A DATAFLOW LANGUAGE FOR SIGNAL PROCESSING MODELING 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

This paper describes λ-FLOW, a new functional synchronous dataflow language for DSP applications. It is independent of the handled data. It plainly supports the modular design. Its sound semantics allows proofs of programs and time/memory determinisms. The target code is dynamically loaded into the compiler with a target description that is defined with less than twenty lines of definitions. Due to the static feature of the solving model, it is possible to implement programs onto a static parallel architecture.

1 INTRODUCTION

Here, digital signal processing (DSP) applications are limited to the application based on state model. A **state-model** is shown in figure 1.

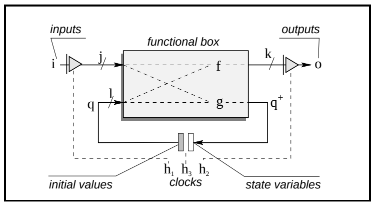

Figure 1: State model.

A state-model application has several inputs and several outputs, respectively i and j, and a **state-vector** with k values. The state-vector is the memory of the system. The outputs are a function f of the inputs and of the state-vector. The state-vector is also a function g of the inputs and the current state-vector. The functional box encloses true function, without inner loop. This system is scheduled with three clocks h_1, h_2 and h_3, in this order. Clock h_1 samples the inputs and clock h_2 sends the outputs. Clock h_3 copies the new state-vector values in their location. The mechanism that handles the clocks is out of the system.

A such state-based application can be easily modeled with a diagram. DSP designers know the formalism based on the Z **delay operator**. A filter conceived with this method is shown in figure 2.

The filter is a box which can have several inputs and several outputs. The nodes are either operations or links, and lines are data paths. In such a **diagram**, there is no direct cycle: cycles can occur only through Z delay operators. This operator plays the role of a cell of the state-vector in a state-model. It allows the time to be handled in the diagram. The filter in figure 2 has only one clock, but it could be conceived with more that one clock. From this diagram, it is possible to obtain the transfer-function $H(Z)$ of the filter, and then, its recurrent equation $o(t)$.

Tools used to implement this kind of DSP application must have several properties. A **graphical programming interface** is welcome, but complicated applications are often difficult to understand this way. So, a **syntactic programming language** is also necessary, with a translation between the two formalisms. In addition, these programming environments have to support a **modular design** of the applications.

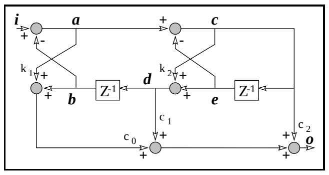

Figure 2: DSP filter.

The DSP designer wants to be sure it programs what he thinks. So, the programming language must have a **sound semantics**, as simple as possible. Particularly, the semantics tool has to allow proofs of programs.

Generally, **speed** of DSP implementations is a strong constraint. In addition, they are often embarked systems, which are expected to have zero-defaults. So, the used tools must allows efficient implementations.

The existing tools are shortly examined (§ 2). Then, our functional synchronous dataflow (FSD) language is informally explained (§ 3), and we try to light the differences between the formalisms cited above. Programs can be either solved with a functional abstract machine which gives the semantics of the language (§ 4) or compiled into several target code (§ 5). Due to the time/memory determinisms of the language, programs can be implemented onto a cheap, simple and static parallel architecture (§ 6). Last, we explain our future work (§ 7).

2 EXISTING TOOLS

As shown in figure 2, DSP applications are closed to the **dataflow** concept. A dataflow program is a diagram with lines as data paths and boxes as operations. It exists two methods to run a dataflow program.

The first method to run a dataflow programs is the **data-driven** method. When a data is available on each input of an operator, the operator computes a new data that it puts on its output. The researches on dataflow parallel computers started with MILLER and KARP in 1966 [17]. But this kind of dataflow suffers to the lack of a global semantic description of the program and the inefficiency of the implementations.

The second method to run a dataflow program is the **demand-driven** method. Here, a result is asked to an operator. The operator propagates the demand to its empty inputs. When all the inputs data are available, the operator computes a data and it returns it. This kind of dataflow is closed to the functional programming style [3].

Functional languages [18, 5] have all a valuable property: they are built on the mathematically based λ-calculus [6, 21]. Functional programming languages can be efficiently implemented onto a classical VON NEUMANN architecture, which provides low cost specialized DSP processors and well known programming environments.

The first functional dataflow language is LUCID [4]. It is the first to demonstrate that a dataflow programming style can replace iteration, with the advantage of the functional property. But LUCID contains several features not well adapted to DSP. Particularly, it is not **time/memory deterministic**.

The SISAL is introduced to implement general purpose FSD program onto parallel architecture. But it is not adapted for DSP for the same reasons than LUCID [13].

LUSTRE and SIGNAL are two FSD languages well adapted for DSP. Their kernel is based on recurrent clocked equations. SIGNAL does not define explicitly a root clock while the clocks in LUSTRE are all defined on a base clock [15, 14].

Our language λ-FLOW is a part of a CAD tool chain for implementing DSP application onto parallel architectures [11]. It is more **primitive** than the languages cited above. It has less expressions and less concepts: it defines only one temporal operator, used to built a stream of values. The streams are updated in a synchronous way, so, the language could be used for **real-time** applications.

It provides the **alternative** construction (if-then-else). Associated to the stream object, the alternatives could be used to define some clocks. So, the clock concept is not explicitly defined in the core language.

The integer **algebra** is predefined by default. All the other handled data are dynamically bound to an algebra. This feature gives to the language a great generality and adaptability: it is defined as a "thing to handle some things", without specifying the nature of the handled things, such as the LANDIN's language [18].

λ-FLOW is a **typed language**, but its type checking has less constrains than the languages cited above: it encourages **polymorphic** abstractions. In addition, it supports a full lexical scope binding. λ-FLOW has a graphical interface where the program can be drawn as a dataflow graph [8].

3 λ-FLOW LANGUAGE

Atoms are the basic expressions of the language. Natural integers and their associated operators, user's defined data and their specific operators and identifiers are atoms.

Users can specify their own **algebra** built upon their datatype and relative operators. The integer algebra is defined by default because some λ-FLOW operators use them. The independence of λ-FLOW with the data gives a great generality to the language. The algebra are implemented according to the used target code. This feature could be a basis of a **co-design** implementation.

Alternative is a choice between two expressions depending on a condition. It is written: `IF cond THEN then ELSE else`, where `cond` must be evaluated as an integer (`0` denotes the false value).

λ-FLOW is a **side-effect free** language due to its functional feature. So, the clauses of the alternative cannot create a side-effect, and their parallelization is possible.

Alternative is the primitive concept of **clocks** of the languages cited in section 2. Of course, it is simpler to understand, and its semantic description is trivial.

Application acts on its arguments according to the operator semantics, given by the algebra. It is written: `OPERATOR (arg-1, ..., arg-n)`. The applications with two arguments can also be written with an infix syntax, such as `arg-1 OPERATOR arg-2`.

Definition allows identifier-value associations. In the current environment (see vector) or in the sub-environments, this name becomes a synonym of the expression: the language is said **referentially transparent**. A definition is written: `name := value`

Stream allows to write in a functional way a **recurrent** equation [4]. A stream has two parts: a state which contains the initial value and a contract for computing the next state 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-FLOW streams are **synchronous**.

The stream construction allows writing a state variable in a functional way. It handles the time in the language : it is the λ-FLOW translation of the Z-delay operator. All the applications which can be designed with a state model can be written with λ-FLOW, that adds some modern language feature, such as polymorphic abstraction, lexical binding, ...

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

```
BEGIN
 actor_1;
 ...
 actor_n;
END
```

A vector must have at least one component. It defines a frame of an **environment** [1, 2]. All the definitions it contains are visible from all inner expressions. Identifiers are statically linked into an environment, as in the language SCHEME [1]. This static linkage allows efficient compilation [16].

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 refer directly or not to a vector. `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 of figure 3, index `x` in the extraction of the main module matches with `x` output of the filter.

Output exports a value for extraction. An output is written: `name ! value`. Notice the outputs of the main module are outputs of the system.

Abstraction abstracts an actor with some parameters. It is written: `lambda p_1, p_2, ..., p_n. actor`.

Parameters `p_i` are identifiers. The parameters of the main module must have a signature, and they are written `parameter:signature`. The parameters of the other abstractions are untyped, that encourages polymorphic abstraction definitions. Of course, the operators inside the abstraction body have to exist for the given arguments.

The **free variables** of an abstraction are linked in a natural way with a lexical scope in the environment where they are defined (such as in language C, and most of the modern languages). This feature allows to share some global features without putting them in the abstraction parameters. The language LUSTRE which is closed to λ-FLOW does not support free-variables in an abstraction [15].

An instantiation of abstraction is written as an application. The instantiation mechanism acts as a macro-replacement, but it takes into account the variables bindings. In addition, it does not allow the definition of inconsistent expression, because recursive instantiations are forbidden.

```
filter := lambda i. BEGIN
 x  ! a + b + c + d;
 a := i - b;
 b := 0 FOLLOWED-BY d;
 c := a - e;
 d := c + e;
 e := 0 FOLLOWED-BY c;
END;

MAIN := lambda i:int. BEGIN
 out  ! filter(i) EXTRACT x;
END;
```

Figure 3: λ-FLOW

The example of λ-FLOW program in figure 3 implements the filter of figure 2. There are two modules, `filter` and `main`. The main module instantiates the `filter` module with the argument `i`, the input of the system.

There is only one main model in a program. The main inputs have a signature with the form `i:int` while the inputs of the other modules are untyped. This feature encourages the definition of polymorph programs. In this example, `filter` does not changes if the number are `real`. Of course, the `real` algebra must be defined and loaded. Thus, it reads the output of the filter with an `X` extraction and writes the result in the main output of the system, named `out`.

The reader could see that the language is **very natural**, and it closed to the Z-formalism generally used. This simplicity is possible because λ-FLOW is built on an accurate abstract language, the λ-matrices.

4 λ-FLOW SEMANTICS

λ-FLOW is the syntactic interface of a functional abstract language, dataflow-based, called **λ-matrices**, and its solving abstract machine [12]. It uses accurate semantics tools, mathematics-based [19, 20]. Four functional solving operators define the abstract machine. This kind of abstract machine cannot be found in languages cited in section 2. This machine emphases the **functional** property of the whole model and allows proofs of results.

```
@1 := + (@2, @8);
@8 := + (@4, @9);
@9 := + (@6, @5);
@7 := 0 followed-by @6;
@6 := - (@2, @7);
@2 := - (@3, @4);
@3 := @in (0);
@4 := 0 followed-by @5;
@5 := + (@6, @7);
@0 := @out (0, @1);
```

Figure 4: Flattened code

Solving an application is a **tail-recursive** equation that defines a functional abstract machine. Each step of the recursive computation corresponds to each instant. This equation can generate an infinite number of steps. This activity is modeled with two functional operators: the *regeneration* operator that regenerates all the streams states of the model in a synchronous way and that returns the new regenerated system, and the *evaluation* operator used by the regeneration operator to evaluates the new stream states. The functional view of the state-variables is due to the regeneration operator.

Because we want **time/memory determinisms**, some systems cannot be solved. So, three criterion functions are defined. A system that contains free variables (unresolved links) is said *unclosed*. A system that contains any fixpoint equation is said *uncalculable* because its computation is time-indeterministic. If the dimension (amount of information used to describe the system) grows with time, the system is said *unstable*. We established in a proved way the relations between the properties and the corresponding criterion functions. We also proved that if a system has some properties at initial time, it keeps them during all the others instants. Solving operators are fully functional, so, the obtained results are proved ones [10].

In addition, the solving process is static due to the stability property. Easy **parallelism** exploitation is permitted by this strong feature.

5 COMPILING λ-FLOW PROGRAMS

```
main() {
 int ad_0, ad_1, ad_2,
     ad_3, ad_4, ad_5,
     ad_6, ad_7, ad_8,
     ad_9;
 start:
 init:
  ad_7 = 0;
  ad_3 = getint(0);
  ad_4 = 0;
 loop:
  ad_2 = ad_3 - ad_4;
  ad_6 = ad_2 - ad_7;
  ad_5 = ad_6 + ad_7;
  ad_9 = ad_6 + ad_5;
  ad_8 = ad_4 + ad_9;
  ad_1 = ad_2 + ad_8;
  ad_0 = putint(0,ad_1);
 next:
  ad_7 = ad_6;
  ad_3 = getint(0);
  ad_4 = ad_5;
  goto loop;
}
```

Figure 5: C code

The first step of the compilation process is to convert the syntactic form in the λ-matrices abstract language [12]. This is accomplished with a classical **parser** [2]. The second step is to instantiate each abstraction where it is used. This **instantiation** acts as a syntactic macro-replacement, but it takes into account the variables bindings (unlike the c preprocessor). When a program is instantiated, the only remained abstraction is the `main` one. It defines the inputs and the outputs of the system.

Then, the instantiated program is **checked**. After a strong type checking, the program is analyzed in order to reject the time/memory indeterministic ones. This analysis is performed according to the formal methods we have described [12, 10].

After the semantics analysis of the programs, the main module is **flattened**. This is performed with a formal and functional method that is shown to keep the semantics of the original program. The resulting flattened abstract program from the example in figure 3 is shown in figure **??**; it is a subset of the λ-matrices. This code is functional, but it looks like the classical three addresses code of the internal forms in most of the compilers [2].

```
(let* ()
 (let loop ((@7 0)
            (@3 (getint 0))
            (@4 0))
  (let* ((@2 (- @3 @4))
         (@6 (- @2 @7))
         (@5 (+ @6 @7))
         (@9 (+ @6 @5))
         (@8 (+ @4 @9))
         (@1 (+ @2 @8))
         (@0 (putint 0 @1)))
    (loop @6 (getint 0) @5))))
```

Figure 6: Scheme code

This program uses a subset of the λ-matrices abstract language. It doe not contain any module and the extractions are replaced with the extracted expression.

Because the flattened code is simple, the **code production** is easy to implement. The compiler does not define statically the **target code**: a description of the target is dynamically loaded at compile-time. A target description is a short file that contains the specific part in the code production.

The structure of the code clearly appears in the example in figure 5: the `start` part contains the code that gives the initial value of the streams, the `init` part initializes the stream states and the inputs values, the `loop` part computes some temporary variables, and finally, the `next` puts the next values of the streams and the new inputs values.

This organization is the same whatever the considered target. With another set of definitions, the compiler produces a purely functional SCHEME code, such as in figure 6.

The nature of the produced code is static, and it is time/memory deterministic. In addition, the use of formal method in the most important transformation phase of the compilation process guarantees the resulting program.

6 PARALLELISM

It is obvious that the resulting code produced by the compilation phase of a λ-FLOW program is **static**. The memory accesses are known at compile time, and they are invariant. The variables are once assigned to a value inside the main loop: before this assignment, they are not used, and after, they are not changed.

All these properties allow to design a static **parallel** architecture. In this parallel architecture, all the processors are directly connected to a main bus, itself connected to the main memory. When a processor does not access to the bus, it is placed in high impedance (unconnected).

The **scheduling** of the tasks uses traditional methods [7]. The gain of the model is in the way how the tasks are organized around a main loop, with single assignment. We have built a software simulator of this architecture [9], and designed the parallelizer compiler principles.

7 FUTURE WORK

In order to validate the whole CAD tool chain, we are implementing the **G726 norm**: it implements an adaptative differential pulse code modulation (ADPCM) filter for digital transmission systems. We are expecting to produce an efficient C program in an automatic way and to parallelize this implementation.

The **CAD tool chain** is programmed with a powerful SCHEME based language that allows fast prototyping. We plan to rewrite this chain with language C in order to get a portable and faster tools. In addition, the graphical interface will be rewritten with the graphical tool kit MOTIF.

8 CONCLUSION

In this short article, we have presented a language designed for implementing DSP applications in parallel architecture.

The language does not depend on the handled data. This feature provides a great generality to the language, without a complicated type construction. The language is very simple, and it is closed to the Z-formalism.

It is built with an accurate semantic language that allows both proofs of programs and proofs of transformations, due to its full functional feature. The time/memory determinisms are checked at compile time.

Due to the determinism of the programs, they can be flattened in an intermediate form. Because the resulting program is very simple, the code production phase is easy to implement.

The target code is not statically defined in the compiler. It is dynamically loaded in a short description file. A new target description file is easy to define.

So the λ-FLOW language is very useful because it is independent with the data algebra and with the target code, and is uses all the modern language concepts.

λ-FLOW is more primitive than the main FSD languages. But it seems to be as expressive as them, and simpler to the DSP designer, because it is closed to the Z-formalism.

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] M. Amamiya and R. Hasegawa. Dataflow computing and eager and lazy evaluations. *New Generation Computing*, 2(2):105–129, 1984.

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

[5] 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.

[6] A. Church. *The Calculi of Lambda-conversion.* Annals of Mathematical Studies, vol. 6, Princeton University Press, Princeton (N.J.), 1951.

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

[8] G. de Wailly. A graphical interface for the functional synchronous data-flow language λ-flow. In *Dynamic Object Workshop (DOW'96).* Object World East, may 1996.

[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. A cad tool chain for signal processing applications, with parallel implementation issues. In *Groningen Information Technology Conference*, february 1996.

[12] 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.

[13] 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.

[14] T. Gautier, P. le Guernic, and L. Besnard. SIGNAL: A declarative language for synchronous programming of real-time systems. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 257–277. Springer-Verlag, Berlin, DE, 1987.

[15] 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.

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

[17] R.M. Karp and R.E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal of Applied Mathematics*, 14(6):1390–1411, 1966.

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

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

[20] B. Meyer. *Introduction à la théorie des langages de programmation.* Inter Edition, 1992.

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