and destroyed. The **clipboard** allows easy cut/copy/paste operations. The data flow diagram of a second order recursive filter was drawn in the canvas area in figure 2. It is constituted with actors (`-`, `i`, `fby`,...) handles (`A'`) and links (`A`$\rightarrow$`A'`). An actor has some inputs (in top) and some outputs (in bottom).

An output can only be **linked** to either an input or a handle, and it can only have one link. Handles can be viewed as deferred outputs. They have the same properties than outputs, but they can be separately moved, selected, linked and destroyed. Inputs, outputs and handles can have a name (`A`, `A'`). Links can be destroyed with a click.

Atoms and applications are configured according to the current used algebra (**mixed algebra** in the same module is allowed). An algebra defines a **data type** featured by its name and its **check function**. The check function controls the value the user enters in the configuration dialog-box. In addition, several **opera-**

tors are defined in an algebra. They are basic entities featured by their signature that defines the type of the arguments and the type of the returned value. Operators only have one output. In the example, `+` and `-` operators are provided by the integer algebra. In addition, the string `"0"` is checked as an integer.

A **program** (also named a module) is defined by its body that contains several actors linked together, inputs and outputs ports. It can be saved, loaded and printed in a Postscript form. Modules are organized in libraries. A module can be copied into a library, that allows its reuse. In the example, `filter` is a main module.

This editor is unable to run a program. It only can call **translators** that transform a program into other forms. Several translations are possible. The authors are working in a SCHEME translator that allows direct simulations of programs, with a special library that contains wave-generators and wave-viewers. They are also working in a λ-FLOW translator (§ 3). In addition, other targets can be considered: SISAL [12], SIGNAL [5], LUSTRE [14], PTOLEMY with Lee's Synchronous Data-Flow (SDF) [17].

λ-graph is written with the STK SCHEME language that interfaces the TK library [13, 18].

3 Syntactic language

The **syntactic** description of programs uses our language named λ-FLOW. It is a user-readable language based upon the λ-matrix semantics (§ 4).

The example described in a graphical way in the previous section can be written with this language, as shown in figure 3.

In this example, two modules are written, `filter` and `main`. The main module instantiates the filter module with argument `i#int`, the input of the system, and it names this instantiation `instance`. Thus, it reads the output of the filter with an extraction and writes it in its own output `output`.

As into λ-graph, atoms (`"0"`) and application operators (`+`) are relative to an algebra: they are added to the lexical units of the language. The used algebra is defined at compile-time. So, the same λ-FLOW

```
filter = lambda i. BEGIN
 O ! +(+(+(A,B),C),D); ' output
 A = -(i,B);           ' definition
 B = 0 FOLLOWED-BY D;  ' stream
 C = - (A, E);         ' application
 D = +(C,E);           '
 E = 0 FOLLOWED-BY C;
END

MAIN = BEGIN
 instance = filter(i#int);
 output   ! instance EXTRACT 0;
END
```

Figure 3: Second order recursive filter with λ-FLOW

program can have different behaviors, according to the used algebra. λ-FLOW defines the following objects:

Atoms are the basic expressions of the language. Natural integers and their associated operators, user's defined data and their specific operators, identifiers, some comparators are atoms. Users can specify their **own algebra** built upon their data-type and relative operators. The syntax of atoms is checked by the algebra. The integer algebra is available by default because some λ-FLOW operators use integers as parameters.

Alternative is a choice between two expressions depending on a condition. It is written:

```
IF  condition  THEN
  then-clause
ELSE
  else-clause
```

λ-matrices use integer as booleans: 0 denotes **false** while the others inetgers denotes **true**.

Application acts as a filter of its arguments according to its operator semantics. It is written:

```
OPERATOR (arg-1, ..., arg-n)
```

The semantics of the operator is given by the used algebra.

Definition allows identifier-value associations. In the current environment (see vector) or in the sub-environments, this name becomes a synonym of the expression. It is written:

```
name = value
```

Stream allows to write in a functional way a recurrent equation [3]. A stream has two parts: **state** which contains the initial value and **contract** for computing its next values. It is written:

```
state  FOLLOWED-BY  contract
```

All the streams of the program will be regenerated (the action that updates the state) in the same time. So, the λ-matrices are synchronous.

Vector is a structured object that gathers some expression in an indexed way. A vector is written:

```
BEGIN
  components-1;
  ...
  components-n;
END
```

A vector must has at least one component. It defines a frame of an **environment** [1]. All the definitions it contains are visible into all the inner expressions. Identifiers are statically linked into an environment, as in the language SCHEME [1]. This statically linkage allows efficient compilation [15]. The top-level environment contains only the module definitions. In addition, the λ-matrices are referentially transparent, so, an identifier can be replaced with its associated value everywhere it is used.

Extraction can read an indexed value inside a module. It is an explicit functional mechanism for multi-outputs. It is written:

```
indexed  EXTRACT  index
```

`Indexed` must be an identifier, a vector or a module instantiation. `Index` can be either an integer for direct addressing, or an identifier. If `index` is an identifier, it must match with an output with the same name in `indexed` (see output). In the example in figure 3, the index `O` in the extraction of the main module matches with the `O` output of the filter.

Output exports a value for extraction. An output is written:

```
name ! value
```

Note that the outputs of the main module are outputs of the system.

Module instantiation instantiates a module with some arguments. Arguments of the instantiation are statically linked. It is written as an application:

```
module (arg-1, ..., arg-n)
```

Abstraction abstracts an actor with some parameters. It is written:

```
lambda p-1, p-2, ..., p-n. actor
```

Parameters p_i are identifiers. The actor can be a vector or another expression, but the expression must not contain free variables. Inputs of the system are identifier with the form *name#type*.

This syntactic language is translated with a parser to the formal λ-matrices language. In this point of view, λ-matrices can be viewed as internal representations of the compiler [2].

4 Formal language

λ-matrices are an **abstract language** with a functional synchronous data flow semantics [11]. The main goal of this formalism is to describe and solve applications in a functional way, and to prove time and memory **determinisms**.

The λ-matrices can be entirely implemented with λ-expressions [7] that confirms the **functional** feature of the model. This implementation is interesting because it shows that the solving expression of a program is resumed to one λ-expression.

Objects of the language are atoms, alternatives, applications, definitions, numeric extractions, streams and vectors. This abstract language does not define symbolic extractions and modules: this is the role of the syntactic language λ-FLOW. But a **primitive** modularization is possible with vectors an numeric extractions.

The **solving operator** is fully tail-recursive (the solve operator is the only part where a recurrent equation is allowed). The solving method alternates **evaluation** of the systems that acts as a valuation of all the expressions of the program, and **regeneration** that synchronously recycles all the streams of the program.

In addition, some **criterion** functions are defined. They establish in a proven way time and memory determinisms of systems [10].

A **closed** system does not have any free variables. So, a definition in the current environment always matches each encountered identifier.

When a system does not contain any fixed-point equation ($x = f(x)$), it is said **calculable**. These equations are detected with a cycle-detector which operates on the system definitions. It is proven that the evaluation of a calculable system always exists, as well as its regeneration. In addition, all the regenerated systems from a calculable system are also calculable.

The amount of memory necessary to store a **stable** system is constant during its solving. This criterion is the more severe of all. It imposes some conditions in the alternatives, streams and extractions. All the regenerated systems from a stable system are also system.

The λ-matrices can be either executed by the **abstract machine** they define, or compiled. The structure of the **compiler** is in figure 6.

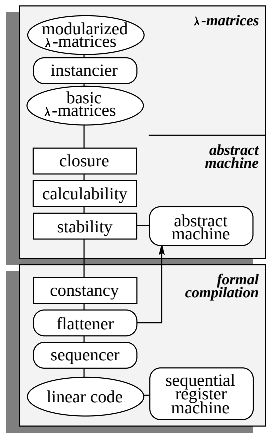


Figure 4: *Running λ-matrices: a) interpretation mode with the abstract machine, b) formal compilation to run it in a sequential machine.*

Only **constant**[2] programs can be compiled. Then, programs are **flatened** and the resulting code is given to a **sequencer** that produces the linear code, according to the selected target.

5 Parallel Architecture

The wanted architecture is shown in figure 5. Simplicity is its main characteristic.

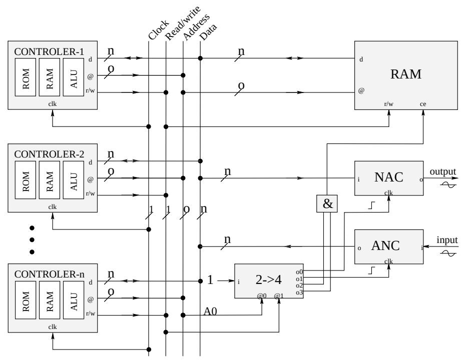


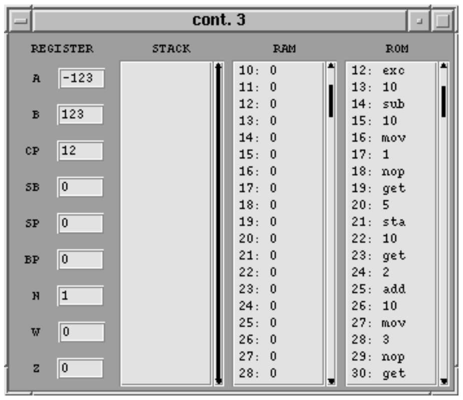
Figure 5: *Model of the specific parallel architecture that implements the λ-matrices.*

The parallel architecture is a very simple cheap board built with **common controllers**. Controllers are directly connected to the bus: sharing of bus accesses is allocated at compile-time. The board could be divided into three main parts: controllers, a **common memory** which includes the global RAM and the IOs lines, and the **main bus**. In the model, one input line and one output line are connected to converters, themselves connected to the global bus. These lines are accessible by address decoding: in the example, the address 0 is reserved for IOs.

A software **simulator** [9] of this architecture was conceived. As shown in figure 6, `cont. 3` is the graphical representation of one controller. A controller has its own RAM, ROM, and some registers. The stack is mapped on the RAM.

The graphic representation of the **common memory** can be seen in figure 7. It can be seen the global RAM which begins at the address `1`, because the address `0` is reserver for IOs. In addition, there are

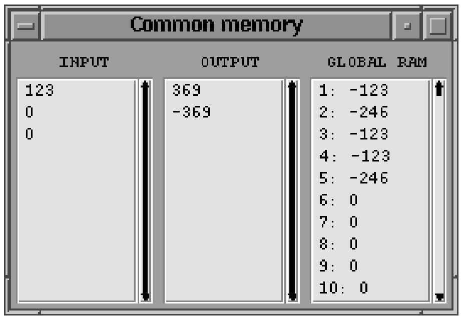
[2]The constancy deals with the signature of data.



```
REGISTER      STACK   RAM        ROM
A   -123              10:  0     12:  exc
B   123               11:  0     13:  10
CP  12                12:  0     14:  sub
SB  0                 13:  0     15:  10
SP  0                 14:  0     16:  mov
BP  0                 15:  0     17:  1
N   1                 16:  0     18:  nop
W   0                 17:  0     19:  get
Z   0                 18:  0     20:  5
                      19:  0     21:  sta
                      20:  0     22:  10
                      21:  0     23:  get
                      22:  0     24:  2
                      23:  0     25:  add
                      24:  0     26:  10
                      25:  0     27:  mov
                      26:  0     28:  3
                      27:  0     29:  nop
                      28:  0     30:  get
```

Figure 6: *Interface of a controller.*

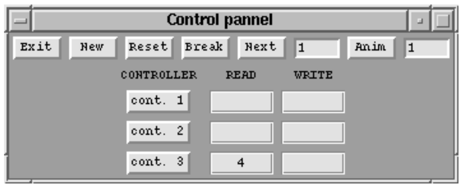
two lists of values: input values read from the input file, and output values written into the output file.



```
INPUT   OUTPUT   GLOBAL RAM
123     369      1:  -123
0       -369     2:  -246
0                3:  -123
                 4:  -123
                 5:  -246
                 6:  0
                 7:  0
                 8:  0
                 9:  0
                 10:  0
```

Figure 7: *Interface of the common memory, with the common memory and the input and output values list.*

Control panel is the graphical representation of the main bus, as in figure 8. Global memory accesses can be seen for each controller. The system can be clocked either step by step or in an animation way. Breakpoints can be instantiated for each controller.



```
Exit  New  Reset  Break  Next  1   Anim  1
CONTROLLER   READ   WRITE
cont. 1
cont. 2
cont. 3      4
```

Figure 8: *Interface of the main bus.*

The complete graphic interface code takes about 600 lines because the TK interface and its object-oriented layer is particularly efficient in STK.

6 Parallel Compiler

The **solving** 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. These operators are said dynamic because they carry away the current environment built as one goes along.

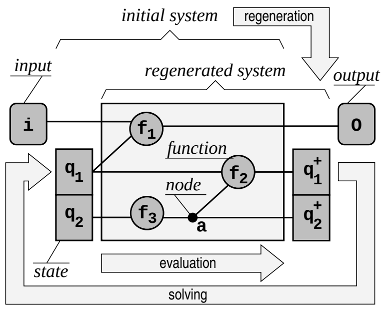


Figure 9: *Solving of system.*

In figure 9, the solving process computes each component of the target state vector. The order of these evaluations is unimportant, but they must be all terminated when the state vector is regenerated. It can be thought that the target vector is **copied** in the source one, as an abstracted view of this regeneration. The solving method clearly separates the parallel code from the sequential code: the exploitation of the **parallelism** is optimal.

But this mechanism is well adapted with the assumption of an infinite number of processors. Indeed, as shown in figure 9, the node a is twice evaluated. A temporary variable can be put in place of the node. The value of this variable is evaluated, and the resulting value is twice used. But the introduction of temporary variables implies some functional dependencies: the node a has to be evaluated before f_2 and q_2^+.

So, the solving **scheduling** is shown in figure 10. Stream are first initialized with their state value. Then, inputs are sampled, and all the temporary variables are evaluated depending to their functional

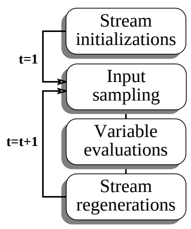


Figure 10: *Scheduling of a data-flow system.*

dependencies. The stream states are evaluated, and we return to input sampling.

Note that in a real implementation, there is only one state vector: the memory. So, the stream state evaluations are also ordered according to their own functional dependencies.

The **list-scheduling** produces a list from the set of tasks, depending on some considerations. Here, tasks with the **largest processing time** are first chosen: the processing time is the sum of the duration of the task and the duration of all its successors [6]. This algorithm has a $O(n^2)$ complexity.

In our model, tasks are divided into instructions, themselves composed by read, write or other instructions. The main idea of the **task interleaving** mechanism is to start a task before all its predecessors are ended: tasks can read their available inputs and only wait for their unavailable ones, as shown in figure 11.

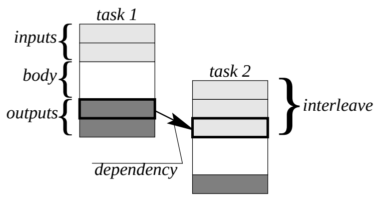


Figure 11: *These two sequential tasks can be efficiently interleaved.*

In a theoretical way, the gain cannot be measured, and the GRAHAM 69 limits is still valid [6]:

$$w \leq w_{opt}(2 - \frac{1}{p})$$

where w_{opt} is the duration of the optimal scheduling and p the number of processors, with $p \geq 3$[3].

In practice, the interleaving greatly enhances the efficiency of program implementations.

The **bus sharing** mechanism allocates a task to each free processor. Then it tries to write the current instruction of each processor. The instruction is always written if it does not use the bus. Otherwise, the processor are ordered depending on their own level and the current level. If the chosen processor want to read a variable that it is not yet available, it tries to read its others inputs. If it is not possible, the next processor is chosen.This algorithm has a $O(n^2)$ complexity.

7 Furthers works

The main parts of the CAD tool chain was already conceived: λ-graph interface, λ-FLOW language, λ-matrices semantic language, parallel compiler principles and parallel simulator.

The authors are currently working to the **code production** of the λ-matrices compilation. They want to produce code for other tools such as PTOLEMY or SISAL.

In addition, the tool chain can compile only stable programs. The stability criterion is very rigorous and does not allow **recurrent equations** inside a program. The authors work in order to allow recurrent equations inside a module without a cost in the parallelism exploitation.

Another direction is to use the local controller memories as **caches** of the global memory. When a controller cannot access the bus as it wants, it listens the bus and copies the transfered data into its local memory. This mechanism is expected to significantly reduce the bus traffic.

8 Conclusion

In this paper, a CAD tool chain is described. This chain allows a graphical description of **signal processing** programs and their implementations on a specific cheap parallel architecture.

The **graphical editor** is the most user-friendly interface of the tool chain. Pro-

[3]This expression does not take into account the bus sharing.

grams can be edited with some facilities, such as copy/cut/past operations.

A graphical description can be translated into a **syntactic language**. This language is more powerful than the graphical description and more constructions can be written. It especially deals with the modularity concept of an abstract language defined for a semantic purpose.

These user-friendly interfaces are based on a functional synchronous data flow **formal language**. It defines a full functional solving method and some criterion functions. Time and memory determinism can be established in a proven way. This abstract language can be compiled for a specific parallel architecture.

The **parallel architecture** that implements programs of the model is simple and cheap. It was conceived with the assumption that data transfers can be known at compile-time. So, the controllers are directly connected to a common bus that accesses to the common memory and the IOs lines. This architecture avoids the management of the possible access conflicts.

In order to test the principles of the tool chain in without great investments, a software **simulator** of the parallel architecture is written. It is the last tool of the tool chain.

References

[1] H. Abelson, G.J. Susman, and J. Susman. *Structure and Interpretation of Computer Programs.* MIT Press, 1987.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers.* Addison-Wesley Publishing Company, Inc, 1986.

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

[4] 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.

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

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

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

[8] 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.

[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] G. de Wailly and F. Boéri. Specification of a functional synchronous dataflow language for parallel implementation with the denotationnal semantics. In *Symposium on Applied Computing.* ACM, february 1996.

[12] 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.

[13] E. Gallesio. Stk reference manual, version 2.2. Technical report, Laboratoire I3S-CNRS URA 1376 - ESSI, e-mail:kaolin.unice.fr:/pub/, october 1995.

[14] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. In *Proceedings of the IEEE*, pages 1305–1319, 1991. Published as Proceedings of the IEEE, volume 79, number 9.

[15] S.L. Peyton Jones. *The implementation of Functional Programming Languages.* Prentice Hall International, 1987.

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

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

[18] J.K. Ousterhout. *Tcl and the Tk toolkit.* Addison-Wesley, 1994.

[19] C. Queinnec. *Les langages* LISP. Inter Édition, 1994.

[20] G.E. Revesz. *Lambda-Calculus, Combinators, and Functional Programming.* Cambrige University Press, 1988.