A GRAPHICAL INTERFACE FOR THE FUNCTIONAL SYNCHRONOUS DATAFLOW LANGUAGE λ-FLOW

Guilhem de Wailly

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@unice.fr

Abstract

In this paper a graphical interface for a functional synchronous data-flow language is described.

The intent of this work is to show how a dynamic object-oriented language can be used to model complex and realistic "things" in a very short time.

The object paradigm allows a homogeneous view of objects with common methods. An inheritance mechanism significantly reduces the code because of the feature sharing.

Here, the classical database object-model shows the relationships between object instances. In addition, the inheritance-graph describes the inner construct of objects. The program is specified with smalltalk.

The graphical toolkit is simulated with a root class, so the code can be adapted to numerous object oriented languages, because the graphical aspects are not a part of the model. Particularly, in this paper, we use GNU smalltalk without any graphical possibilities. In our final version of the graphical interface, we use the Stk language that offers a graphical interface with tk, and an object oriented layer with `Clos`.

1 Motivations

We have designed in our laboratory a CAD tool chain which allows signal processing applications to be implemented onto a specific parallel architecture [11]. The structure of the chain is shown in figure 1.

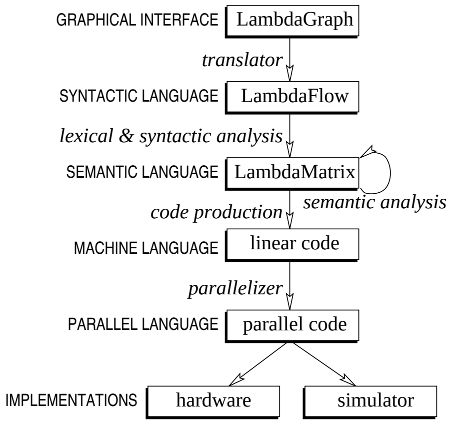


Figure 1: *A CAD tool chain for parallel implementation of signal processing.*

The heart of this tool chain is a semantics language named λ-matrix [12] defined with both the functional [1, 3] and the data-flow [2, 4, 19] paradigms.

The use of a semantics methodology such as the denotational semantics [20] allows proving time and memory determinisms of the programs[9].

The data-flow semantics is expected to support the inner parallelism of the applications. In addition, it allows a graphical representation of the programs. Due to the static solving method of our abstract language, the parallelism can be easily exploited. Moreover, the parallel architecture is expected to be very simple and cheap.

Last year, we presented a paper at the LUV'95 Conference which dealt with the simulator of the parallel architecture [8]. This paper is about the graphical language named λ-graph, on the top of the chain. This language is a graphical interface for a syntactic language built upon the λ-matrices. As of this writing, λ-graph is fully operational [7]. It is written with the powerful Stk [14] language that offers a TCL/TK [23] interface to SCHEME [5, 1], and an Object Oriented layer based on Clos, the Common Lisp Object System [17].

The object oriented paradigm, and more precisely Stk, allows writing complete and complex programs in a very easy way. For example, λ-graph is written with less than 4000 lines of code, a dream for a programmer!

In this paper we present the preliminary analysis done before the Stk implementation of λ-graph. This analysis uses GNU SMALLTALK as a specification language. In section (§ 2), we informally present the syntactic language λ-FLOW. Then we start the analysis in section (§ 3) with the screen model, the object model and the inheritance graph of the application. The SMALLTALK code of the main functions is in section (§ 4). Finally, we discuss the graphical editor λ-graph itself in section (§ 5).

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

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

Figure 2: *Second order recursive filter with* λ-FLOW

2 The λ-flow programming language

The Functional Synchronous Data-flow language λ-FLOW is based on an abstract language defined with a sound semantics. It contains only a few objects to stay as simple as possible and to support modularity. In addition, the language is independent of the handled data: the data and the corresponding operators are defined into some user's algebra. The integer algebra is predefined in the language.

The aim of this paper is to define a graphical interface for this syntactic language. An example of a program can be viewed in figure 2, a simple second order recursive filter.

In this example there are two modules, `filter` and `main`. The main module instantiates the `filter` module with the argument `i`, the input of the system, and it names this instantiation `instance`. The main inputs have a signature with the form `i#int` while the inputs of the other modules are untyped. Thus, it reads the output of the filter with an extraction and writes it in its own `output`.

2.1 Expressions

In λ-graph, atoms (`0`) and application operators (`+`) are relative to an algebra: they are dyamically added to the lexical units of the language, with a compiler command line option. So, the same λ-FLOW program can have different behaviors, according to the algebra used. λ-FLOW defines:

Atoms are the basic expressions of the language. Natural integers and their associated operators, user's defined data and their specific operators, identifiers, and 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 defined by default because some λ-FLOW operators use them.

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

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

λ-FLOW use integer as booleans: 0 denotes **false** while the others integers denotes **true**. λ-FLOW is a side-effect free languagen due to its functional feature. So, the clauses of the alternatives cannot create a side-effect, and their parallelization is possible (the parallelization of the alternative is a big problem that LEE has with the synchronous dataflow [19, 18]).

Application acts as a filter of its arguments according to its 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 infixe syntaxe, 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 “referencially transparent”. A definition is written:

```
name := value
```

Stream allows to write in a functional way a recurrent equation [2]. 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-FLOW streams are synchronous.

The stream construction allows writing a “state variable” in a functional way. It is the operator that 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 all the modern languages features.

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 have at least one component. It defines a frame of an **environment** [1]. 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]. 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 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 in figure 2, 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)
```

The module instanciation acts as a macro-replacement, but it takes into account the variables bindings.

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 asignature, and they are written `param#sign`. The actor can be a vector or another expression, and the inside variables are bound with a lexical scope à la SCHEME.

2.2 Semantics properties

λ-FLOW is the syntactic interface to an abstract language, dataflow-based, called λ-matrice, and its solving abstract machine [12, 10]. It uses accurate semantics tools, mathematics-based [20, 21]. Several functional solving operators define the abstract machine. This kind of abstract machine cannot be found in the other available dataflow languages. This machine emphases the functional property of the whole model and allows proofs of results.

In fact, this language gives a full functional view of state variables that are modeled with dataflow-based *streams*. The stream semantics is closed to one introduced by the Z^{-1} delay operator. So, a non-specialist can easily write a λ-matrix.

In addition, the language supports a modularized design of applications. Each module can be conceived separately, such as a component. When a module is defined, it can be used such as basic operators, with no-distinction. This feature emphases the reuse of programs.

λ-matrices are built independently to the user-algebra: They are defined as a "thing" for handling "things". One can use them for integer-based operations, while another for floating-point operations, and a third one for bottles in a factory, according to the user-algebra that defines data and corresponding operators. Several algebra can be mixed. So, they are not closed to predefined data-type, such as cited languages, that enhances the generality of the language.

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 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 and 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 fix-point 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.

In addition, the solving process is static due to the stability property. Easy parallelism exploitation is permitted by this strong feature. We have proposed a specific parallel architecture designed for λ-matrices implementations [8]. In this architecture, all controllers are directly connected to a main bus, itself connected to a common memory with three-state chips. This architecture can be built with common controllers, for cheap implementations, or with specialized DSP chips.

Access conflicts are solved at compile-time, due to the static feature of the solving process. The parallelism exploitation uses classical list-scheduling methods [6]. In order to increase bus-sharing performances, we have defined a particular task interleaving that allows a task to begin before all its predecessors are ended. Unfortunately, this enhancement cannot be calculated in a theoretical way [6] because tasks cannot be interleaved in the worse case. An architecture simulator has been built for fast testing [8]. Our parallel architecture is one way to implement λ-matrices, the other possibility is to use specialized chips, such as ASICS.

The features of the modeling tool are:

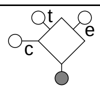
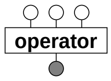
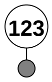
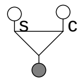
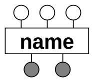
- Especially designed for signal processing, that allows a direct translation of Z-equations.
- Full support of modularization with lexical variable bindings.
- User-algebra independent.
- Sound mathematics-based semantics, with full functional feature.
- Functional abstract machine that solves the model.
- Static solving that allows parallelism architecture and specialized chips to be designed.
- A cheap parallel architecture is proposed, as well as its simulator.

3 Analysis

The analysis of this section deals with the graphical interface layout, the object model and the inheritance graph.

3.1 Layout of the interface

Each actor shown in section (§ 2.1) has a graphical representation in λ-graph:

	an **alternative** has tree entries: a condition, a then clause and an else clause. It has also one output.
	an **application** has several entries, depending to the operator, and it has only one output.
	an **atom** has only one output that gives a constant value. The value is algebra dependent.
	a **stream** has two inputs, a state (or initial value) and a contract. It has only one output.
	an **input port** is one input of the edited module. This object is the graphical representation of a parameter in an abstraction.
	an **output** port is one output of the edited module. This object is the graphical representation of an output object.
	a **module** is a composition of several actors. It can have several inputs and several outputs and it looks like an application.

The graphical elements above are from the grammar of λ-FLOW. This graphical grammar is used by most of the signal processing applications designers when they draw a filter with some lines, some operators and the delay operator used to handle the time. The constructor object is the module that allows building composed objects.

Notice that the extraction object is not directly expressed in λ-graph. In fact, an extraction occurs when one of the outputs of a module is linked.

When a module is used in a program diagram it is illustrated as in the box above. When edited, the module representation becomes a window of the λ-graph program as shown in figure 3.

The window is organized in three parts:

- a menu bar that allows to process some actions such as file saving/retrieving/printing, copy/cut/past operations or compiling according to a target language;
- a command panel for creating actors;
- a canvas area for drawing the content of the current module. By default, the current module is named '`unnamed.lg`'.

An actor in the canvas area has several inputs/outputs colored in differents ways. An input can be linked to only one output. An output can be linked to several inputs. A handle is up linked to either one output or one handle, and it is down linked to several inputs / handles.

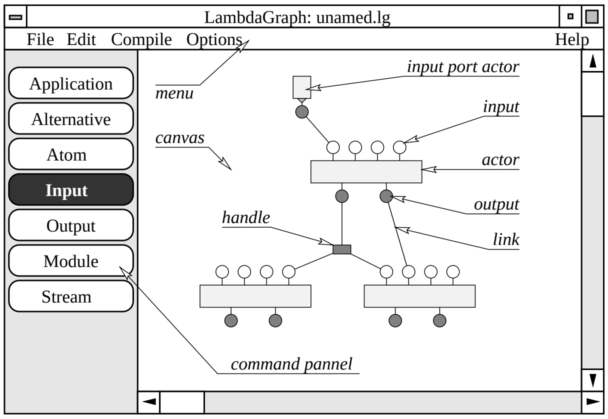


Figure 3: *Screen layout of λ-graph.*

Each object (actor or handle) can be moved in the canvas area, and all the links it is attached follow its moving. If an object is destroyed, all the attached links are destroyed too.

The canvas area supports copy/cut/paste operations. In addition, several objects can be selected. In this case, all the operations concern the selected objects.

A program can be saved in a file, and loaded from a file. In addition, a used module can be edited in a new window. Any change in the interface of an edited module (input/output ports) notifies each window that uses this module.

Atoms and applications are configured in a dialog box according to the available algebra. An algebra exports some operators defined with a name and a signature, and a data type defined with a regular expression used to identify the data. Modules can be loaded/saved into some libraries.

3.2 Object model

The main purpose of this section is to introduced the required objects, their attributes, and the relations between these objects. The method used is the classical object-model of the database analysis [22], as shown in figure 4.

An actor has zero, one or more inputs, and zero, one or more outputs. An input is connected to zero or only one link, and an output is connected to zero, one or more links. So a link is connected to zero or one input and zero, one or more outputs. A handle is always connected to an upLink, and a link has zero, one or two handles. Notice a link has either an upOutput or an `upHandle`, and is has either a `downInput` or a `downHandle`.

From this object model of the application, the inheritance graph can be directly deduced, as shown in the next section.

3.3 Inheritance graph

This section deals with the classes of λ-graph, deduced from the object-model. The object model focuses on relationships between object instances. The inheritance graph

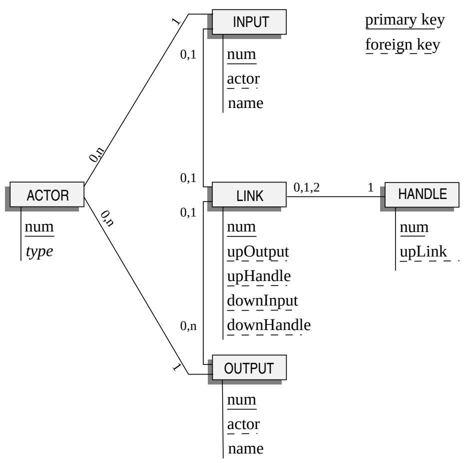


Figure 4: *Object model of λ-graphshowing the relationships between the object instances.*

shows how objects are built by specialization of more generic objects, as shown in figure 5.

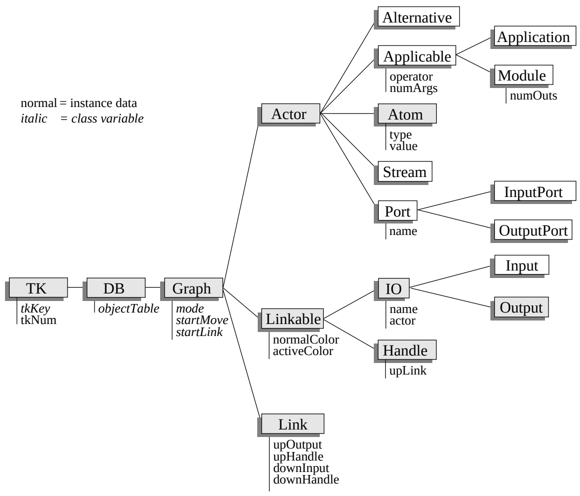


Figure 5: *Object inheritance graph of λ-graphshowing the object classes. Grayed classes are explained in this paper.*

The TK class represents the graphical toolkit with an interface. Basically, this toolkit allows to creates some graphical objects such as lines, box, circles, and to handles them with their key. In fact, it is mapped on the canvas widget of STK. This class exports some graphical methods. In this specification, the toolkit is simulated. The class defines the class variable `tkKey` that is the number of created graphical objects, and the instance variable `tkNum` that is the number of graphical objects that inherit from this class.

DB is a database. There is one database for each instance window of λ-graph[1]. This database is a set that contains all the actors, inputs, outputs, handles and links of the application. It allows to perform selections on these objects with criterion functions. Currently this method is inefficient and it is not directly implemented in the final version of λ-graph, but it is simple.

The Graph object is the main application class. It contains some global variables (class variables) such as the current mode of the command panel.

Then, we have three groups of objects: the actors, the linkable and the links. The actor class is always specialized. The linkable class is the root class of the inputs, outputs and handles, and the link class is directly used.

4 Implementations

This section gives the coding specification of λ-graph. It uses the GNU version of SMALLTALK. In the first part, the classes definitions are given, and in the second part, the methods are programmed.

4.1 Classes definitions

4.1.1 Roots classes

All the classes of λ-graph inherit from these root classes. The first root class is TK that offers an interface with the graphical toolkit. The TK class defines one instance variable `tkNum` for each graphical object that is created in the toolkit. In addition, the class defines a class variable that is a counter of `tkNum`. Notice that a class variable is the same for each instantiated object that inherit from this class.

◇ **TK**

```
Object subclass: #TK
   instanceVariableNames: 'tkNum'
   classVariableNames:    'tkKey'
   poolDictionaries:      ''
   category:              nil
!
```

The DB class declares a class variable named `objectTable`. This variable is shared by all the instantiated objects that inherit from DB. It is a way to define a global variable for an application.

◇ **DB**

```
TK subclass: #DB
   instanceVariableNames: ''
   classVariableNames:    'objectTable'
   poolDictionaries:      ''
   category:              nil
!
```

The Graph class is the application class of λ-graph. It defines some class variables such as `mode` (the current mode of the command panel), `startLink` (the starting Linkable object of a link operation) and `startMove` (the starting location of a move operation).

◇ **Graph**

```
DB subclass: #Graph
   instanceVariableNames: ''
   classVariableNames:    'mode startMove startLink'
   poolDictionaries:      ''
   category:              nil
!
```

4.1.2 Actor

The Actor class is the root class of all the actors. It defines no variables and it used only for sharing methods.

[1] In order to keep the code simple, this database is defined as a class variable (objectTable). This means that each object that inherit from this class has the same common objectTable variable.

◇ **Actor**

```
Graph subclass: #Actor
   instanceVariableNames: ''
   classVariableNames:    ''
   poolDictionaries:      ''
   category:              nil
!
```

The Atom class inherits from the Actor class. It defines `type` and `value` instance variables. For example, the value of the integer atom in section 3.1 is the string `123`.

◇ **Atom**

```
Actor subclass: #Atome
   instanceVariableNames: 'type value'
   classVariableNames:    ''
   poolDictionaries:      ''
   category:              nil
!
```

4.1.3 Linkable

Linkable objects are inputs and outputs of the actors and the handles. The root class Linkable defines two instance variables `normalColor` and `activeColor`, respectively used when the mouse is not, or is over the object.

◇ **Linkable**

```
Graph subclass: #Linkable
   instanceVariableNames: 'normalColor activeColor'
   classVariableNames:    ''
   poolDictionaries:      ''
   category:              nil
!
```

The IO class is the root class of the inputs and output objects. Because these objects belong to only one actor, the class defines the `actor` instance class. In addition, it is useful to name them with the `name` variable.

◇ **IO**

```
Linkable subclass: #IO
   instanceVariableNames: 'actor name'
   classVariableNames:    ''
   poolDictionaries:      ''
   category:              nil
!
```

◇ **Input**

```
IO subclass: #Input
   instanceVariableNames: ''
   classVariableNames:    ''
   poolDictionaries:      ''
   category:              nil
!
```

A handle is always attached to one `upLink`.

◇ **Handle**

```
Linkable subclass: #Handle
   instanceVariableNames: 'upLink'
   classVariableNames:    ''
   poolDictionaries:      ''
   category:              nil
!
```

A Link object can have either an `upOutput` or an `upHandle`, and either a `downInput` or a `downHandle`.

◇ **Link**

```
Graph subclass: #Link
   instanceVariableNames: 'upOutput upHandle downInput downHandle'
   classVariableNames:    ''
   poolDictionaries:      ''
   category:              nil
!
```

4.2 Methods

4.2.1 Creation methods

The creation methods are class methods. That means they are applied to the class objects. They are used for the creation (instantiation) of the objects by calling the standard SMALLTALK `new` method.

The following method is used for Actor creation with the parameter `loc`, which is the Location in the canvas area where the object has to be drawn. The local variable `act` is instantiated with the `new` method. Then the class method `add` (defined for DB) is invoked: this call adds the new actor into the database. Then the instance method `init` is invoked. Finally, the graphical object is created into the toolkit at the `loc` location with the method `tkCreate`

◇ **Actor.create**

```
!Actor class methodsFor: 'creation'!
create: loc                  "this is a Smalltalk comment"
   | act |             "local variable"
   act _ self new.     "affectation"
   Actor add: act.     "class method invocation"
   act init.           "object instance method invocation"
   act tkCreate: loc.
   ^act                "returned value"
!!
```

. The creation of the IO and Handle object are similar to the previous method.

A Link object is created with more parameters: input and output linkable. These parameters are used by the `init` method.

◇ **Link.create**

```
!Link class methodsFor: 'creation'!
create: upHandle upOutput: upo
               downHandle: dnh downInput:  dni

   | lnk |
   lnk _ self new.
   Link add: lnk.
   lnk       init: upHandle
         upOutput: upo
       downHandle: dnh
        downInput: dni.
   lnk tkCreate.
   ^lnk
!!
```

4.2.2 Initialization methods

The initialization methods are called by the `create` class method of each object, when the user creates an object. These methods return the object that is initialized, such as in the Actor initialization.

◇ **Actor.init**

```
!Actor methodsFor: 'initialization'!
init
   ^self
!!
```

The initialization of an Atom object is slightly more complicated. It asks to the user the type and the value of the created Atom, via two dialog boxes from the TK toolkit.

◇ **Atom.init**

```
!Atome methodsFor: 'initialization'!
init
   type   _ TK
         ask: 'type'
        type: String
     control: [:read| read size > 0]
       error: 'unexpected empty string'.
   value _ TK
         ask: 'value'
```

```
      type: String
   control: [:read| read size > 0]
     error: 'unexpected empty string'.
  ^self
!!
```

A handle is always created in order to cut a link in two parts. So, the initialization method has the link to be cut as parameter.

◇ **Handle.init**

```
!Handle methodsFor: 'initialization'!
init: downLink
   "creates the new upLink"
   upLink _
     Link    create: (downLink upHandle)
           upOutput: (downLink upOutput)
         downHandle: self
          downInput: nil.

   "updates the downLink"
   downLink upHandle: self.
   downLink upOutput:  nil.

   activeColor  _ 'red'.
   normalColor  _ 'green'.
   ^self
!!
```

A Link is always created and initialized with either an `upOutput` or an `upHandle`, and either a `downInput` or a `downHandle` as parameter.

◇ **Link.init**

```
!Link methodsFor: 'initialization'!
init:            uph
       upOutput: upo
     downHandle: dnh
      downInput: dni

   upHandle   _ uph.
   downHandle _ dnh.
   upOutput   _ upo.
   downInput  _ dni.
   ^self
!!
```

4.2.3 Selecting/changing methods

The following methods are selectors for the instantiated objects. The Actor selector methods return the set of its inputs (`inputs`) or the set of its outputs (`outputs`).

◇ **Actor.select**

```
!Actor methodsFor: 'selector'!
inputs
   ^Input select:
      [                    "defines a procedure..."
       :in|                "... with one parameter..."
       self = (in actor)   "... that performs a test"
      ]
!
outputs
   ^Output select: [:out| self = (out actor)]
!!
```

The `actor` selector of the IO class returns the `actor` instance variable. In addition, the `links` selector method returns the set of the links attached to this object with a selection into the database.

◇ **IO.select**

```
!IO methodsFor: 'selector'!
actor
   ^actor
!
name
   ^name
!
links
   (self isKindOf: Input) ifTrue: [
     ^Link select: [ :lnk| self = lnk downInput]
   ]
   ifFalse: [
     ^Link select: [ :lnk| self = lnk upOutput]
   ]
!!
```

The `upLink` selector of the handle object returns the instance variable `upLink` while the `downLinks` method returns the set of links "down attached" to this handle, with a selection into the database.

◇ **Handle.select**

```
!Handle methodsFor: 'selection'!
upLink
   ^upLink
!
downLink
   ^Link select: [ :lnk | self = lnk upHandle]
!!
```

The first four selector methods of a Link return the corresponding instance variables. The `isValid` selector method returns a boolean which is true is this link is in the database, and false otherwise. This strategy is used by the destroying method in order to avoid the recursion on a link deletion.

◇ **Link.select**

```
!Link methodsFor: 'selector'!
downInput
   ^downInput
!
downHandle
   ^downHandle
!
upOutput
   ^upOutput
!
upHandle
   ^upHandle
!
isValid
   ^1 = ((Link select:[:lnk | self = lnk]) size)
!!
```

The changing methods of a Link object alter its instance variables.

◇ **Link.change**

```
!Link methodsFor: 'changing'!
downInput: dni
   downInput _ dni.
   ^self
!
downHandle: dnh
   downHandle _ dnh.
   ^self
!
upOutput: upo
   upOutput _ upo.
   ^self
!
upHandle: uph
   upHandle _ uph.
   ^self
!!
```

4.2.4 Graphic creation methods

The graphic creation methods create the objects which can be viewed by the user in the screen. They call the TK interface methods. The Actor `tkCreate` methods create the inputs and the outputs of the actor. For example, the Atom `tkCreate` creates the corresponding objects, and it creates the output object.

◇ **Atom.tkCreate**

```
!Atome methodsFor: 'graphical drawing'!
tkCreate: loc
   tkNum _ TK tkKey.
   TK draw: 'atom(type, value)' key: tkNum.
   Output create: (loc ox: 50 oy: 100)
           actor: self
            name: 'output'.
   ^self
!!
```

4.2.5 Moving methods

The next methods are defined in order to move some objects in the canvas area of λ-graph, in a new Location `loc`. Before moving an Actor object, the following method moves all its inputs/outputs. Then the toolkit is appealed to move the graphical object.

◇ **Actor.move**

```
!Actor methodsFor: 'moving'!
move: loc
   self inputs  do: [: in| in  move: loc].
   self outputs do: [:out| out move: loc].
   TK move: self
!!
```

When a IO is moved, all the attached links has to moved too before the graphical object in the toolkit is moved.

◇ **IO.move**

```
!IO methodsFor: 'moving'!
move: loc
   | links |

   "list of links attached to self"
   links _ self links.

   (self isKindOf: Input) ifTrue: [
     links do: [:lnk| lnk downMove: loc]
   ] ifFalse: [
     links do: [:lnk| lnk upMove:   loc]
   ].
   TK move: self
!!
```

When a Handle is moved, its `upLink` is first moved, then all its `downLinks` are moved one by one.

◇ **Handle.move**

```
!Handle methodsFor: 'moving'!
move: loc
   | downLink |

   "moves the upLink"
   upLink downMove: loc.

   "moves the downLinks"
   self downLink do: [:lnk| lnk upMove: loc].

   "moves the graphical object"
   TK move: self
!!
```

When a Link is moved, it is simply redrawn.

◇ **Link.move**

```
!Link methodsFor: 'moving'!
upMove: loc
   TK move: self
!
downMove: loc
   TK move: self
!!
```

4.2.6 Linking methods

The attaching methods create links between inputs and either outputs or handles. These methods are bound to the mouse events in the TK toolkit. In addition, they use the class variable `startLink` of the Graph class. The conditions required for linking are: inputs can have only one link to either an output or a handle, and outputs and handles can have several links. The `pair:` method of the Linkable object determines if the parameter `pair` can be linked to `self`. The `attach` method gives always an input as parameter.

◇ **Linkable.attach**

```
!Linkable methodsFor: 'ask'!
pair: pair
   (self isKindOf: Input) ifTrue: [
     ^   (pair isKindOf: Output)
     or: (pair isKindOf: Handle)
   ] ifFalse: [
     ^(pair isKindOf: Input)
   ]
!
attach: pair
   (self pair: pair) ifTrue: [
     (self isKindOf: Input) ifFalse: [
        self attach: pair
     ] ifTrue: [
       pair attach: self
     ]
   ]
!!
```

The attaching methods of the Handle and the Output objects are very similar. They checks if the Input parameter `in` is not already attached and then, they create a Link object with the good arguments.

◇ **Handle.attach**

```
!Handle methodsFor: 'attaching'!
attach: in
   "checks if in is not already attached"
   (0 = (in links) size) ifTrue: [
     Link  create: self  upOutput: nil
       downHandle: nil  downInput: in
   ]
!!
```

◇ **Output.attach**

```
!Output methodsFor: 'link'!
attach: in
   "checks if the input is not already attached"
   (0 = (in links) size) ifTrue: [
     Link  create: nil  upOutput: self
       downHandle: nil downInput: in
   ]
!!
```

4.2.7 Destroying methods

When an Actor is destroyed, all its inputs / outputs are first destroyed. Then, the graphical object is removed from the screen and the actor is removed from the database.

◇ **Actor.destroy**

```
!Actor methodsFor: 'destroying'!
destroy
   "destroyes all the input/outputs"
   self  inputs do: [: in|  in destroy].
   self outputs do: [:out| out destroy].

   "destroyes the graphical object"
   TK destroy: self.

   "removes self from the database"
   Actor destroy: self
!!
```

When an Input is destroyed, all the link attached to it are destroyed. Then, the graphical object is destroyed, and finally, the object is removed from the database.

◇ **IO.destroy**

```
!Input methodsFor: 'destroying'!
destroy
   "destroyes the downLink"
   self links do: [:lnk| lnk destroy].

   "destroyes the graphical object"
   TK destroy: self.

   "removes self from the database"
   Input destroy: self
!!
```

The destruction of a Handle is slightly more difficult. If the handle simply cuts a link, it is remover and the two links are joined. If the handle has several `downLinks`, each of them are destroyed one by one.

◇ **Handle.destroy**

```
!Handle methodsFor: 'destroying'!
destroy
   | downLinks upLinkValide |

   "asks to the dtabase is the upLink is alive"
   upLinkValide _ upLink isValide.

   "list of the downLinks"
   downLinks _ self downLink.

   "two cases according to the number of downLink(s)"
   (1 = downLinks size)
     ifTrue: [   "is only one downLink"
        downLinks do: [:theDownLink|
           upLinkValide
            ifTrue: [
              "updates the upLink and redraws it"
              upLink  downInput:
                 (theDownLink downInput).
              upLink downHandle:
                 (theDownLink downHandle).
              TK redraw: upLink
            ].
            "detroyes the downLink"
            theDownLink destroy.
        ]
     ]
     ifFalse: [ "if several downLinks"
         upLinkValide ifTrue: [
           upLink destroy
         ].
         downLinks do: [:oneDownLink|
           "destroys all the links"
           oneDownLink destroy
       ]
     ].
   "destroyes the graphical object"
   TK destroy: self.

   "removes self from the database"
   Handle destroy: self
!!
```

When a Link is destroyed, it first removes itself from the database, in order to avoid recursive deletion. Then it suppresses the `upHandle` and the `downHandle`.

◇ **Link.destroy**

```
!Link methodsFor: 'destroying'!
destroy
   | downLink upLink |

   "suppresses the graphical object"
   TK destroy: self.

   "removes self from the database"
   Link destroy: self.

   "supresses the upHandle, if it exists"
   (upHandle isNil) ifFalse: [
     downLink _ upHandle downLink.
     (0 = downLink size) ifTrue: [
       downLink do: [ :lnk | lnk destroy ]
     ]
   ].
   "supresses the downHandle, if it exists"
   (downHandle isNil) ifFalse: [
      downHandle destroy]
!!
```

5 λ-graph

The graphical editor λ-graph [7] allows graphical programming in the **data flow** style. The semantics used are functional synchronous data flow (FSDF), based on an algebraic abstract language. The layout of this editor can be seen in figure 6.

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 6. It is constituted with actors (`-`, `i`, `fby`,...) handles (`A'`) and links (`A`→`A'`). An actor has some inputs (in top) and some outputs (in bottom).

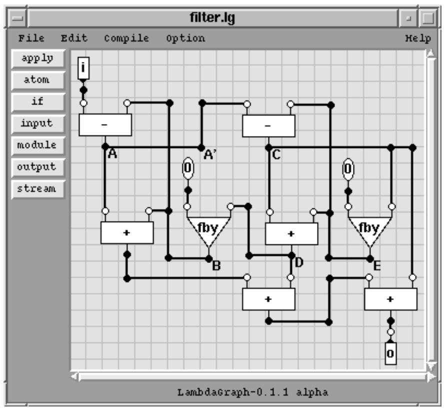


Figure 6: *Graphical editor.*

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 using a mouse 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 **operators** 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.

λ-graph is written with the STK SCHEME language that interfaces the TK library [14, 23].

6 Future work

The first version of λ-graph is unable to run directly an application. It can only call **translators** that transform a program into other forms. Several translations are possible. The author is working in a SCHEME translator that allows direct simulations of programs, with a special library that contains wave-generators and wave-viewers. The main problem with the use of foreign translators is the errors report: it would be helpful for the designer to see in the diagram where an error occur. For the moment, the designer sees an error into the translated form of the diagram, in a separate window, which is, in some case, difficult to read.

A λ-FLOW translator (§ 2) of a λ-graph is being completed. The λ-FLOW compiler produces several target codes: SISAL [13], SIGNAL [4], LUSTRE [15], PTOLEMY with Lee's Synchronous Data-Flow (SDF) [19]. A target code is defined in less than twenty lines of code, and one line per operator of each algebra.

7 Conclusion

In this paper, a practical use of a dynamic object-oriented language is shown. The efficiency of such languages permits to put in the pages of this article the specifications of the problem, a short object-oriented analysis, and the main functions of the program.

A graphical interface to a syntactic language is modeled here. The analysis shows how to use the well known database object-analysis in other kind of applications. This model focuses on the leaf objects (that can be instantiated) and on their inner relationships. From this first step of analysis, the inheritance graph can be deduced: it shows how classes of objects are built.

The coding of this model has two phases: the classes description and their associated methods. The classes descriptions use the inheritance mechanism which deals with a static sharing of features. Method descriptions use the method-specialization mechanism to deals with dynamic sharing of features. These mechanisms greatly increase the reuse of pieces of code. The resulting code is actually short and efficient. Anyone can rapidly modify it. Comments are not needed everywhere because the STK language acts such as a specification language.

λ-graph and the associated tools, such as λ-FLOW and the abstract language of the λ-matrices are out of the preliminary phase of definition. They have been created to allows graphical representations of signal processing applications to be implemented onto a parallel architecture. The CAD tool chain enters into a phase of tests and optimizations.

References

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

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

[3] 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. Reproduced in "Selected Reprints on Dataflow and Reduction Architectures" ed. S. S. Thakkar, IEEE, 1987, pp. 215-243.

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

[5] W. Clinger and J. Rees. Revised^4 report on the algoritmic language scheme. Technical report, MIT Artificial Intelligence Laboratory, CSDTR 174, october 1990.

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

[7] 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.

[8] 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.

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

[10] G. de Wailly and F. Boéri. Abstract language for signal processing modeling with parallel implementations issues. In *VIII European Signal Processing Conference (EUSIPCO'96) - TO APPEAR.* EURASIP, september 1996.

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

[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] GL. Steele JR. *Common Lisp: The Language, 2nd Edition.* Digital Press (Bedford, MA), 1990.

[18] E.A. Lee. Consistency in dataflow graphs. *IEEE Transactions on Parallel and Distributed systems*, 2(2):223–235, April 1991.

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

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

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

[22] S. Miranda. *L'art des bases de données - Tome I : Introduction aux bases de données.* Eyrolles, 1988.

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