Fractalide
HyperCard on Flow-based Programming

Author : Denis Michiels
Collaborator : Stewart Mackenzie

Promoter : Peter Van Roy
Reader : Charles Pecheur

2 June 2014

Thanks

I will never thank Stewart enough. I want to thank him for sharing with me this wonderful idea of making this project, and incorporate me in the development process. Then, for having motivate me during a whole year. Then, for all the discussions we have about Fractalide and the behind concepts. Always ready to heard my ideas and arguments, adding his own ideas, and pushing them all a step further with his personal and IT experience. It was a real pleasure to imagine Fractalide in these conditions. Finally, I want to thank him for his English.

I also want to thanks Peter Van Roy. Each time I discuss with him, I got new ideas. Moreover, he always gives me the right word on my ideas, and the right concept to explore them deeper.

Merci à mes parents de m'avoir permis de faire mes études, donc ce mémoire, et de m'avoir toujours encouragé même dans mes moments d'égarements.

Abstract

Since the dawn of computer programming, developing high-level applications has been a complicated task. The programmer has remained steadfast, the unmovable rock, who's services are required for the implementation of even the most simple of applications. The onset of our multicore world has exacerbated the problem, now even the most simple programs have become hard to run at scale. Language paradigms and concepts were uncovered and implemented thus raising the level of abstraction with the sole goal of simplifying solutions to these complex problems. Fractalide is but one of these approaches, achieved by mixing concepts of two well known and tested technologies, HyperCard and Flow-Based Programming. Fractalide aims to increase the efficiency of program design and construction, yet reduce the cost of software maintenance.

HyperCard was a powerful tool used to develop software. The drag-and-drop interface, with an English inspired scripting language made programming fun and engaging. Even non-programmers were able to efficiently use HyperCard to develop custom applications that scratched their itch.

Flow-Based programming, a paradigm used to easily create and maintain high-level applications had a strong focus on reusability. It ensured the application's business logic became apparent, yet completely abstracted away the building block's implementation. Each block is completely independent and side-effect free, allowing for a weak coupling between blocks and a strong separation of concerns. Programming a Flow-based programming application is much like manipulating a visual graph consisting of vertices and edges.

Fractalide draws the best from these two approaches, combining them is such a way that the drawbacks of one become the strengths of the other.

Contents

Chapter 1

Introduction

Fractalide is an experimental platform for conveniently building component oriented programming applications. Component oriented programming (COP) is a way to design programs with a focus on the separation of concerns. Each component is a single purpose entity which may be combined with other components via composition to achieve more diverse logic applications. The weak coupling that exists between components offers a high degree of re-usability.
More precisely, Fractalide uses Mozart Oz to implement a Flow-Based programming abstraction. Flow-Based programming (FBP) is an implementation of the COP, which in turn is a subset of the Actor model computing paradigm. Designed by J.P. Morrison in the early 1970s, he realized conventional programming languages executed according to the von Neumann model, in that code execution is strictly sequential. This model is well adapted for low-level software, but not for high-level applications. His goal was to build a paradigm that simply focused on the input, data transformation and output. The result was a visual programming language whereby the application's business logic becomes apparent, and the component implementation details are completely abstracted away. In this model components communicate via message passing. Components have one or more input and output ports, data or state enter these inputs via Information Packets (IPs) then gets transformed and sent along their way to whichever component is hooked up to the output port.
FBP applications have a high degree of component re-usability, maintainability, and the application is faster and cheaper to assemble. The main advantage of using such a system is the box-and-line design of the program *is* the actual executing application. There are no multiple development stages, where one might first use a technique like Class-responsibility-collaboration cards (CRC cards) or Unified Modelling Language (UML) to visualize the business logic, then undergo a process of translating the design into lines of codes. This translation, often leaves the design documents far behind the actual implementation, as any alteration to the code is not reflected in the original design documents. Thus they quickly become stale and out of date. Whereas the FBP graph is the actual executed design logic, replacing the need for tools such as UML or CRC cards.

Fractalide presents a FBP abstraction as it's lowest common denominator to its users. The entire environment is implemented with this level of abstraction. Indeed, the very first program we implemented on Fractalide's FBP is a visual editor that allows one to drag-and-drop components, link them together, and run the resulting graph. This work forms part of a proof of concept HyperCard port.

HyperCard, developed in 1987 by Bill Atkinson and Apple Inc., is an application that may be used as a personal organizer, database, an authoring tool, a presentation creator, a hypermedia tool, an application creator. In 2004 Steve Jobs discontinued HyperCard for various unsubstantiated reasons. B.Atkinson made HyperCard to share knowledge about programming with people. It was his intention for non-programmers to make and share programs. As such it was heavily adopted, people used HyperCard in ways one couldn't have imagined, creating things from databases to prototype applications. Essentially, HyperCard allows the user to create cards and put them inside a stack. A simple analogy, a webpage could be likened to a card and a website, to a stack of cards. Another analogy would be your trusty Rolodex filled with cards. On each card, the user had ability to insert data, graphical widgets such as buttons and fields. Data is automatically stored in its built-in database allowing the user to simply explore and modify it accordingly. The user interface was constructed as one would with Visual Basic (this approach was copied from HyperCard), by dragging-and-dropping buttons, fields, lines onto a canvas. Logic was implemented with simple English-like scripting language called HyperTalk, thereby allowing user to make a richer, more complex applications. HyperCard has an intrinsic notion of hierarchy : Data is on cards, cards are in a card stack, which in turn is part of the main stack. HyperTalk was an event driven programming language. The event hierarchy used the same hierarchy as above to propagate events. An event begins at a GUI widget, if not handled, the event escalates to card level, then to the stack level, and finally the main stack. The event may be caught and handled at any level, thus allowing fine-grained control of the events.

HyperCard has a very intuitive and dynamic user interface, the main feature that attracted non-programmers. There are no boiler steps like needing to explicitly compile or launch programs. The user simply runs a stack, and immediately uses the cards as programmed. HyperCard made it dead easy to switch between run mode and develop mode. In develop mode the user edited widgets on the card, then wrote associated scripts. All the user needed to do to switch to run mode was click a button, the changes were automatically saved and compiled in the background. The barrier between running and designing an application was thin indeed. These different modes allowed one to progressively learn HyperCard at your own pace. The new users will not approach areas they were unfamiliar with or didn't need, thus the initial HyperCard experience was simple and became more complex as the user gain courage and knowledge.
HyperCard was the first mainstream hypermedia software, it gave the user the ability to manage large amounts of data. Linking cards or searching the built-in database was trivial. These same features are quite similar to what the world-wide web offers, with an added notion of hierarchy. As every cards is put into a stack, the stack is a common value, so people shared stacks of cards. In contrast the world-wide web has no sense hierarchy (excluding the DNS hierarchy), every page is essentially in the same namespace. When one sees a URI as a unique name and nothing else.

Fractalide tries to take the best of both FBP and HyperCard fuse them together in such a way as to make it simple to share components over the Internet.
In order to do this we need to make a few modifications to both FBP and HyperCard. The first thing we do is replace HyperTalk with FBP, thus lines of code are swapped out for a visual graph. Our motivation for this is to allow everyone, even non-programmers to design applications in a simplistic manner. Indeed it will require a programmer to implement the initial component, but as FBP has a high degree of

re-usability these components may be shared with the rest of the community over the Internet. The second modification is the simplification of HyperCard itself. We have replaced HyperCard's concepts of items, cards, stacks and the main stack with a single concept, that of a card. In essence a card is a FBP component. It reacts to events and may also generate events. Hence everything is a card. HyperCard forbids the creation of new items, or indeed displaying more than one card on a screen. So Fractalide's simplification allows a card to be used as an item, or for the construction of new items, indeed the integration of a complete application into a single card becomes possible. This degree of re-usability is lacking in the original HyperCard and is thus compensated by making use of FBP. Every Fractalide application, indeed Fractalide itself is just a FBP card or component. We've adopted the philosophy used by Emacs, that being Emacs was written in Lisp so Fractalide is constructed with cards.
Thirdly we needed to make our implementation of HyperCard extremely dynamic. This was achieved by making FBP dynamic. Thus a few modifications to FBP was called for. It must be possible to add and remove components, and make new links, without stopping the current execution. Also, to go a step further in the dynamism, the component must allow for Erlang style hot-swapping of their structure and behavior. Also a component should be able to pack itself up and ship itself off to another machine if need be, yet retain its existing links. This degree of dynamism in FBP peculated into our implementation of HyperCard. It is now possible to edit a running program, swap out components of the graph and immediately see the modifications without the application having to be restarted.
Finally, Fractalide aims to support network communication in the near future. Erlang concepts of failure handling should thus be introduced. Each component must be autonomous. There is no scheduler as found in other implementations of FBP. This makes distribution of the components over a network trivial. Fractalide's introduction of Erlang's failure model allows components to generate error streams that don't crash the component, but are handled at run-time. Actions such as restarting components, stopping them, or obtaining statistics becomes possible by means of a supervisor tree similar.
Our long-term goal is to create a kind of "HyperCard for the web". Apple's HyperCard 3.0 planned to integrate cards in the web browser, but was terminated prematurely. Fractalide will pickup the dropped relay baton and take it to the next frontier by making use of Mozart Oz's open Internet failure handling Distributed Sub System. This system will allow people to easily build cards, and then distribute them over the network or indeed share a running-graph over the network like a website, hence introducing the concept of Service Oriented Architecture to Fractalide.

Many of these concepts will be illustrated by following trivial example. The application simply adds two numbers together as shown in figure 1.1. There are two text entry fields allowing the user to input numbers, here 42 and 24. When the user presses the button *calc*, the addition is made and put in the result field. Here we will demonstrate the construction of the user interface and the associated logic.

The user interface is built using a tree; the leaves are GUI widgets such as buttons, fields, etc and the nodes are boxes that display the children in a specific layout : left-right(lr) lays widgets out from left to right, top-down(td) lays widgets out from top to bottom. Figure 1.2 shows the hierarchy of the calculator. The user interface is built by means of composition. Black nodes manage the layout, the blue nodes represent gui widgets. We see the top node has three children. It displays the children in a top-down

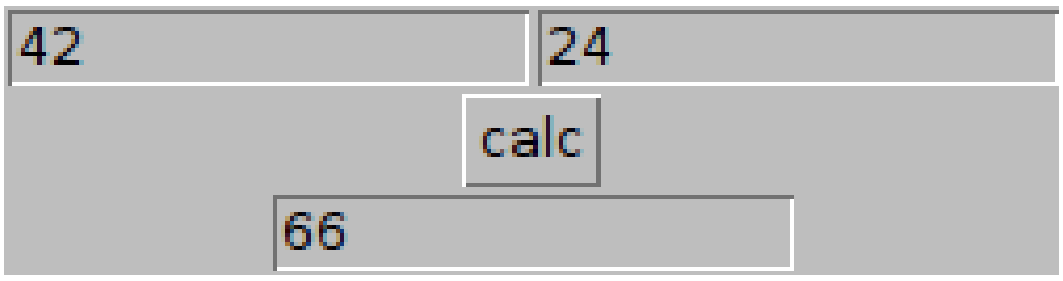


Figure 1.1: The calculator example

fashion. The branch on the left contains a left-right layout widget, the centre branch contains the *calc* button and the branch to the right contains the result text entry. Note the node on the left itself has a sub-tree that contains two children, being two text entry widgets. The logic of this application can be intuitively guessed. The figure

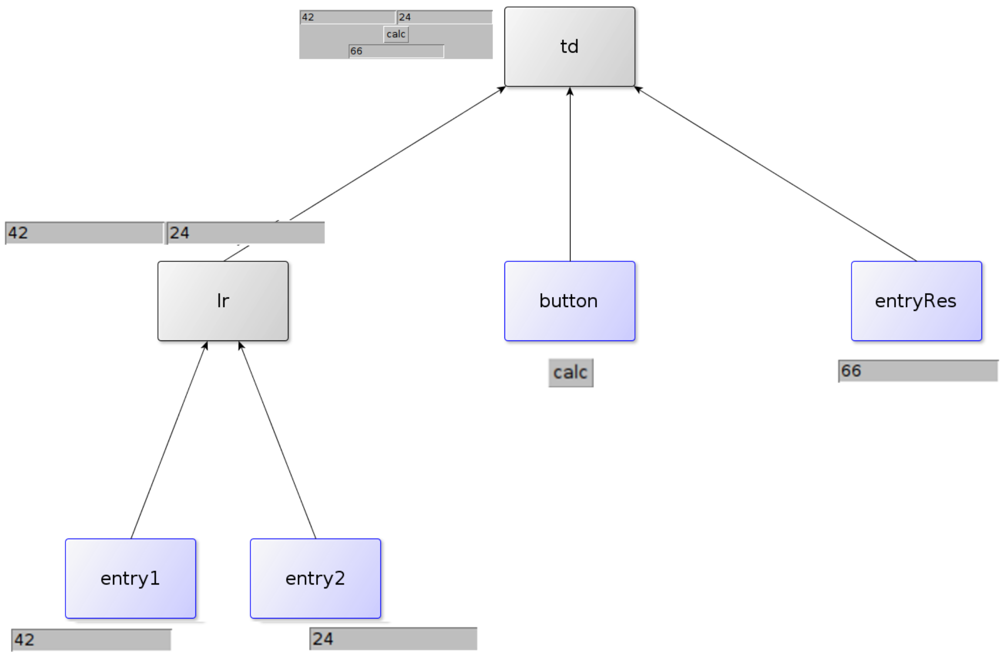


Figure 1.2: The ui hierarchy of the calculator

1.3 shows a graph that represents it. When a button is pressed we'll need to extract the value from each text entry field. Then add the numbers together and display them in the result field. The basic logic of this simple calculator is illustrated with figure 1.4.
Now we need join the vertical plane that represents the User Interface (in black), and the horizontal plane representing logic (in red). The intersection is a set of nodes that play a role in the two graphs. This is nearly a complete Fractalide application. Fractalide is a visual programming tool to build and run graphs like this one. All *UI* nodes are in fact Fractalide cards, that manage different UI events. The *adder* node is a normal component or card. The basic flow of UI events propagate up in the *UI* hierarchy. If an entry generates an event, it will go to the *lr* node, then the *td* node. It's possible to catch an event at any level. If the event is handle, it will halt the propagation preventing further escalation.

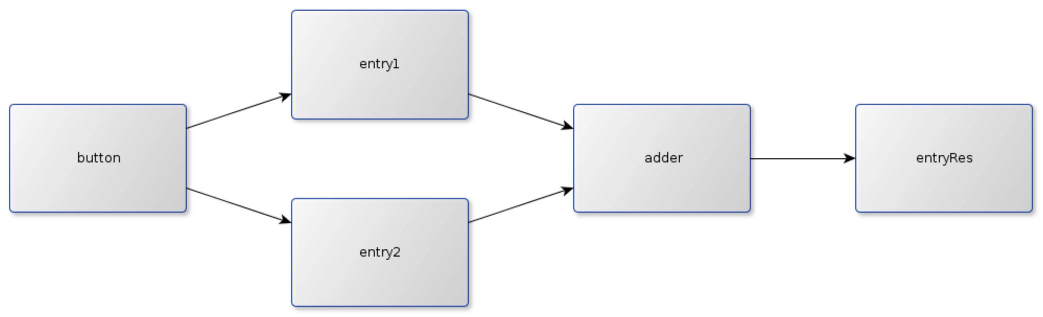


Figure 1.3: The logic graph of the calculator

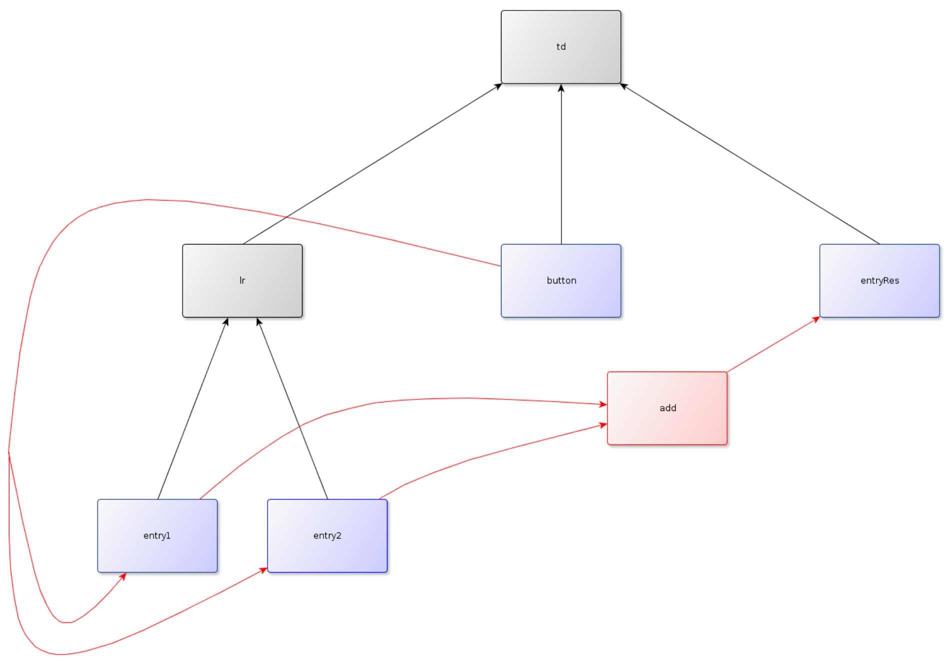


Figure 1.4: The whole graph of the calculator

1.1 Contribution

- Implementation of Flow-Based Programming in Mozart2 Oz. This implementation provide autonomous, dynamic and robust components.
- The *card* concept, that allows to make hierarchical message passing and reusable applications.
- A basic graph editor, build with the Fractalide concept.
- Port the Oz QTk library in Fractalide

Chapter 2

Flow-based programming

This chapter goes over more details of Flow-Based Programming concepts. The first point will be the history, to replace the context of the creation. Then, the general concepts are explained. A rapid sum-up of the behind concept ends the chapter.
The FBP book [7] goes over all that much more deeper. It also gives good habits and good patterns in FBP. His reading is encouraged.

2.1 Introduction

Flow-based programming is a way to design programs invented by Paul Morrison. When Morrison started building programs at the dawn of Computer Science, he found the task quite challenging. Even simple programs were difficult to program and maintain. He and his colleagues discovered the problem was related to the programming paradigm.
The source of the problem was the von Neumann model. This model follows a sequential execution, one instruction at a time with a counter to keep track of position. This model is good for low-level programs, but not for high-level applications. Even today this paradigm is commonly used. The two main limitations are that the application's business logic is hidden in the code and that they are designed for a single processor. Actual programming is still done in files understood only by programmers. Logic is still hidden in thousands of line of codes, sometimes very hard to maintain. Moreover, programs are forced to operate in a distributed manner over several CPUs, the network, etc. Several paradigms appeared during the evolution of programming languages, like Object-oriented programming, the Actor-model indeed the 70's could be seen as a Cambrian explosion for programming languages, though, none of them depart from the idea of the von Neumann model.
Paul Morrison thought this new FBP paradigm to be more intuitive, the design and maintainance of these applications were simple in comparison to prior models. FBP production installations yielded over 40 years of changing legal and business environments. Designing programs with FBP facilitates fast evoluting human contracts for several reasons. First, the basic idea of component-oriented development cordoning off applicable logic into specific components, each component is independent, with precise specifications and goals. Then there is the sub-component.
The sub-component, or sub-net, are components built with other components. These sub-components have an interface and an implementation just as normal components do, except the implementation of these components are not in Mozart Oz but consists of other FBP components. A good analogy to illustrate the nature of sub-components

would be that of hardware logic gates. The logical AND gate consists of a NAND gate and a NOT gate as the implementation but the interface consists of two terminals A & B and OUT. This greatly enhances the reusability of the components and allows for greater powers of abstraction during the design process.
This allows laser focus on data transformations, ignoring the sequential nature of CPUs. Another analogy to illustrate the nature of this data transformation would be the humble beer brewery, more accurately how bottles are manufactured. Empty bottles are placed on a rail. They arrive one at the time to the filling machine, then onto the capping machine, finally they arrive at labelling machine. Should we replace the bottle with data packets or Information Packets, they would travel from component to component through bounded buffers. Each component transforms the data as specified.

A case example illustrating the powers of abstraction FBP offers would be a particular graph created by Chuck, a colleague of J.P. Morrison. Chuck built a network of some 200 components, without even the semblance of sketch on a paper napkin. This was largely possible due to the powerful nature of abstractions sub-components offer. They allow for a top-down approach to building programs. For example, a simple component could read a file, another make calculations on it and yet another would write the result to another file. The task can be broken into three coarse sub-components : the first reads the file and pre-process the data, the second make the calculation, the third post-processes the data and write it in a file. This natural way of dividing a program also permits the division of work. It's possible to implement the first component, without touching the other two. Once the first component is completed (it too could be a nested sub-component), the second is then implemented, once the final component is completed the entire lot could be packaged up into a single sub-component with a clear interface.
J.P. Morrison noted that FBP scales with the size of the programs. In classical programming, it's generally accepted the resources needed to develop an application is an exponential of the size of the application. Using FBP, adding a new feature is done incrementally. It's a very controlled environment and the development approach is predictable, one clearly knows in advance which components will be involved. By putting these two curves together on figure 2.1, it is possible to see that FBP becomes cheaper after time. Morrison and his colleagues found that it occurs even with small applications.

FBP components can receive parameters. These parameters can be decided at implementation time (default value), at design time (the graph editor) or even at run-time, where the parameters are given by other components. Experience in FBP development shows there are many components frequently used in different graphs. There will always be unique components, but most will come from a standard template library. The reutilization rate ranges from 14% from specific applications to 80% for more generic applications.
Thus a reduction of implementation time and the risk of communication errors are reduced. J.P. Morrison noticed making reusable components to be a non-trivial task, most cases these components come in pairs, ie. split and collate, read and write etc. As the component has a specific task and there are no-side effects, working on a node will not affect other nodes as long as the interface is respected. The feather in FBP cap is a Canadian bank, they have a FBP system in operation for over 40 years despite years of changing hardware, software, business and legal conditions. FBP made for efficient programs for huge data sets. As components only runs when there is data to be processed on their entry ports, there is no useless running code. Unless of course

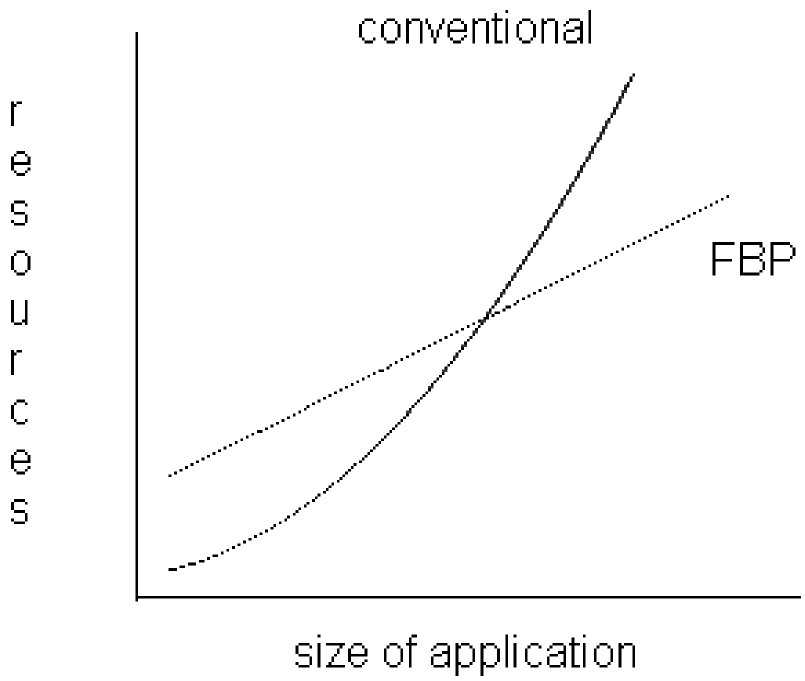


Figure 2.1: complexity of FBP programs, from [7]

you program a useless application. By making use of heavily reused components one has a higher degree of certainty that those components have had many eyeballs, thus the implementation is efficient.

The FBP paradigm also has interesting properties of concurrency. As nodes are completely independent, they are run at the same time. This is a very interesting feature for the current programming world, where computer have multiple cores and the programs are distributed over the network. Essentially it becomes possible to evenly distribute components over multi-cores. Implementing a slightly more intelligent graph manager allows for ensuring heavily connected nodes are on the same CPU.

Facebook has recently shown interest in data-flow programming. They created *Flux*, an application architecture to develop the client-side of the web-pages [1]. It's not FBP, but it's component oriented. Their argument is that traditional MVC design pattern doesn't scale well. Their solution was reactive system with a data-flow aspect to it. The problem they were trying to solve was to get new engineers up to speed as fast as possible, and to make their applications predictable. An interesting quote from their documentation :

> This structure allows us to reason easily about our application in a way that is reminiscent of functional reactive programming, or more specifically data-flow programming or **flow-based programming**, where data flows through the application in a single direction — there are no two-way bindings.

FBP is so well adapted for design and maintain application. Moreover, it seems that these kind of application scales well with the size of the program and the number of users.

2.2 General concepts

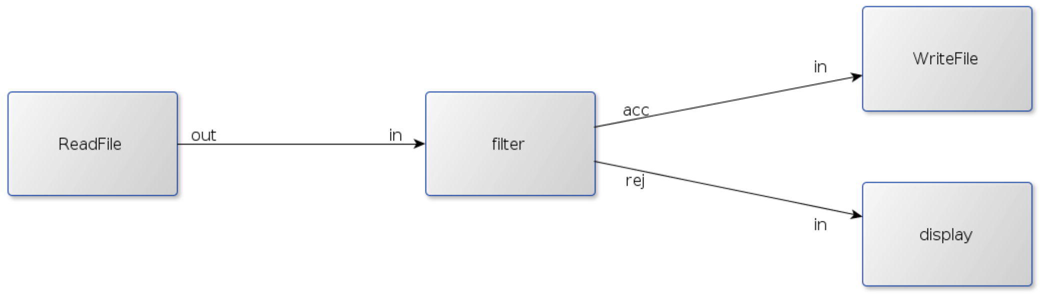


Figure 2.2: Example of FBP graph

The figure 2.2 is a very simple FBP graph. The rectangles are components, links between them are the bounded buffers and labels on the links are port names. This program is pretty much self explanatory. The contents of a file is sent to the filter component, line by line. The filter accepts some lines and rejects others, they are respectively send to the *acc* (accept) and *rej* (reject) output ports. The accepted lines are written to a file, and the rejected ones are displayed in the terminal.
This style of visual programming clearly shows what's happening to the data. The messages, a list of strings, goes through several transformations. Inside each component, data can be edited, deleted or just forwarded. Data can only exist only at one place in the graph. So if it's on a link, it's not inside a component or another link. If it's inside a component, it's not in another component or an a link. This behavior makes the lifecycle of the data packet clearly defined, removing any strange side effects due to multiple owners of the same data.

The data goes over the link in Information Packets (IPs). Each IP contains data and cannot be subdivided. They are described as a handle, or a pointer to a shared memory. IPs allows one to send a small number of bytes over the link, and not copy all the data each time. This approach would not possible if components didn't make use of shared memory. As the data would be too large.
A *flow* is the path taken by an IP. In above example, there are two flows : one from *ReadFile* to *WriteFile*, and another one from *ReadFile* to *display*, depending if the line of text is accepted or not.
More precisely IPs are transmitted on links that are bounded buffers, attached to an output port and an input port. The output port inserts IPs on the link and the input port reads IPs from the link.
IPs have a lifetime as soon as they are created in the graph. The creation and deletion are explicit. So, a component that receive an IP must explicitly say if it destroy it, or if it send it onwards. It is possible to see if all IPs are well managed, and none of them disappear without the developer knowing about it.
An IP can be grouped with the notion of stream. For example, fileReader can read multiple files, but how does fileReader distinguish the beginning and the end of the file? By means of the transmission of special IPs : *begin* and *end*. So the receiver can make blocks of IPs. In the case of fileReader, the sequence of IPs will be `begin line1 line2 line3 end`. This structure is recursive, so more complex data can be send. Say a file directory hierarchy is send by IPs. A stream will represent a directory, and sub-stream will represent sub-directories :

```
begin
  directoryname1
  begin
    directoryname2
    file1
    file2
  end
  begin
    directoryname3
    begin
      directoryname4
      file3
    end
  end
end
```

The first element of each stream is the directory name, thereafter would be a list of the file. If the IP is a sub-stream, that would represent another directory. This system allows one to quite easily send complex data structures. Alternatively, the entire directory tree may be sent inside a single IP.

Components are black-boxes with a single entry procedure point and several input & output ports. The main procedure is used to build the logic of the component. An example is given below. Multiple processes of the same component appear on the graph. For example, filter appears twice, yet there is only one implementation filter.

IPs arrive on the input ports and leave with the output ports. A component may have many input and output ports. IPs are permitted to come from different sources, with different goals. A port can be seen as a door to a house. If you are inside and you let your cat out, you don't really care about the shenanigans it'll get up to. Should a shadowy figure knock on your door and want come in, you peek through the eyehole to see if you should open the door. Sending a packet outside doesn't require any knowledge about the rest of the graph, simply put the IP in the outbound buffer and let it go. When an IP arrives, one needs to examine it and decide what to do with it; discard it or fulfill the computation then send it on it's way if need be. There is no constraint except that you have to handle it.
There is a notion of array ports. These ports are used to create a variable number of input or output ports at graph design time. They are used with a name and index. For example, "send this IP to the first (or n'th) OUT port" [7]. When the component main procedure is executed, it has information about the array port; which ones exists and regarding array input ports, how many nodes are connected. Array ports provide flexibility at design time. A simple example is the *add* component. A basic version would have two ports : x and y then send the sum to *res*. But why limit the addition of two numbers? The logic inside the component is the same for two or for ten numbers. So it's possible to build a component with array input port *numbers*, and then send the result of all the created ports to the *res* output port. It's also possible to choose only one sub-port, but not all. The same applies for the output ports. The figure 2.3 illustrates an array port. Notice name of the port is followed by a hash "#" and then the selection. The *add* component sends the result to a *loadbalancer* component. The

loadbalancer then sends the IPs in a round-robin manner over all the ports of the *out* array port. So the first IP is send to 1, the second to 2, the third to 1 etc.

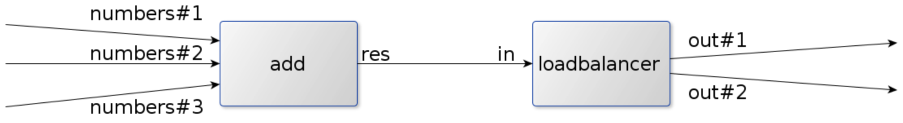


Figure 2.3: Example of array ports

FBP allows for sub-components or sub-nets, two different names for the same concept. The figure 2.4 illustrates it. A sub-component is sub graph. It's possible to build components with other components. The sub-component has external ports that are directly linked to ports inside the component. All external ports must be linked but not all the internal ports need be. From an external point of view, they look like and react as a simple component does. They have input ports and output ports and may be bound in exactly the same way. This feature allows for encapsulating complex parts of a graph, to allows the simplification of the global graph perspective. Thus a top-bottom design approach of the software becomes possible, by initially constructing a very general component. Then, implementing each component as needed. If it has a precise simple goal, one could implement immediately. Should the component be complex, then abstract it away as a sub-component with simpler components.

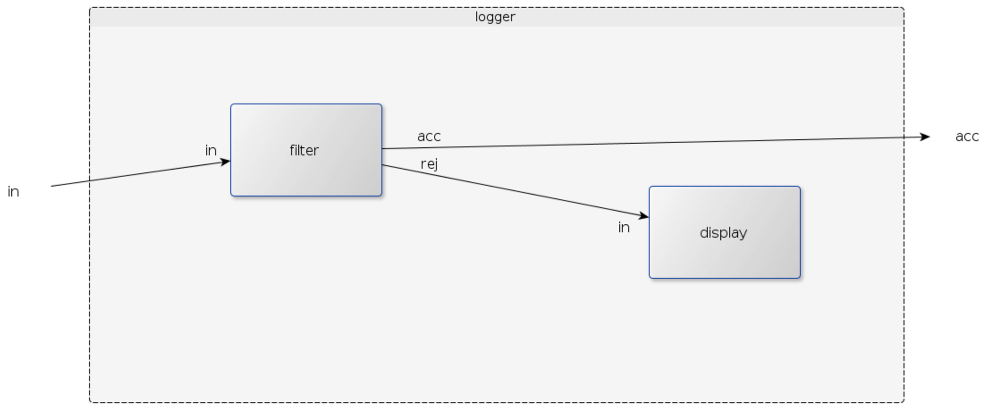


Figure 2.4: Example of sub-component

A component is executed by running the main entry procedure. Figure 2.5 shows the pseudo-code for the filter component.

The line "receive from IN using A" is used to receive a single IP from the input port *IN*, and inserted into the handle A. The "do ... enddo" is used to read a complete stream of IPs. The "send X to Y" commands are used to send the IP X to the Y port. This simple procedure reads all IPs from the *IN* input port, and tests it (represented by C). If true, the IP is send to the *ACC* output port, otherwise it'll be sent to the *REJ* output port.

Paul Morrison had a strong focus on reusability. As components are well defined and without side-effects, it becomes possible to easily hook up a pre-built component to a graph, and use it. Morrison noticed programmers in the 60's spent considerable amounts of time reinventing the wheel. Moreover, developers were paid by lines of code

```
receive from IN using A
do while receive has not reached end of data
  if C is true
    send A to ACC
  else
    send A to REJ
  endif
  receive from IN using A
enddo
```

Figure 2.5: pseudo-code for a component

written. Morrison preferred an approach whereby developer were compensated using the metric of code reuse, hence if your code was heavily re-used, you'd be paid more. This approach improved productivity and improves the quality of the components.
Though this is hard to achieve when components are hardcoded. All previous examples have been hardcoded. E. Lawton introduced the concept of *Initial Information Packet*, or *IIP*. IIPs are send on a special port which exist in all components, that of the *OPTIONS* port. When component execution starts, the IIP is transformed in an IP, and the component then uses the information accordingly. This allowed one to parametrize components. These options may be set at implementation time, design time or execution time. The former is default option the developer inserted into the component. The second, the graph designer's decision of an IIP send. The last one is when anther component is linked to the *OPTIONS* port, so arguments are dynamically managed.
This approach allows one to make the components highly reusable, without having to make near infinite customizations to hardcode. Take the *ReadFile* component as an example. Should it read a hardcoded file name one will soon be diving into the implementation code to change it as soon as one wishes to read a different file. The figure 2.6 illustrates an improved version. An IIP with the file path, and perhaps the method of file reading; line by line, character by character, one block at a time is used.

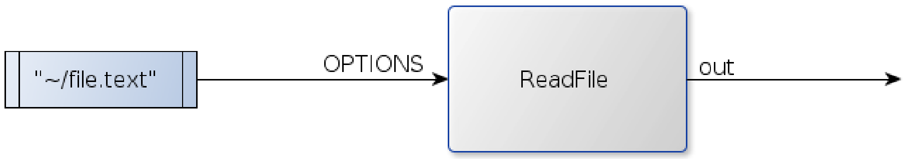


Figure 2.6: Example of Initial Information Packet

2.3 Programming concepts

FBP is an implementation of COP, which it itself an implementation of the agent model. There are two types of agents in FBP graph : components and IPs. The former are the active agents, holding a state and executing a procedure. The latter are the passive agent. They are just data, that goes from component to component. They are also unique, that make thinking about them more easy. Other particularities of FBP is the bounded buffers. They avoid live-lock, but make dead-lock possible. But dead-lock are always a design issue, and are thus avoidable. The asynchronous ports and array ports allow to do a kind of Erlang selective receive, and avoid state explosion in the component. They also make the execution non-deterministic. As discuss in [5], non-determinism makes the correctness proof and the debugging more complex, but respond to a real need when the applications are asynchronous.

A particular strong point of FBP is its concurrent nature of execution. As all components are independent and nothing is shared between them, there are no execution side-effects as each component manipulates different memory. So their location of execution does not matter, component code may be on a single processor, or on the other side of the planet. All that is needed is the ability to access to the input buffer.
Decentralized applications nowadays are no longer a novelty, they are hard and fast requirements in many business settings. In traditional languages, in many cases find it hard to crack this concurrency nut. Many languages of today use shared memory with mutexes, locks and complex asynchronous callback logic to ameliorate the problem. Typical FBP implementations make use of a scheduler to handle. Our approach makes use of light weight Mozart threads, thus negating the need for a scheduler. There will be times when manual locks are needed, for example should several components access a flat file. Here locking logic is achieved by using Mozart's locking primitives and is done only once inside this component, as mentioned earlier each components has a specific task, so the lock will be a simple and does not send you into a locking quagmire.

Chapter 3

HyperCard

This chapter is structured the same as the last, except presents HyperCard. Starting out explaining the interesting parts wrt Fractalide. The HyperCard book [2] goes much greater detail regarding concepts and utilization.

3.1 Introduction

HyperCard is a complete environment enabling the creation of programs, databases, presentations, and a whole slew of other possibilities. Developed in 1987 by Bill Atkinson. Apple included HyperCard in the Mac System 5 release as a free add-on or on sale for 50$, it was remove from sale in 2004.

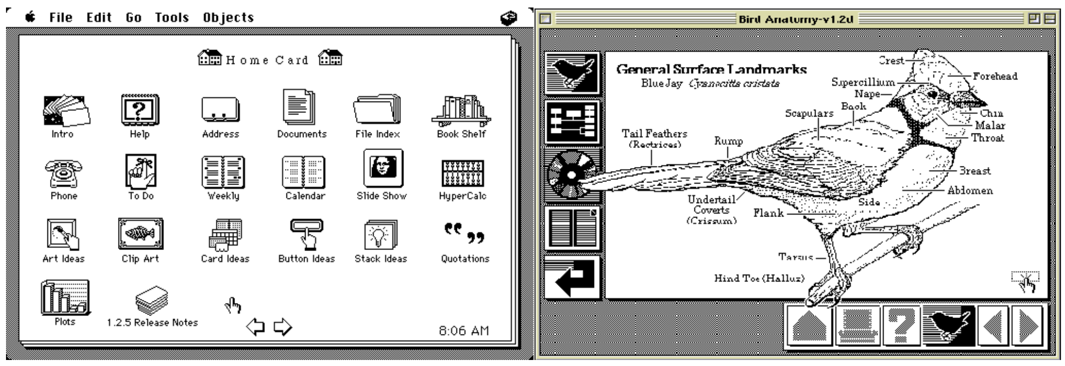


Figure 3.1: Two screen-shots of HyperCard. Left : the main stack. Right : a card.

B. Atkinson describes it as an authoring tool and an information organizer. It was one of the first hypermedia program that was distributed to masses, before the world-wide-web. Bill Atkinson says : "we're making this because we want to share something. One of the things that I find myself sharing is my understanding of programming". That explanation succinctly motivates many a design decisions made in HyperCard. The simple and user-friendly interface allows people to do more with their computer. Various applications were made on HyperCard, from the simple diary book to a whole encyclopedia. The former were done by a non-programmer. These people have no intention in mastering the art of programming, they simply have an itch they wish to scratch. Some enthusiasts became took to HyperCard built more complex tools, specifically a car salesman developed an application to help customers select a new car. Though HyperCard had a high learning gradient, complex and professional

software were also created. An encyclopedia was a good example, though they soon hit problems when distributing the card stack. Some companies developed entire stacks to manage their stock portfolios, debts, profit and losses. Even simple first version of SAP was built with HyperCard. It was not all business as usual, games too were developed, the original 1991 Macintosh version of Myst was constructed using a tweaked version of HyperCard.
Special attention was given to the initial experience a user has. Everything was meant to be extremely simple at first, it was critical that the user was able to come away from the experience with a positive experience. This is the reason some more in-depth features were hidden. With time, as the user gained courage and confidence, she was able to achieve more; build scripts, customize programs and move onto more complex applications. Large swathes of users probably would never have been aware that editing was possible, in many of those cases they simply did not need it. HyperCard demonstrated the importance of not confusing the user.
HyperCard in a sense brought the Homebrew Computer Club culture of sharing to the masses, in that card stacks we liberally distributed on floppy disks and in a sense were "open-source". Stacks were completely editable and thus encouraged the recipient to experiment and tweak the program by modifying the behavior, logic, UI etc. This ability to share card quite possibly made HyperCard very popular at that time. It could be argued that one of the factors to the downfall of HyperCard was the introduction of for pay editing in version two of HyperCard.

A major drawback of HyperCard was the complete absence of network access. B. Atkinson mentioned this in a 2002 interview [6] :

> I have realized over time that I missed the mark with HyperCard

and

> I grew up in a box-centric culture at Apple. If I'd grown up in a network-centric culture, like Sun, HyperCard might have been the first Web browser. My blind spot at Apple prevented me from making HyperCard the first Web browser.

In hindsight, HyperCard could have been the first World Wide Web browser. Today's WWW shared only documents until the onset of JavaScript. Hence learning from the successes of today's WWW such as only sharing a webpage and not an entire website should be adopted. Therefore sharing complete stacks would be an error. The ability to go from card to card, by downloading only the needed components, just as one does with a web page, is probably a more appropriated approach. HyperCard could have been the web with a hierarchical data structure, and a well integrated scripting language.

3.2 General concepts

HyperCard came with several concepts : stack, background, card, items, home stack, HyperTalk. This section elaborates on all of them and how they work together.
A stack is the location a user may place a card. Navigation between cards happens with reference to the stack. Basic operations such as next, previous, goto card X, or more complex operations such as a card finder, add/remove a card. The stack manages all

the database interactions. The stack indexes each of the inserted items (text, images, widgets etc) such that they are easily found.
Each stack has at least one background or canvas. Thus giving a common or specific background for each card. Should one develop a presentation with HyperCard, each card would have next and previous buttons, possibly in the bottom right corner. One could assign these buttons directly onto the background. Cards are transparent allowing the viewing of the background. A stack can have multiple backgrounds to offer multiple type of cards.
The card is a transparent canvas allowing the user can insert *user interface* widgets or items. They might be geometric forms (line, rectangle, circles etc.), buttons (simple, checkbox, etc.), fields, images, sound, etc. All these items are highly configurable. A button may have an image background, be disable, a field be set to disabled, etc. These items are data and generate events which get transformed by logic, updating the database. Lastly it is possible to drag-and-drop GUI widgets, change the size of a field via click and drag, as one would do with a paint program.
The home stack is the main stack. Upon HyperCard's launch, the user lands in this location. Essentially it is a stack of stacks. Hence a user may select the stack in question and navigate into it. Much akin to the Android operating system. The home screen consists of personalized widgets which represent downloaded applications.
Figure 3.2 illustrates all these elements and how they are linked together.

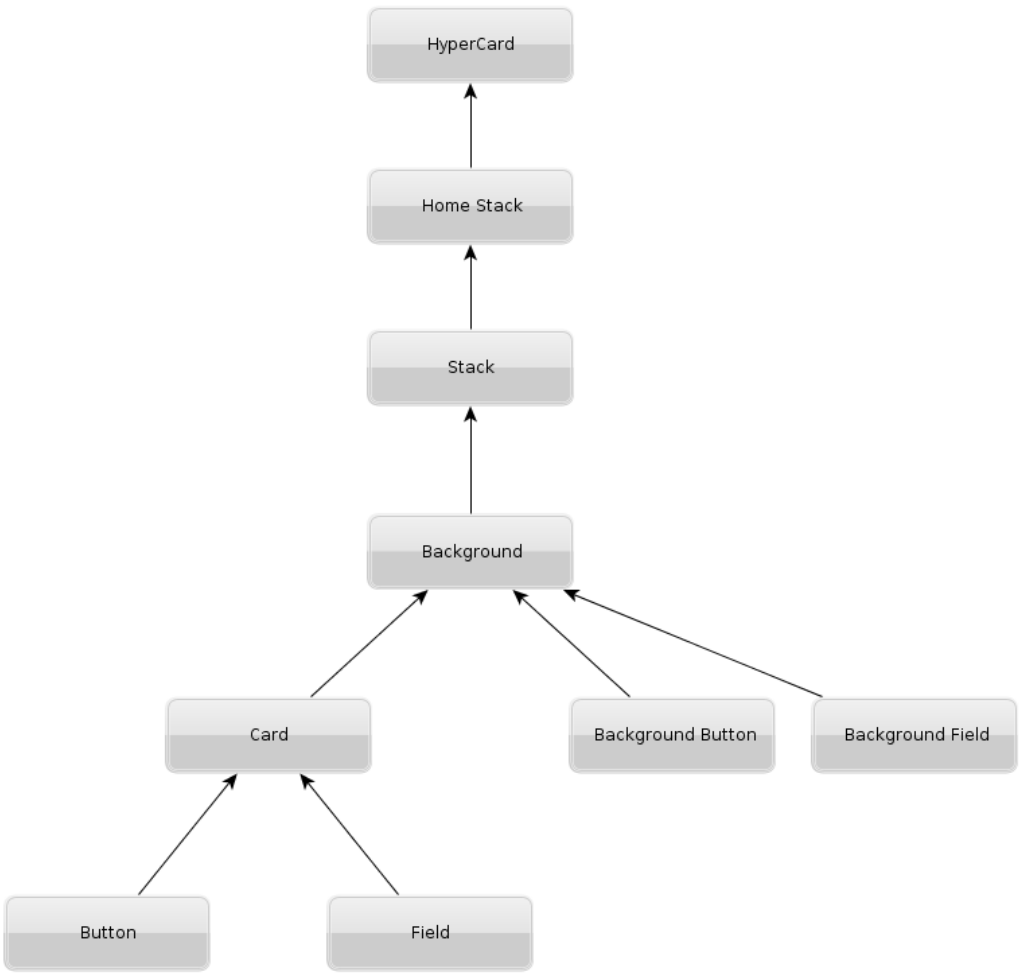


Figure 3.2: The hierarchy of HyperCard

```
on mouseUp
  put field "first number" into holder
  add field "second number" to holder
  add field "third number" to holder
  put holder into field "total"
end mouseUp
```

Figure 3.3: HyperTalk script. Example taken from [2]

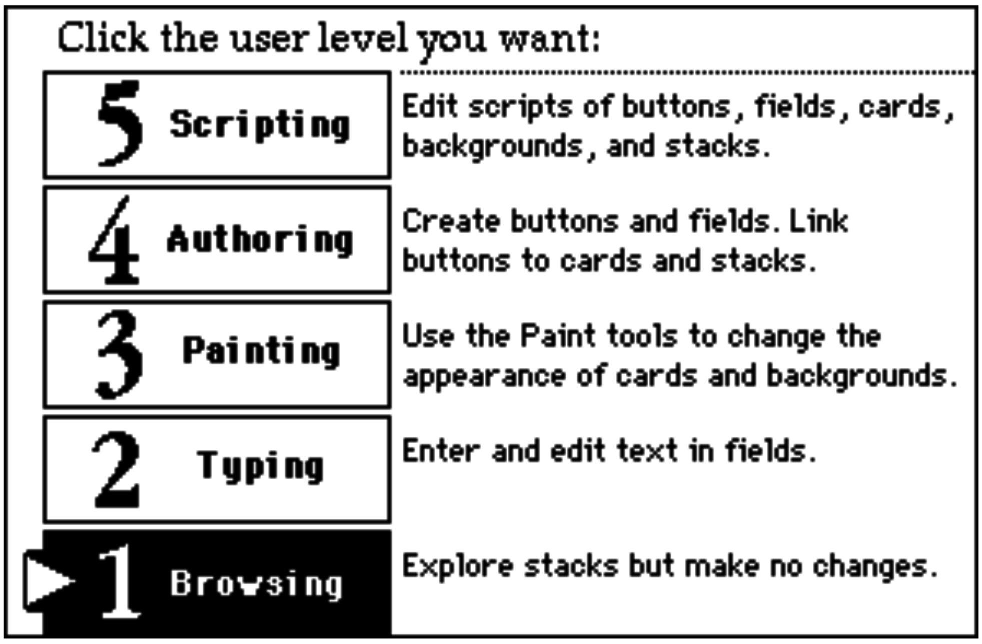
HyperTalk is the glue that bonds all these cards together. HyperTalk is a programming language designed to look as similar to English as possible. HyperTalk makes use of the event driven programming paradigm. Each item generates some sort of event. For example, the button can generate mouseUp, mouseDown, doMenu, click etc. The event is generated at the bottom of the hierarchy, and begins to escalate up the hierarchy. Events immediately rise to the card level, where they may be caught or not. If they are not caught, they continue to escalate until the top-node. The user can catch them by adding script like figure 3.3:

This snippet of code decides to do something with the message and then send it at the next level. The user can add scripts at each level of the hierarchy. This hierarchy in the messaging system offers a good system to do fine-grained behavior. A card could have a specific defined behaviour, by putting the logic at stack level this allows each card that has that particular widget to execute the same code, wherever in the stack.

HyperCard showed only what the first time user exactly what they needed without overwhelming them. This was achieved by creating five levels of complexity, known as *User Preferences*. Detailed in figure 3.4. The user starts out in the first level, and can easily switch from level to another. Each level had his own menu and tools associated with it.
Switching from the editing modes to the *Browsing* mode is also trivial. There is no compilation step. As soon as a design is finished, the effects were immediately visible. All compilation was performed in the background for the best user experience.

3.3 Programming concepts

HyperCard's contribution was a very intuitive way of handling events, albeit these events did not use the message passing concept, its unique way of handling messages and the ability to catch them at any level make possible a lot of things in an intuitive way. Thus making a specific behavior be globally applied, or not, either way the same level of complexity achieves the outcome.



Click the user level you want:

5 Scripting	Edit scripts of buttons, fields, cards, backgrounds, and stacks.
4 Authoring	Create buttons and fields. Link buttons to cards and stacks.
3 Painting	Use the Paint tools to change the appearance of cards and backgrounds.
2 Typing	Enter and edit text in fields.
1 Browsing	Explore stacks but make no changes.

Figure 3.4: The five levels of complexity

Chapter 4

Fractalide

4.1 General presentation

Fractalide elegantly mixes the best of HyperCard and FBP. We do this by replacing HyperTalk with FBP and HyperCard cards are simplified into FBP components. The outcome is a unique paradigm : Hierarchical Flow-based Programming.
Fractalide is built on Flow-based Programming, on which the HyperCard concept is added. To ensure we retain the benefits of HyperCard, some additional features were added to FBP. This will be explained in the first section. The second section explains how HyperCard was adapted to FBP.
All examples and documentation is in the next chapter.

4.2 Foundation : FBP

4.2.1 The Fractalide approach

Multiple failed attempts at implementing FBP shows that careful reading of Morrison's work was required in order to achieve a correctly executing FBP implementation. Therefore making any extensions to FBP was carefully thought through. Our first goal was to extend FBP to allow for global distribution of components and executing graphs. The second modification is to allow for a highly dynamic system to enable compatibility with our modified HyperCard.

In order to allow for a globally distributed executing FBP graph it was imperative to make every component independent. They should share nothing, except of course ports. Therefore the graph shouldn't have a scheduler. Thus each component becomes a little servlet waiting for IPs to arrive on inbound ports. This allows adding a component to a graph to be minimal fuss and bother. This approach allows us to distribute components over multiple cores or over the network.
As we wish to create long running distributed graphs we need to make every component extremely dynamic. Each component must be able to introspect itself and perform reflection. A component in Fractalide is a representation of state, which is explained later, which is fully accessible and editable. It is possible to see how many input ports a component has, or what is the IIP is pertaining to that port. Moreover, reflection is possible because component state is fully editable at runtime, this is achieved in a consistent manner. Therefore it is possible to add ports, change the main procedure, indeed manipulate any aspect of the component, without stopping the graph or even

that component in question. Say for example a bug is found in a running component. Should stop the component to debug it, the connection will be lost. This degree of dynamism gives great leeway to maintainers of long running components. As a side effect it is now much more simple to implement an error handling mechanism.
It should be stated that this proof of concept prototype does not support network distribution. Though in the near future this will become a reality.
Lastly, any extension to our implementation of FBP must not hinder any Mozart Oz language concepts. We therefore make no assumptions on IPs, so any data arriving could consist of any Mozart Oz concept, be it a functional data structure, object, or the humble dataflow variable.

4.2.2 Definitions

Each concept needed to understand how Fractalide's FBP implementation is detailed below. These are only definitions, without explanations or example use cases.
A Flow-based program is a graph composed of **components** and **sub-components**. They are linked together to create **streams** of **Information Packets (IP)**. IPs are send on **channel**.

A **channel** is a bounded buffer between two components. It has a maximum capacity size. Should there be no space, it is not possible to write an IP to the buffer. Though it is still possible to retrieve IPs from the buffer.

A **flow** is composed of all the channels or paths an IP follows during its lifetime, from creation to destruction.

An **IP**, is an information packet sent over a channel. It is a simple Oz variable with no type limitations.

Components are composed of input and output ports. A component executes a procedure when processing an IP. During execution, the procedure accesses the options and state buffer.

There are two types of **ports** an input and output port. For each type, there are two sub-types; a simple port and an array port.
Input ports are bounded buffers. The array input port are a set of simple input ports. It is possible to setup many-to-one connections to an input port. It is of course possible to retrieve (put) and insert (get) IPs into inbound and outbound buffers.
The put operation inserts an Oz dataflow variable into the buffer. If the buffer is full, the operation waits till there is space then it completely. Bounded buffers are essential to avoid graph execution live-lock.
The get operation returns a single Oz dataflow variable. If no IPs are present on the buffer, the operation blocks till an IP becomes available.
Output ports are links to input ports of other components. An output port can be linked in a one-to-many fashion. If the case arises that an IP needs to be duplicated for each port, then the use of an array output port is required. An array output port is set of simple output ports. A set of which is unknown at component implementation time but known at graph design time.

The component has a main **procedure**. This procedure acts as a single point of entry and has access the all the component's input port buffers, they may also send messages out via the output port buffers. The entire component state is also given to this procedure. This is what allows introspection and reflection.

IIPs are saved in a record. Each parameter had a feature and a value. This record

can be changed with IIPs sent to the *options* input port. These IIPs contain a record with some values. The behavior of this port is to adjoin the new record to the old one. The new value overwrites the old value.
The options input port is an asynchronous simple array port. Asynchronous in the sense that this port joins two records as soon as it receives an IP, without any other condition.

There is an **internal state** for each component. It is a dictionary that is shared by all executions. The state is not reset at each execution. This state is quite different from the options state as it may be modified by the execution.

A **sub-component** is a FBP graph with external ports which simply forward IPs to input ports of the components in the sub-graph.

A **external port** simply forward all messages received to the attached port. A virtual port can be attached to several ports. There is no limitation on the port types : simple, array, input or output. But a single external port must be linked to a set of port of a single type.

4.2.3 Semantics

An **execution** of the component is the execution of its main procedure. This is done as soon as four requirements are met; the first is there is a least one IP in at least one input port, the second is that all options are bound, the third is that previous execution is completed and the last requirement is that the run variable is set to true. The run variable is set to true at the creation of the component and can be changed with the start and stop actions.
There is only one execution at any one time. If at the end of an execution the start requirements are met, the component repeats execution.
For component with no input ports, it's possible to manually override and jump-start execution sending the 'start' atom to the component or by sending an IP to "START" input port.
Under some conditions, it is possible that an IIP sent to the options input port starts execution. This case is only if the component has unbound options and all the other requirements are met. As soon as all the options are bound, the component is ready to start. This is not true if the component has no input ports. In this case, a bounded options record will not start component execution.
Any changes to the component's state do not take effect immediately, but only the next time the component is executed (see 7.5 for more detail).

4.2.4 Difference with classical FBP

Fractalide's first major difference with classical FBP is that our components are completely autonomous. There is no management layer that schedules component execution. This approach has some positives, namely components are completely independent of each other (save for ports). Running components on different Mozart Oz Virtual Machines is trivial. A possible drawback is perhaps performance. FBP requires specific execution order that follows the data, so schedulers in other implementations incorporate optimizer for the specific tasks. We negate this need as Mozart Oz provides light weight threads, hence this dragon is holed away quite nicely.
Another distinct point about Fractalide's array ports is that we are able to assign a name in order to do selection. Classical FBP requires one to specify the nth selec-

tion. Fractalide specifies the selection with any atom. An example: Classical FBP would look something like this: `OUT[1] OUT[2]` whereas Fractalide's approach is this: `action#Motion out#1`. This is a much more civilized approach to managing events in FBP.
IIPs are not managed in the same way as classical FBP, which transforms IIPs into an IP at the beginning of component execution. Fractalide forces IIPs to be a record, this approach allows for more information regarding component's implementation, yet at the same time satisfies the intended behaviour of IIPs. Setting parameters with records is a natural fit. The last point regarding IIPs is that an IIP will never launch execution in a classical FBP graph.
One of the biggest most serious changes made to Fractalide's implementation is the lifetime of IPs is not managed. Classical FBP demands that all components that create or receive an IP must explicitly say when they discard or send it on. Fractalide drops this notion for several reasons. The first is that this information is not necessary for the execution of our independent components. Fractalide aims to have a minimal implementation of the component, a fewer code means fewer bugs. Secondly, the developers have become accustomed to the garbage collector, it is better to let the GC handle these IPs. Classical FBP forbids one-to-many and many-to-one links as it would wreak havoc on the scheduler that needs to know about the comings and goings, creations and destructions of every IP. The classical FBP scheduler must take account of every IP. Our lack of scheduler immediately opens up Fractalide a much richer and diverse set of graph configurations, without the need for Splitter and Concatenation components as advocated by classical FBP.

4.3 Extension : HyperCard

4.3.1 Fractalide approach

Implementing a HyperCard prototype required studying many design choices and concepts. A little analysis of them is necessary to rebuild a such system, thus ascertaining the strong and weak points to enable our distributed HyperCard FBP implementation. One of HyperCard's biggest weaknesses, as Drew Ivan expounded on at Notacon 5, is that HyperCard can do a lot; presentations, databases, applications, hyper-media browser etc. but is excellent at none. Fractalide's main goal is to build distributed applications easily, so relevant design choices were made to achieve this. This degree of flexibility is not dropped, though it will require a little more work to be as intuitive as on the original HyperCard.
A rather serious choice was to replace HyperTalk, the backbone of HyperCard with FBP. This allows us to benefit from FBP features, such as concurrency, reusability and separation of business logic from component implementation. This approach, we hypothesize, to be more intuitive to non-programmers than line of codes. Our goal is to allow anyone speaking any natural language to design applications. The drawback is that a developer will be needed to implement an initial set of usable components, but once it is done, it becomes available to the world.

HyperCard has many strengths. Namely introducing a single environment that supports trivial switching between run mode and design mode without a compilation step. In a sense this makes Fractalide a fourth generation graphical development environment.

As elaborated on earlier HyperCard supports event programming with hierarchy. This approach is a natural fit for UI applications, as user actions are close match to events. The introduction of a hierarchy in the event handling is a powerful concept. Allowing for the handling of specific behaviors at any level in the hierarchy.
Similarly a notion of hierarchy is also used from a database point of view. Data is easily classified, depending of how the user created the stack. Thus it is easier for the user to use this hierarchy, along with all the associated data links.

Fractalide tries to balance the best of these concepts. In order to build better applications, better building blocks are needed something HyperCard is sorely lacking. HyperCard provides items like buttons, fields, images, geometric shape, etc. These are fixed building blocks. As cards can only contain items and it is impossible for the user to add custom items, this limits the user greatly. Thus rendering the re-use of already created cards impossible. Especially when a user wants to display two cards at the same time. This approach is not ideal for building complex applications.
HyperCard's concept of a single "background" that manages a stack, limits the building of complex cards, this approach is more suitable for creating presentations and not application development. Hence an adjustment to HyperCard was called for.
Fractalide's approach was a simplification; everything becomes a card. Items are cards, cards are cards and even stacks are cards. Giving us a single highly configurable building block. Much akin to Smalltalk's "everything is an object". This allows modularity and reusability. A simple button is at the same level of abstraction as an entire complex application. It thus becomes possible to replace a button by an application. It is also possible to include a card in another card, and in this way provide extend a HyperCard "item" to endless means. As each card also contains its own logic, strong encapsulation results.
Then, what is a Fractalide card? In HyperCard, a card is an entity that receive events, process the events and possibly emits events. A card may also be displayed, the description of which sounds much like a FBP component. Therefore in Fractalide, a card is a component/sub-component which deals with events. The UI is managed by a set of specific cards. This makes for an elegant mix of the HyperCard and FBP concepts, as explain later.
Building an application with the card concept keep the notion of hierarchy from HyperCard, but with a particularity, inthat HyperCard's hierarchy is mandated to always be static; items -> card -> background -> stack -> main stack -> hyperCard. In Fractalide, the user is at liberty to construct their own hierarchy, depending on the the application specifications. Should the application call for no hierarchy, then a single branch will be created. This approach allows fine grained control at different layer of the application, it then becomes possible to consider a part of the application as a single block, or even to achieve component specific behavior.

4.3.2 Definitions

Therefore a card is the mix of classical flow-based programming and event-programming. There is an *action* flow traversing the cards. Actions come into the component and possibly leave it. A card may also generate new actions.

An action is simply a record. The label of the record represents the name of the action. The record can have a set of feature/values that represent the arguments of an

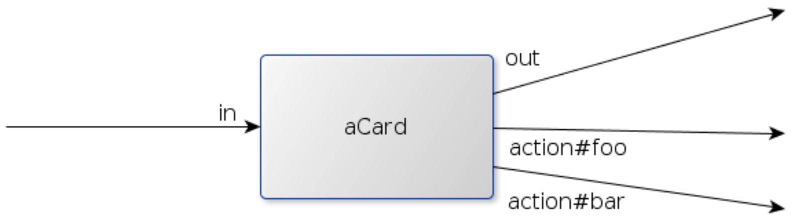


Figure 4.1: A card example

action.
Actions come into the card via the simple input *in* port. The component handles the action and/or sends it out. Regarding the output, there is two output ports : *out* and *action*. The first is a simple port and the second is array port. By default, every action is sent to *out*. Though an action may be specified with the *action* port, and the action is just sent on this specific port, and no longer the default one. The specification is implemented by connecting a component to the *action* array port with the name of the action. These are the default ports of all cards, though it can be more input ports. An example in section 5.8 makes use of QTk cards.
Figure 4.1, illustrates a typical card. It has two specifications for the action port. The events *foo* and *bar* will not be send on the *out* port, but on the *action* port.
This simple protocol defines cards and achieve a eloquent HyperCard and FBP fusion, thus keeping the best of breed. The concepts of the HyperCard stack is no longer present in this model. In fact Fractalide's stack is nothing more than a special card. The *stack* card must therefore understand the actions such as *next*, *previous*, *add*, *find*, etc. In this way, it's possible to have very similar capabilities of HyperCard. Due to the dynamism of Fractalide's FBP implementation, a change in the graph will immediately be applied to the running application. The dynamism of HyperCard to quickly applications is thus incorporated in the FBP.
Lastly, all input actions used in a card must be caught after the input *actionWrapper* card in order to explicitly handle actions. If an action is not caught, it will not influence the card's behaviour.

As cards are also sub-components as well as being data. Card become the perfect building block, allowing for the construction user interface applications or indeed simple data storage applications. As cards react to events like `getData`, and respond with the require information, an application such as “The news of the day” becomes easy. When components become easily shareable over the network, the data will be easily shareable. That leading to a complete system for sharing information, applications or components over the open Internet.

Chapter 5

Documentation

5.1 A basic graph

This first section will go over a *hello world* example, to give the intuition from the component creation to the graph loading and starting. Then, the next sections will details the API.

The graph will be very simple. There are two components : an IP generator sending “hello world” and a component to display the IP. The first has a single *output* output port, which connects to the *input* input port of the second component.

The first step is the graph creation. Fractalide uses a personal version of the Domain Specific Language (DSL) created for the first versions of FBP. That looks like :

```
hw(helloworld) output -> input finalDisplay(simpleDisplay)
```

This line means “the component *hw* of type *helloworld* had with the output port *output* links to the input port *input* of the component *finalDisplay* of type *simpleDisplay*”.
This graph can be loaded with the *graph* library. But first let us explain the directory hierarchy of Fractalide.

```
fractalide
|-- lib
|   |-- component.oz
|   |-- subcomponent.oz
|   |-- graph.oz
|-- components
|   |-- simpleDisplay.oz
|   |-- QTk
|   |   |-- window.oz
|   |   |-- lr.oz
|   |   |-- ...
|   |-- genopt.oz
|   |-- calculator.fbp
|   |-- helloworld.oz
|-- launcher.oz
```

The *lib* directory contains the main libraries for the component, the sub-component and the graph. The *components* directory contains all components needed by the platform.

```
functor
import
   % The path to the component library
   Comp at '../lib/component.ozf'
export
   CompNew
define
   fun {CompNew Name}
      % R is the initial record
      R = component(name:Name type:helloworld
                         description:"send the IP \"hello world\""
                         outPorts(output)
                         procedure(proc{$ Ins Out Comp}
                                        {Out.output "hello world"}
                                     end)
                        )
      % The component library is called to create the component
      {Comp.new R}
   end
end
```

Figure 5.1: HelloWorld component

The oz component and the ".fbp" components (like the one showed in example) are place there indifferently. When a graph is loaded, it will examine the *components* directory for the type. The specified type field is the path without the extension. It is possible to have a sub-directory, like the *QTk* in this hierarchy. To load the window.oz component, the declaration will be `win(QTk/window)`. The launcher.oz file is an simple application that runs a graph when given the path to the *.fbp file as an argument. Thus to start the graph, the best is to save it as a file "graph_helloworld.fbp" in the fractalide directory. It will not yet work as the components don't yet exist.

These two components are Oz components, they must not be subdivided as they have very simple specifications.
In Fractalide, the component descriptions are made with an Oz record. This record contains several feature/value pairs. Each of them specify a part of the component, like its name, description and main procedure, etc.
The *helloworld* component is implemented as in figure 5.1.

The first point to note is that the components are built in an Oz functor. This functor must export the *compNew* function which returns a new component, created with the help of the component library. The record is pertinent, the first three fields are metadata about the component. Then there is the output port named *output*. The main procedure is a function that achieves a single goal: send the string "hello world" to the *output* port.
Notice the the display component in figure 5.2.

This component has no output port, only a single input port called *input*. The procedure is also very simple. The first IP from the buffer is saved in a temporary variable, and then printed. When the procedure ends the IP is "destroy", without being sent out or saved.

```
functor
import
   % The path to the component library
   Comp at '../lib/component.ozf'
   % Library to access the console
   System
export
   CompNew
define
   fun {CompNew Name}
      % R is the initial record
      R = component(name:Name type:simpleDisplay
                    description:"print␣the␣received␣IP␣in␣the␣console"
                    inPorts(input)
                    procedure(proc{$ Ins Out Comp} IP in
                                 % Retrieve one IP from the input port
                                 IP = {Ins.input.get}
                                 % Print it
                                 {System.show IP}
                              end)
                   )
      % The component library is called to create the component
      {Comp.new R}
   end
end
```

Figure 5.2: SimpleDisplay component

These two files can be saved in the *components* directory, then compiled [1]. The graph can be started with the help of the *launcher* functor :

```
ozengine launcher.ozf graph_helloworld.fbp
```

This displays "hello world" in the console window. Your very first Fractalide application! The program will not terminate. In that sense each component is autonomous, the display components do not know they will not receive any more IPs, so it stays alive waiting patiently. The correct way to stop the execution will be explained later. A sub-component is created with the *.fbp* file extension. Adding syntax for the external ports, it becomes possible to execute a *.fbp* file as a entire graph by giving its path to the launcher, or to load it as a sub-component if its type is given in another *.fbp* file.

5.2 Declaration of a component

A Fractalide component is created by defining an Oz record which contains all the necessary information. The label of the record is *component*, and several pairs of features/values describe the structure. When a feature is not specified mandatory, it is optional. The *component* library had a procedure *new* which takes the record as argument and returns the entry point.

- Name, type and description

These three features are mandatory. The value of the two first features are an atom respectively representing the name and the type of the component. The *name* of a component is a way to uniquely identify it in a graph. The *type* of a component is used to detail the relative path of the component. It's so possible to reload a component or to export the graph.
The third field *description* contains general information about the component. It's a virtual string which explains the goal and specification of the component. It is mandatory because the documentation is a key for sharing and reusability.
They are used as follow :

```
component(name:adder type:'calculator/adder'
          description:"add␣all␣number␣received␣in␣the␣input␣port")
```

- Input port

All input ports are placed in two different records. The label *inPort* for the simple ports and the label *inArrayPort* for the array ports.
The elements of the records are nested records. The label is the name of the port. There is an optional feature size that specify the size of the bounded buffer.
For example :

```
inPorts(name(size:1)
        name2
       )
```

This record represents two simple input ports : *name* and *name2*. *name* has a maximum of one IP in its bounded buffer. *name2* has a maximum 25 IPs in its bounded buffer. It's the default value.
Array input ports have the same default values :

[1] `cd components && ozc -c display.oz && ozc -c helloworld.oz && cd ..`

```
inArrayPorts(name(size:10))
```

We have an array input port *name*, and all the buffers have maximum 10 IPs.

- Output port

To declare both simple and array output port the only argument required is the name:

```
outPort(output time)
outArrayPorts(variables)
```

These two records will create the simple output port *output* and *time* and an array output port *variables*.

- Procedure

A component had a mandatory procedure which is launched at each execution. It is specified by a record labelled *procedure*. The procedure has three arguments : *Ins*, *Out* and *Component*. The first is the input buffer(s), the second is the output ports and the last is the component itself.

```
procedure(proc{$ Ins Out Comp} ... end)
```

Ins is a record allowing access to all the buffers. The features are the port's names, the values are the buffers. The array input ports are represented with a sub-record. For example, if we had a component with the input port *in* and the array input port *numbers* which has the selections *1* and *2* bounded, the record will be :

```
buffers('in':<Buffer> input:buffer(1:<Buffer> 2:<Buffer>))
```

The IPs inside a port are accepted by executing the function `{Ins.portName.get}`. If it's an array port, it will be `{Ins.portName.selection.get}`.
To send a message to an output port, the *Out* record is used. If it is a simple output port, a message *aMessage* is send on the *output* by doing `{Out.output aMessage}`. The array output ports are accessible in the same way : `{Out.arrayOutput.1 aMessage}`. It is possible to send any Oz values.
The bounded buffer will block if there is no IP to get, or if it's full when sent to it. If the function `{Ins.name.get}` is called and the buffer is empty, it will wait until an IP arrives. If the procedure `{Out.output "message"}` send an IP to a full output port, it will block until there is a place. This system make live-lock impossible.

Putting all together: create a component with a unique input port *input*, print the IPs to the terminal and send the message out will be :

```
component(
   name:display
   type:display
   description:"display the IP and send it to output"
   inPorts(input)
   outPorts(output)
   procedure(proc{$ Ins Out Comp} IP in
                IP = {Ins.input.get}
                {System.show IP}
                {Out.output IP}
             end)
   )
```

If we want to concatenate many input strings, but we don't know how many, we can use an array input port. The utility of array port is to have an unknown number of connection at implementation time of the component. For example, here, it's possible to have 2 input string to concatenate or 100 with the same component.

```
component(
   name:concat2
   type:concat
   inArrayPorts(input)
   outPorts(output)
   procedure(proc{$ Ins Out Component} Concat in
                Concat = {Record.foldL Ins.input
                          fun{$ Acc In}
                             {List.append {In.get} Acc}
                          end
                          nil
                         }
                {Out.output Concat}
             end)
   )
```

- options

The options are declared in a record labelled *options*. They must have a value associated to each feature, but the value can be undefined. If there are options unbound, the component will not be executed. The options can be changed during the graph execution with the help of the options input port. For example, if we want to have a default color option at blue and an text option that must be defined by the user :

```
options(color:blue text:_)
```

- state

This is nearly the same as options. The label is *state*, and undefined feature doesn't block the execution of the component.

```
state(color:blue text:_)
```

An example to illustrate the utility of these different concepts, and so why they exist.

Take a component that have two simple input ports : x and y. The idea is to do one of the for main mathematical operation on the received value (numbers), and add it with the previous result. The final result is send on the *result* output port.
As the desired operation is unknown for the moment, we will use an option *operation*. This option will be undefined, so the graph designer will have to specify which operation he want.
The last result will be saved in the internal state *ans*. Figure 5.3 shows the implementation.

The two following examples explain the concept of array input ports and array output ports. The common idea is that the developer doesn't know how many and which component will be bound to the port, because it's a design choice. The following examples implement the two components *add* and *loadbalancer* presented in figure 2.3.

```
component(name:example1
          type:basicMath
          inPorts(x y)
          outPorts(result)
          procedure(proc{$ Ins Out Comp} X Y Res in
                       X = {Ins.x.get}
                       Y = {Ins.y.get}
                       case Comp.options.operation
                       of add then
                          Res = X+Y
                       [] sub then
                          Res = X-Y
                       [] times then
                          Res = X*Y
                       else
                          Res = X/Y
                       end
                       Comp.state.ans := Comp.state.ans + Res
                       {Out.result Comp.state.ans}
                     end)
          options(operation:_)
          state(ans:0)
         )
```

Figure 5.3: basicMath component

Here is the implementation of the simple *add* component. The idea is to make a fold on the record containing all the buffers of the array port. The result is the accumulator. So it's independent of the number of ports that are been created for the input port.

```
component(name:adder
          type:'calculator/adder'
          inArrayPorts(input)
          outPorts(result)
          procedure(proc{$ Ins Out Comp} Res in
                       Res = {Record.foldL Ins.input
                              fun {$ Acc In}
                                 Acc + {In.get}
                              end
                              0}
                       {Out.result Res}
                    end)
         )
```

It's possible to connect one to many component to the *input* array port, and the result will be the sum of all the first IP inside each buffer.
For the array output port, a good example is a load-balancer. The idea is that the load-balancer receives an IP and sends it to one of the output link. If there is three components connected to the output port, it will send the first IP to the first component, the second to the second, the third to the third, the fourth to the first, ... We see here that is possible to use only one specific port of the array port, and not use all of them.

```
component(name:loadbalancer
          type:roundrobin
          inPorts(input)
          outArrayPorts(output)
          procedure(proc{$ Ins Out Comp} IP Num P in
                      IP = {Ins.input.get}
                      % Num is the next number for the argument
                      Num = (Comp.state.cpt mod
                             {Record.width Out.output}) + 1
                      P = {List.nth {Record.toList Out.output} Num}
                      {P IP}
                      Comp.state.cpt := Num
                    end)
          state(i:0)
         )
```

This code will work for any number of component attach to the output port.

A last example will show how the stream are handled in Fractalide. Figure 5.4 shows the main procedure and a help procedure. The stream will be a sequence of integers to sum : `begin 3 3 1 end`.

The *StreamReader* procedure is a recursive function that accumulates the sum. Each call reads an IP from the buffer. It stops only when an IP containing the 'end' atom is received. The streams are in fact managed at the application level, and not the built-in the library. There is no assumptions on the IP type, no more than on

```
proc{$ Ins Out Comp}
   {StreamReader Ins Out Comp}
end
proc{StreamReader Ins Out Comp}
   fun {Rec Acc} IP in
      IP = {Ins.input.get}
      case IP
      of begin then
         {Rec 0}
      [] 'end' then
         {Out.output Acc}
      else
         {Rec Acc+IP}
      end
   end
in
   {Rec 0}
end
```

Figure 5.4: procedures to read a stream

the values saved in the options and the dictionary. This allows the complete set of language concepts offered by Mozart oz, there is no restriction. A set of components can, for example, use an Oz class as IP, and have very similar capabilities as JavaFBP.

5.3 Declaration of a graph/sub-component : FBP file

A *.fbp file describes a FBP graph. A component/sub-component is represented by a unique name and a type. The first time the name is used, the type must be specified. Thereafter, the name is enough :

```
name(type)
name()
```

The name is a is an alpha-numerical string of character. The type represents the relative path of the component from the components directory. The component and sub-component are specify in the same manner, the extension of the file is omitted. For example, load the component define in the functor "./components/display.ozf"

```
displayName(display)
```

and load the sub-component "./components/QTk/window.fbp"

```
winName(QTk/window)
```

Links between the components are made by specifying an output port, an arrow showing the link and finally the the input port.

```
winName() out -> input displayName(display)
```

It's possible to chain the binding

```
foobar() output -> in winName() out -> input displayName()
```

Every line return or "," make a new analyses of the next characters. So it's impossible to do

```
foobar(foo/bar) output -> in winName() out
    -> input displayName(display)
```

but it's fine to do

```
d1(display) output -> input d3(display), d2(display) output -> input d3()
```

To make a sub-component, one needs to define external ports, this is achieved by means of an arrow: "=>". Using it from an output port create an external output port, and equally so for input port. Several links can be created from or to an external port. For example :

```
input => input d1(display) output -> input d3(display) output => output
input => input d2(display) output -> input d3()
```

This FBP file represents a sub-component that has two external ports : input and output. When an IP is send on the external input port, it is duplicate in the input port of *d1* and *d2*.
This file can also be opened as an autonomous graph, without the external port.

Comments are allow as in an Oz file, so every end of line is ignore as soon as the character "%" is see.
It's possible to forge IP with the help of the double quote character. Everything inside double quote will be compiled inside an Oz value and then send to the specified port. For example :

```
"0" -> input generator(generator)
"create(bg:white)" -> in button()
```

send the *Int* 0 in the generator and the *record* `create(bg:white)` in the button. These IP will be forged when the graph is started.

5.4 Use of a component

It's possible to interact directly with the components, by sending messages to the entry point. The function {Component.new Record}, that created a new component, returns a function with one argument. With this function, it's possible to interact with the component : send message, change a procedure, bind an output port, etc. The argument must always be a record. The label is the name of the action, and the pair feature/value are the arguments.

- getState(?State)

The variable *State* will be bound to the actual state of the component. The state is a record containing all information : name, descriptions, ports, buffers, etc. The section Implementation goes in the detail about the state.

- setState(State)

This operation sets the state to the variable *State*. There is no verification about what is done. These actions can break the component and is irreversible.

- start

This action sets the run variable to true. If the starting requirements are met, the component executes the main procedure.

- stop

This action set the run variable to false. As soon as this action is received by the component, the execution is stopped. This action can lead to inconsistency in the FBP graph.

- exec

Check if the starting conditions are meet. If so, an execution is launched. This action doesn't change the run variable.

- send(Port Msg ?Ack)

The message *Msg* will be put in the buffer of the simple input port *Port*. The Ack variable is unbound. The sender must wait for the Ack variable to continue.

- send(Port#Argument Msg ?Ack)

The message *Msg* will be put in the *Argument* buffer of the array input port *Port*. The Ack variable plays the same role as before.

- bind(OutPort InComponent InPort)

This action bind the *OutPort* of the sender component to the *InPort* input port of the *InComponent* receiver component.

- bind(OutPort#Argument InComponent InPort)

The same as *bind()*, but for the specific *Argument* place of the array output port.

- unBound(OutPort InComp)

The link(s) to the *InComp* component are removed from the *OutPort* simple output port.

- unBound(OutPort#Argument InComp)

The same for the specific *Argument* place of the array output port.

These messages can be easily tested in an Oz interpreter.

```
declare
% Load the functor
[Display] = {Module.link ["./component/display.ozf"]}
% Create an entry point
D = {Display.new defaultDisplay}
% Display the state
{Show {D getState($)}}
% Send a message
Ack
{D send(input "hello␣Fractalide" Ack)}
{Wait Ack}
% Bind
D2 = {Display.new secondDisplay}
{D bind(output D2 input)}
```

This is a simple example to show how the interactions work in Fractalide. Sub-components understand the same messages, and transfer them to the internal ports. There is no distinction between the use of a sub-component and the use of a component.

5.5 Use of the asynchronous port options, START and STOP

There are three default asynchronous input ports in every component : *options*, *START*, *STOP*. These three ports are not accessible in the main procedure. They have an instant effect on the state and the component, without the user participation.
The *options* input port is used as receiver for the IIPs. This port accept a record, and the record is adjoined to the existing record. If the component is created with the option record `options(width:20 bg:white)` and the IP `options(bg:red)` is send to the *options* input port, the next state will have as options record `options(width:20 bg:red)`.
The *START* and *STOP* port have the same role as the action *start* and *stop*. As soon as an random IP is receive in the *START* port, the component will try to start by checking the starting conditions. An IP in the *STOP* port will halt the component. This allow control of running components at execution time.

5.6 The failure model

Directly inspired from the OTP Erlang model, Fractalide has a failure model. The goal of the failure model is to handle the execution errors at run-time. The errors come from two distinct point : the main procedures and the bind/unbound of the output ports.
The way these errors are handled is by using a special flow in the graph. Each component and sub-component has a simple output port named *ERROR*. This port is automatically added to every component. When a sub-component is created, every *ERROR* port of the component inside are bound to the external output port *ERROR*. So, when a procedure fails an error is raised, then caught and sent to a specific port. The IP send on the port resembles this:

```
component_exception(name:Name type:Type error:Error entryPoint:EP)
```

```
cannot_bind(name:Name type:Type error:Error entryPoint:EP)
```

The name and the type identify the failing component. The entryPoint is the procedure that authorizes to interact with the component. The *Error* is the exception that has been raised.
These rules allow a good handling of the errors. The errors can be handled globally, if the all graph is considered as a sub-component. All the errors go on a single output port, that can be binded to a supervisor component. Or at every level, the errors can be caught to do more specific things. For example, an error from a component that authorize user entry is more sensible.

A basic failure handler component is `failure/five`. This component takes in errors from the input ports. It send the errors out on the *output* port, which can be linked to a display component if needed. This component also counts the number of errors a component raise. If a component raise more than five errors, the component is stopped.

5.7 The graph library

To load and start an entire Fractalide graph, one uses the graph library. It's a functor that export 3 procedures : *loadGraph*, *start* and *stop*.
The *loadGraph* method serves to transform a *.fbp* file in a record describing the graph. This record is used to keep the entry point to each component. The *start* and *stop* procedures simply send this action to every components composing the graph.

The simplified graph code uses load and execute a graph is:

```
declare
[Graph] = {Module.link "./lib/graph.ozf"}
G = {Graph.loadGraph "path/to/file.fbp"}
{Graph.start G}
```

5.8 The QTk cards

The QTk library of Oz is ported in a FBP/card style. The widgets are implemented as cards. The cards are black-boxes which manage the state of the widget. In QTk, the state is changed gracefully using a *handler*. The handle has getters and setters. For example, it's possible to modify the text on a button. QTk cards transform the actions in methods understood by the handler. If the handler doesn't understand it, the action is send out to respect the hierarchical flow of actions. All the possible actions are describe in the Mozart Oz documentation of QTk.
There are two special actions a QTk card understands, and that the handler doesn't : create and getHandle.
The *create* action is used to create the widget. It is send in on the main input port *in*. At each time it is received, the card create a new widget, a new handler. An *create* action is sent on the output port *out*. This serves to create the next component, for example a *td* widget inside a *window* card.
The features/values pairs in the record are used as parameters for the creation. To create a QTk button, without card, the record `button(text:"hello" handle:H action:proc`

`... end)` is build. For a QTk card, the IP sent to the *in* port `create(text:"hello")`. The handle and the actions features are managed by the card.
The *getHandle* action is used to get the handle of the widget. The handle will be sent in a new action. The name of the action is precise in the *outside* argument. If the widget receives `getHandle(output:getForExample)`, it will send the action `getForExample(Handle)` out.
All other actions are used to interact with the widget. The actions are send on the same port. There are two types of actions : ones that return no value or the ones that return a value.
For the actions that return no information (no variable is bounded), there is no distinction between the record send to the handle and the action record. IE : `set(bg:white)` or `setCoords(10 100)`. For the get action, the record is modified. For example:

```
get(bg:BG text:Text)
```

on the handle, the action is

```
get(bg text output:anOutputName)
```

The component will send the action `anOutputName(bg:white text:"foobar")`. The arguments must be given, like this: `canvasx(1:10 2:Y)`. An special field *arg* must be added with the arguments. So :

```
canvasx(2 arg:arg(1:10) output:getCanvasX)
```

which give `getCanvasX(1:10 2:110)` as result.

If an actions arrives into a card before the creation, they are put in a internal buffer. This buffer is read once the widget is created, to apply all actions.

Therefore, all widgets have the main ports: *in*, *out*, *action*. Some of them have special input ports, to facilitate their specific features. The files testWidgets.oz and testCanvas.oz show the examples of specific behaviours of each.
The following cards have no special ports, all the actions are send in the *in* input ports : button, checkbutton, entry, label, message, numberentry, radiobutton, text, window. Their approach is very similar to pure QTk widgets.
All the following cards have specials ports. These ports are always asynchronous. The input port *in* is used to manage the main handle. Other ports are used to manage other widget that are inside the desired one.
The listbox and dropdownlistbox have another input port called *list*. It receives a list of Oz virtual string. It's the normal init feature. So instead of `dropdownlistbox(init:["foo" "bar"] glue:ns handle:H)`, it is a graph like

```
'create(glue:ns)' -> in ddlb(QTk/dropdownlistbox)
'["foo" "bar"]' -> list ddlb()
```

The *td* and *lr* cards both have asynchronous array input port named *place*. This port manages the widgets inside the card. The specification of this array ports are numbers, corresponding to the place of the widget in the *td* / *lr* card. Instead of `td(label(init:"foo") label("bar"))`, the graph is:

```
'create' -> in td(QTk/td)
'create(init:"foo")' -> place#1 td()
'create(init:"bar")' -> place#2 td()
```

These cards are also fully dynamic. So it's possible to replace the content of a place by another widget, by resending a create event + arguments. Also, the number of place is not fixed.

The *grid* card work in a similar way, but it's an asynchronous array input port and is named *grid*, it receives the two coordinates of the inside widget. For that, the specification takes the form of "XxY", so the X coordinate, the letter 'x' and the Y coordinate. So in classical QTk, the record can look like `grid(label(init:"foo") empty newline empty label("bar"))` and the graph is:

```
'create' -> in grid(QTk/grid)
'create(init:"foo")' -> grid#1x1 grid()
'create(init:"bar")' -> grid#2x2 grid()
```

The *panel* card is similar. The *in* port and an additional *panel* are asynchronous array input ports. The specification is a number, and the panels are in the order of these number. The QTk record is `panel(label(init:"foo" title:"panel 1") label(init:"bar" title:"panel 2"))` is replaced by the graph

```
'create' -> in panel()
'create(init:"foo")' -> in label1() out -> panel#1 panel()
'create(init:"bar")' -> in label2() out -> panel#2 panel()
label1() action#create -> in addTitle1() out -> panel#1 panel()
label2() action#create -> in addTitle2() out -> panel#2 panel()
```

The created records goes out of `labelX()` and doesn't contain information about the title of the panel. So the action must be caught and edited to contain this information. Again, this card is fully dynamic. So you can add panel during the execution at any place. If the place is already taken by another widget, it is deleted and the new one replaces it.

The *placeholder* card has a *place* input port. The children cards must be connected to the *place* port. When they send their create event, they are displayed in the placeholder. Once created, they can be displayed with the help of the *display* action.

```
'create' -> in ph(QTk/placeholder)
'create(text:"b1")' -> in b1(QTk/button) out -> place ph()
'create(text:"b2")' -> in b2(QTk/button) out -> place ph()
'display' -> in b1()
```

This will initialize the two button to be displayed in the placeholder. When the button *b1* receive the *display* action, so it will be set in the *placeholder*

The *scrollframe* card has a *frame* input port. This input port receives the card that must be displayed in the frame.
As the QTk scrollframe is not dynamic in the sense the frame cannot be changed once the handle is created, the *create* event of the *in* port and the *place* event is synchronized. That imposition to recreate a new widget each time must be changed.

```
td(QTk/td) out -> frame scrollframe(QTk/scrollframe)
"create(tdscrollbar:true glue:nswe)" -> in scrollframe()
```

The QTK image card is a special QTk component because it's not a card. It receives a virtual string containing the path to the image on the *input* input port. It then sends the TK image to the *output* output port.

The QTk canvas card has the same logic than the other ones. The widgets that want to be displayed on the canvas send their actions in the *widget* input port.
All canvas items are placed in the sub-directory *canvas*.
The *canvas/arc*, *canvas/bitmap*, *canvas/line*, *canvas/oval*, *canvas/polygon*, *canvas/rectangle*, *canvas/text* have just the *in* input port, as a normal card.
The *canvas/image* had the additional *image* asynchronous input port. The image created with the card *QTk/image* are send in this special port to be displayed.
The *canvas/window* had the additional *window* input port. It works like the *scrollpanel* card. So the *create* action are synchronized with the two input ports, though the other actions are asynchronous. Again, that came from the fact that the Oz QTk library doesn't accept dynamic changes of the window inside this widget.

5.9 Examples

This section give the basic notion of how to use Fractalide in general, making use of the card concept. The first example will be the complete calculator example given in the introduction. Then a little example will show the dynamism of the system. A third example will explain the hot-swapping feature.

5.9.1 The calculator

Now that all the features are explained, the complete graph of the calculator example can be given. As a reminder, the calculator is show in figure 1.1. The goal is to create a new card for this application. This card can be connected to other UI components to be displayed and interact with it.

First, we need the basic items. The QTk library of Mozart Oz has been ported to Fractalide cards. In the example here, we are going to use *entry*, *button*, *lr*, *td*. The first is the field, the second the button, the third and the fourth are used to make the geometry of the card. The *lr* widgets place the contained widgets from left to right, and *td* does the same from top to down.

Figure 5.5 is a graph that focuses on the UI, several things are added to allow a comparison to the basic one given in the introduction. The main structure is the same, with the four main items and the two layout managers. The first change is the port names. The card that manages the layout receives the widget on an array port. It's an easy way to manage the order of displaying the items, without care of the order in which the IPs arrives. So for example, the button *out* port connects to the array input port *place#2* of the *td* component. Another thing that are added are the IIPs. Each card must have an IIP for the creation, specifying the width, the text, ...
This example also hints at the intuitive nature of FBP modularity and reusability. As everything is a card, one can easily replace an entry with a button, or indeed with any card. So it's possible to incorporate a complete application into another card, by keeping interaction with actions.

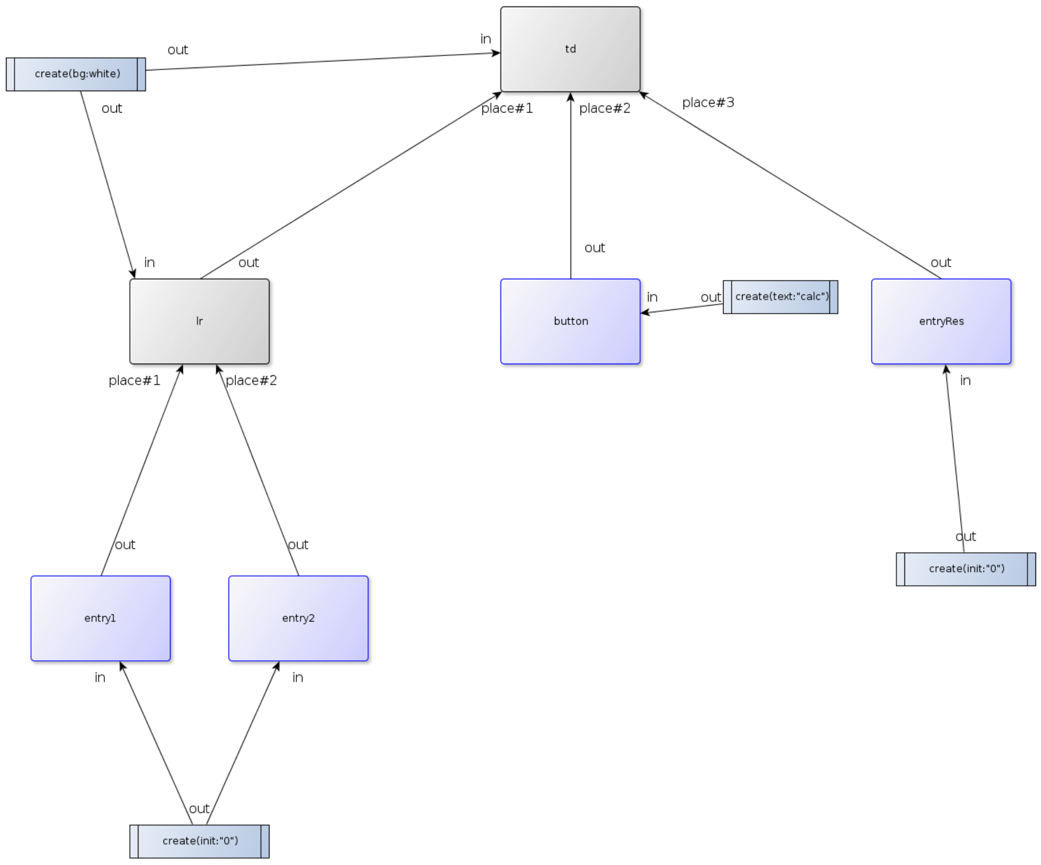


Figure 5.5: The ui graph

The *logic* of the application is shown in figure 5.6. Again, the graph is nearly completely identical as the one show in the introduction. A component has been added between the button and the entry to transform the IP into a "get" IP. When the entries receive this IP, they send their content inside a specified action. What is important, is to see that the action *button_clicked* is caught from the button card. It sends an IP to the two entries containing the numbers to the add component. Once the entry receive this IP, it sends the content out in an action called *getNum*. The actions are caught and sent to a logic component *adder*. This component is just a simple FBP component, and technically shouldn't be classified as a card, it take the two inputs in an array port and compute the sum. This result is send to *entryRes* to be displayed. The result must be put in an action to be understood by *entryRes*.
Actions may be caught at different levels in the hierarchy. If the sum must be updated each time, one number is changed in an entry box, it's easy to do. The action *KeyPress*, corresponding to an key of the keyboard pressed, can be caught at the *lr* card. So, all the *KeyPress* events generated by *entry1* and *entry2* are send to *lr*. Then *lr* send them out, and where they are caught. In a similar way, the Enter key can always press the button. It's is possible to catch this event at the *td* level. All the actions arrive in this card, so all the Enter key press will go through it. This hierarchy allow a fine-grained control of how the applications react.

Here, the *ui* graph and the *logic* graph are separated to show the flows clearly, but these two "different" graphs are actually one in the card. To complete the card, we need to add the exterior ports *in*, *out* and *action*. It's not always very clear toward

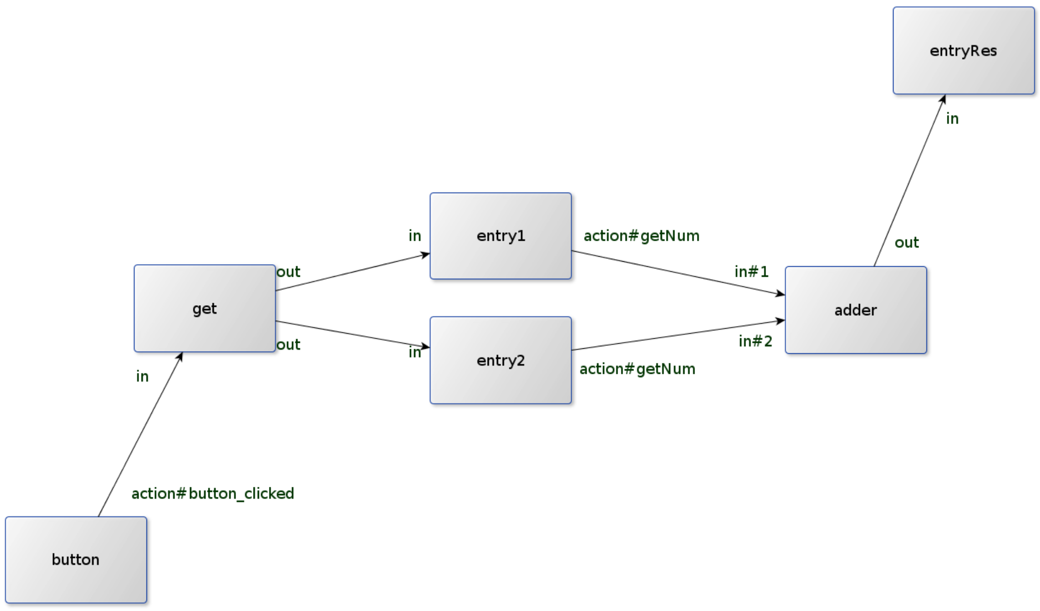


Figure 5.6: The logic graph

which ports they must be linked. The *out* and *action* virtual port can be the one of the *td* card, but what about the *in* port? A good habit is to make use of a special card *actionWrapper*. This card does nothing with the actions it receive, only transmit it on the correct output port. It is very useful to make a new card, it serves as entry point and exit point, making the actions explicit. It is shown in figure 5.7.

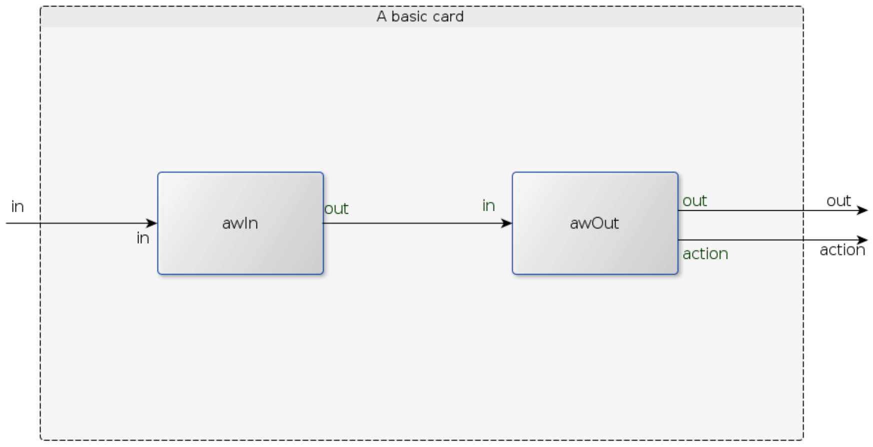


Figure 5.7: a basic card with input and output

We can now easily use the default flow of actions. The *out* port of the *td* card will be bound at the *in* port of the *awOut* card. So, all actions of the *ui* will be send outside. The *in* port have no utility for this card, but in fact, it's easy to imagine that the calculator will understand the action *makeAdd*. Then, the action *makeAdd* will be caught just after the *awIn* card, to do the same logic as the button. The entire graph of cards is in the Annexe "Calculator Card".

The only specific application component here is the *adder* component. It's interesting to note that all cards save for the adder already exists and doesn't require a developer. This is the power of reusability in FBP. The average user will be able to easily link and design logic. A key will be the sharing of these complex Oz cards, build by hackers/wizards, to everyone. As the Wolfram language propose a huge set of tools to compute everything, the goal is to build a huge library of cards to respond to a maximum of demand.

5.9.2 The dynamism

It's hard to show dynamism in a paper, so I encourage you to test this example in the *Editor*. The installation instruction are in the Appendix Install Fractalide.
This example will begin by displaying a single button in a window. For that, three components are needed : *QTk/window*, *QTk/button*, *getopt*. The two first are used for the ui, and the last one is use to generate the IIP of the button. The figure 5.8 shows the result. The output port of the *genopt* component is linked to the *in* port of the button. The *out* output port of the button is linked to the *in* input port of the window. The parameter for the *IIP button* component is `create(text:"test")`. When the IIP is fired by selecting the *genopt* component and selecting start, the button receive the IP. It creates itself and send an IP to the window. This component will create the window with a single button. Click on this button will do nothing, as the action *button_ clicked* is send on the *out* port and go inside window component, which will drop it. Before generating the IIP, be sure the button is connected to the window, with the correct ports. Due to a design error of the QTk component, they will block if their creation action doesn't finish in a QTk window. See Github issue for more details.

Now, without closing this window, the action *button_ clicked* will be caught and displayed in the console. For that, a new component of the type *display* is needed. It is linked from the *action#button_ clicked* output port of the button component to the *input* input port of the new one, as shown if figure 5.9. As soon as the link is done in the editor, the button will send this action on this port. A single click will be shown in the console `button\_clicked`. The important thing is that the IP is now sent on the good port. The interface was not restarted, this is exactly the same button and the same window. That is the dynamism of Fractalide. For example, to see all the events sent by *button* to *window*, change the *action#button_ clicked* port by *out* will be enough. Here is the result :

```
’’#’Motion’(state:[48] x:13 y:10)#’’
’’#’Motion’(state:[48] x:13 y:11)#’’
’’#’Leave’(detail:[78 111 116 105 102 121 65 110 99 101 115 116 111 114] focus:1
’’#’Enter’(detail:[78 111 116 105 102 121 65 110 99 101 115 116 111 114] focus:1
’’#’ButtonPress’(button:1 x:13 y:11)#’’
’’#’ButtonRelease’(button:1 x:13 y:11)#’’
’’#button_clicked#’’
’’#’Motion’(state:[48] x:15 y:10)#’’
’’#’Motion’(state:[48] x:19 y:9)#’’
```

We see that there are many messages that are generated by the button. That can be useful to find new one, or to debug the application. Now, the *display* component can just be deleted, and no more IP will be displayed.
This example demonstrates the dynamic nature of Fractalide. The interesting point is that this is not a complex implementation. No code has been added to enable these features. When a link is created on the editor, the only thing done on the real graph

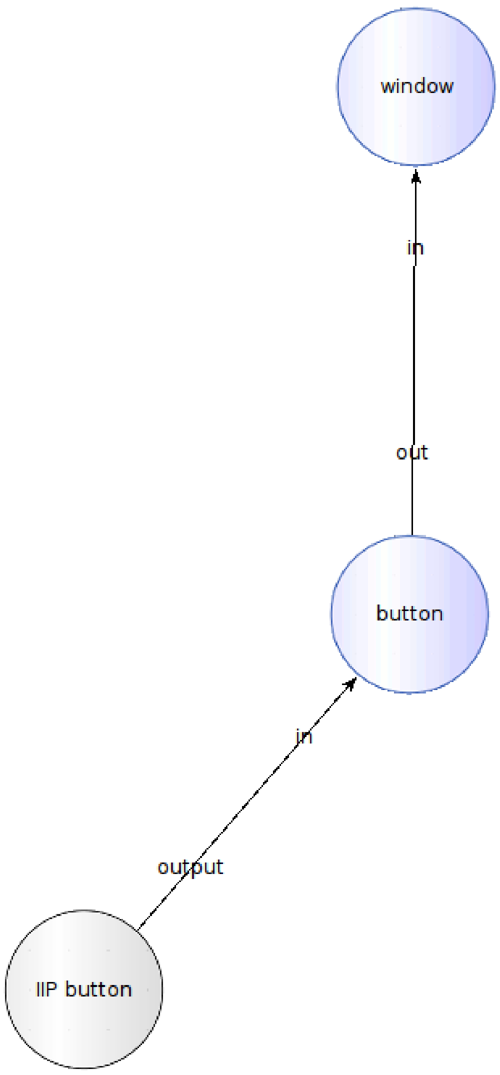


Figure 5.8: A single button

is the link, and no more operations are necessary. Creating a component doesn't need to make operation on a global graph of something like that. It's so easy due to the dynamism of the implementation of FBP, due to the dynamism of the components and sub-components, on which the entire Fractalide system is built.

5.9.3 Hot-swapping

Let imagine a situation to demonstrate hot-swapping capabilities of Fractalide.
Imagine that Alice had a little socket server build on Fractalide. It's a very simple one, the user send *string* on the socket, the "server" reverse it and send it back on the socket. To make this example simple, the socket server accept only one connection, so the graph is limited at three nodes : *socketReader*, *reverser*, and *socketSender*. They are connected as shown in figure 5.10. The *Reader* sends the packet receive on the socket on the *output* port, they are reversed in the *Reverser* component and then sent out with the *Sender* component.

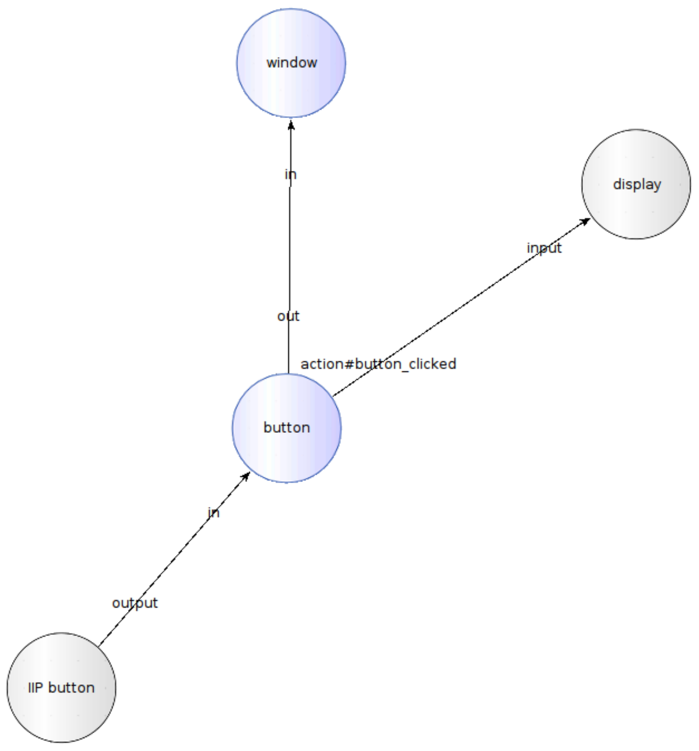


Figure 5.9: Display the *button_ clicked* action

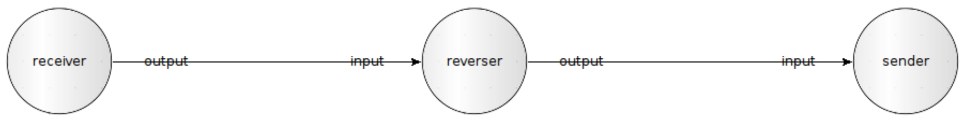


Figure 5.10: Hot-swapping example

Imagine now that Alice realises she made an implementation mistake in the *Reverser* component. As her client is currently connected, she wont want to close the connection. More over, the client uses the component regularly. So Alice stops the component to fix the bug. The client can send a new request, it will be put in the buffer of the *Reverser* component. Then, the main procedure is replaced without changing anything else in the graph. The component still exist with their link. The *Reverser* is restarted, and the IPs in the buffer are feed. The client now has his correct response, and the socket connection was not lost during the modification.

What is interesting here is that the main procedure of component was changed in a running graph in again a very simple manner. The *Receiver* and *Sender* component were not aware about this change, even in they are directly connected to him, so their execution were not influenced. There is no side-effect by doing such an operation on a node, unless of course you delete connected port. Also, all these changes are made in a consistent manner (see Implementation). If the *Reverser* component was not stopped, all the executions will be correct, as soon as the implementation is correct. It is either the old implementation or the new one, but no strange mix of the two. This feature is very important to make living application in a distributed manner.

5.9.4 A new card : Stack

This example will implement a basic stack card, and mimic the behavior of a Hyper-Card stack, albeit a simple one, that doesn't have database features. Several cards will be place in, in an ordered manner. It will then possible to go to the next card, to the previous card, or to a specific card.

The card will have two input ports : *in* and *card*, like the *lr* and *td* cards have. It will have the two classical output port *out* and *action*. The input port *card* will be an array, where the specification will determine the order of the card in the stack.
The card will understand three special actions linked to the stack behavior : *next*, *previous* and *goto*. The specification is easy : *next* go to the next card, *previous* to the previous one and *goto* go to the card specified in parameter. A basic utilization is show in figure 5.11.

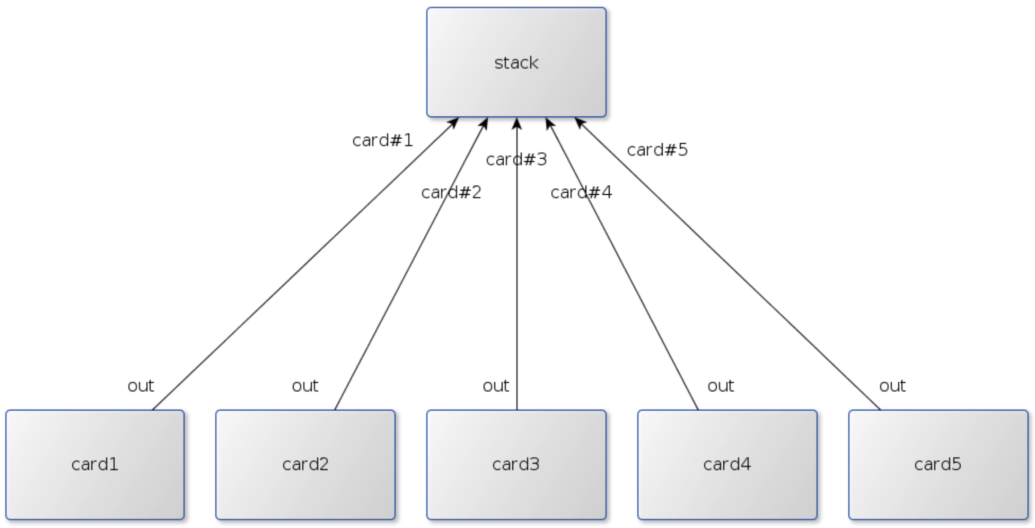


Figure 5.11: stack example

The graph of this card is :

```
in => in stack(stack/stackManager) out -> in ph(QTk/placeholder) out => out
card => card stack()
ph() action => action
```

The card will be displayed in a placeholder. As the stack needs more control on which sub-card is displayed than a placeholder, a manager is put to filter the incoming messages. This is a good example of the extension capabilities of FBP. This system is used instead of inheritance in OOP. The manager will catch all actions *next*, *previous* and *goto*. It will also catch the *create* event that arrives in the *card* input port. As the creation of the stack is the same as the creation of the placeholder, the *create* action receive in the *in* input port can just be send to the placeholder. The creation of the stack will accept the same parameters as the creation of the placeholder.

```
functor
 import
    % The path to the component library
    Comp at '../lib/component.ozf'
 export
    CompNew
 define
    fun {CompNew Name}
       % R is the initial record
       R = component(name:Name type:stackManager
                     description:"maintain␣a␣stack␣of␣card"
                     inPorts('in')
                     inArrayPorts(card)
                     outPorts(out)
                     procedure(proc{$ Ins Out Comp} IP in
                                  ...
                               end)
                     state(stack:stack() actual:0)
                    )
       % The component library is called to create the component
       {Comp.new R}
    end
 end
```

Figure 5.12: stackManager component

```
"create(bg:white glue:ns)" -> in stack(stack/stack)
```

First look at the structure of the *stackManager* card, figure 5.12, without going in the implementation of the main procedure.

The card will maintain a state variable *stack*. It will be a record, where the features are the index of the card, and the values are the card. When a card is sent the create action, the record is saved. This will be used to display the card inside the placeholder. The *actual* variable is used to remember which card is actually displayed. It's important for the relative movement *next* and *previous*.

Lets build the procedure for the *card* array input port as in figure 5.13.

The procedure loops over all the selections. The condition is used to make the port asynchronous. If the procedure tries to read without checking if an IP is inside before, the port will be synchronous, as the get operation blocks the execution. Here, an IP is only read if there is an IP. The *create* action is a handle to save the IP in the stack. All the other actions are managed by a common procedure, figure 5.14, of *in* and *card* input port.

When a card is displayed, the action *set* with the card's handle must be send to the placeholder. If it's the first time the card is displayed, the handle doesn't exist yet. In this case, the entire record is sent.

The main procedure can now be implemented, figure 5.15. The two ports are asynchronous.

This is the construction of a simple stack card. It can be improved by allow dynamic

```
proc {CardProc Ins Out Comp}
   proc {Card Index} IP in
      IP = {Ins.card.Index.get}
      case {Label IP}
      of create then
         % save the creation
         Comp.state.stack := {Record.adjoinAt Comp.state.stack Index IP}
      else
         {ManageIP IP Out Comp}
      end
   end
in
   % All selection work in an asynchronous way
   {Record.forAllInd Ins.card
    proc{$ Index Buf}
       if {Buf.size} > 0 then {Card Index} end
    end
   }
end
```

Figure 5.13: card input port procedure

adding of card, replacing of existent one, deletion, ... But the logic of the card is there, and shows how to work asynchronous port and composition.

The use of the stack is showed in the file *test.fbp*. The interesting parts are:

```
% First step : create the stack
stack(stack/stack) out -> place#1 td(QTk/td) out -> in win(QTk/window)
win() out -> input disp(display)
% put cards in the stack
calc1(calculator/calculator) out -> card#1 stack()
calc2(calculator/calculator) out -> card#2 stack()
calc3(calculator/calculator) out -> card#3 stack()
"create()" -> in stack()
```

This graph created a stack and placed it in a window. There is a *td* card to display button under the stack. Three cards are added to the stack. It's three card of the same kind, but that is just one example. The last line creates the stack, without any parameter for the placeholder. See that the cards placed in the stack are the calculator. It's a good example of reusability!
Then, buttons are added.

```
% next
nextButton(QTk/button) out -> grid#1x1 grid()
nextButton() action#button_clicked -> input next(ipEdit) out -> in stack()
"opt(text:next)" -> options next()
"create(text:"next calculator")" -> in nextButton()
```

The button is placed in a grid. When the button is pressed, the action is transform in “next” and send to the stack. The other buttons works in the same way. Notice

```
proc{ManageIP IP Out Comp}
  proc {Change NIndex} Card NIP in
    % Select the card at the index
    Card = {List.nth {Record.toList Comp.state.stack} NIndex}
    % Check if the first display
    NIP = if {IsDet Card.1.handle} then set(Card.1.handle)
          else {Record.adjoin Card set} end
    % Display
    {Out.out NIP}
    % Save the new position
    Comp.state.actual := NIndex
 end
in
 case {Label IP}
 of next then
   % Don't go over the max width
   {Change (Comp.state.actual mod {Record.width Comp.state.stack})+1}
 [] previous then NI in
   % Don't go under 1
   NI = if Comp.state.actual == 1 then {Record.width Comp.state.stack}
        else Comp.state.actual - 1 end
   {Change NI}
 [] goto then
   {Change IP.1}
 else
   % The unhandled actions are send to the placeholder
   {Out.out IP}
 end
end
```

Figure 5.14: manageIP procedure

```
procedure(proc{$ Ins Out Comp}
             if {Ins.'in'.size} > 0 then
                {ManageIP {Ins.'in'.get} Out Comp}
             end
             {CardProc Ins Out Comp}
          end)
```

Figure 5.15: main procedure

that the actions *next*, *previous* and *goto* can come from a card. If the calculator generates any of these actions, the stack will react on the same way. Also, the manager works transparently for the placeholder. If a card sends an event understood by the placeholder, like `set(bg:white)`, it will be proceed correctly.

Chapter 6

Editor : an application in Fractalide

This section details a more permanent application of Fractalide, that of a visual graphic graph editor. This is the main non-trivial proof of concept tool to demonstrate Fractalide's powers of composition, event handling and dynamic behaviour as well as serving as an initial concept stage HyperCard implementation for future user modification. Our approach was heavily inspired by Emacs, inthat each Emacs key combination can be extended by elisp, so each of Fractalide's features is extended in Mozart Oz.

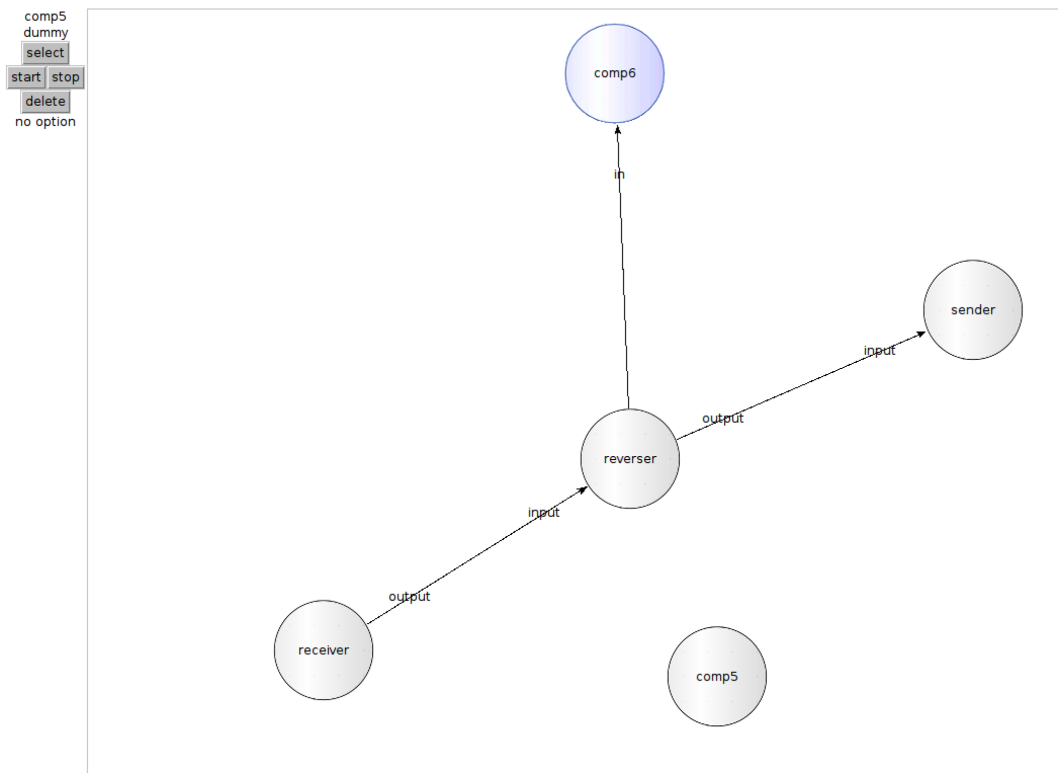


Figure 6.1: The Fractalide editor

Our goal is to implement a user interface similar to the one in figure 6.1. The main screen is divide two parts. The right section allows the user to manipulate the FBP graph topology; make links, move components about. The left panel allows for component and link modifications on the graph. For example, rename a component, add a component on the graph, or any other operation that the Fractalide library permit. Named the "graph panel" and "edit panel" respectively, they have precise

specifications that will be elaborated on by means of an example. This graph editor has been implemented and works according to specification, see the annex “Install Fractalide” to install it.
The first section will give a general overview of the program. It will not go into details, but the explain the different flows. Shortly after, a section of the application will be explained in detail. The whole graph application will not covered fully. Some specific and complex parts will be elaborated on.

6.1 Global overview

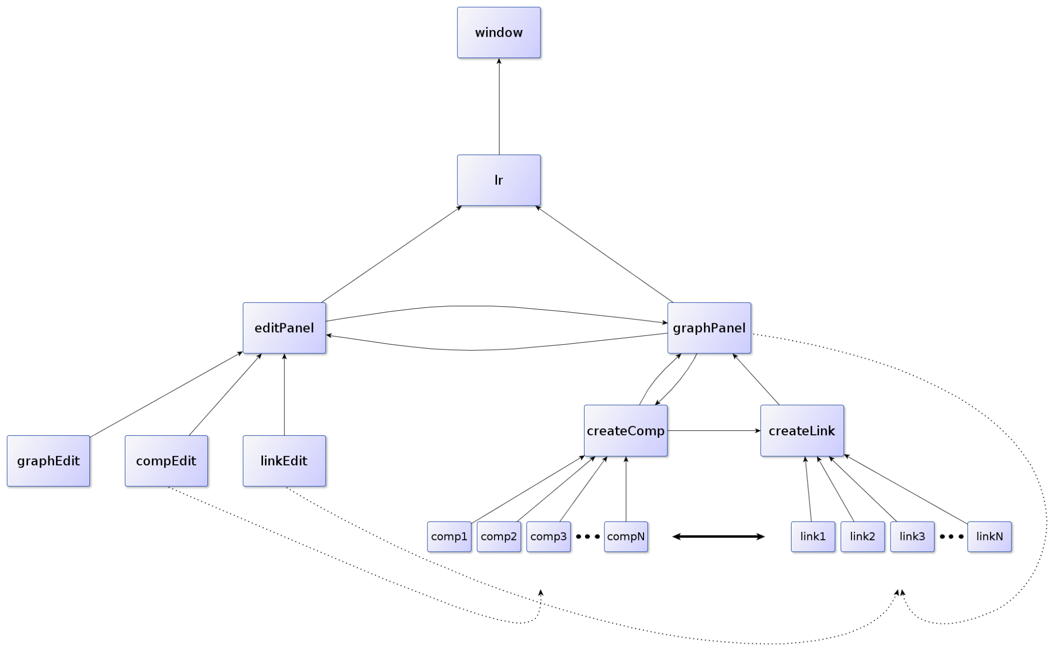


Figure 6.2: Fractalide overview

Figure 6.2 shows an overview of the whole Fractalide hierarchy. The top node is a *QTk/window* card. It displays a QTk card descendent in a new window. The second node is a *QTk/lr* card. It displays children from left to right. Then there is major branches. The left one corresponds to the *edit panel*, the right one corresponds to the *graph panel*. The *edit panel* is a *QTk/placeholder*. One of its children can be display at the time. The *graph panel* is a canvas. Its children generate canvas items that will be displayed on the canvas.
These two panels interact with each other. The *edit panel* sends an action to the *graph panel* when the user want to create a new component. The *graph panel* sends actions to the *edit panel* when the user want to edit a component, a link or the graph.
The right branch has a canvas higher up in the hierarchy. The canvas’s responsibility is to display objects pertinent to graph composition, namely components and links. These objects are dynamically created. The canvas has two major branches as children. The left manages the component objects, the right manages the link objects. The *createComp* and *createLink* cards modify the layout of the graph at run time. The output ports of these new cards are linked to the components that created them. Accordingly the generated actions will escalate up through the hierarchy. There is an interaction between component objects and the link objects. When a component is

moved, a side of the link must also adjust position. For that, the component sends an action over the *move* port detailing specific location it needs to redraw to. Also, the port names that the link bound together are given to the component by the link. To resume, this branch must generate component objects from an external action and generate link objects internally. These objects are coupled because their behavior are well connected.
The dotted lines illustrate messages that are sent to components, but there is no real link in the graph. The emitter directly sends the IPs to the entry point of the receiver, and not to an output port. The reason for this is that they send the actions to only one of the component objects or links, not always the same. It's a dynamic link. The *graph panel* sends actions to the link objects when only one of the two side is attached to a component. The other side attaches to the mouse pointer, and the position of the mouse is sent by the canvas. The actions that came from the edit panel are used to change values of the receiver like the name, component options and type of a component.

A first drawback of the Fractlide paradigm soon appears in the get methods of the graph editor. It's rapidly becomes heavy and rather clumbersome. Imagine an application must known the position of the mouse on the canvas. Not in the window, but in the canvas once it has been scanned. It is not the case, but is good for the example. The component object will generate an action like "get(position output:getPosition)". This actions must go through *createComp*, then into the canvas component, then the output action must be caught and go back down in *createComp* again and finally into the component. This stream is a little complex, and make a lot of process for an simple read.

6.2 Explore one branch

This section explore in deep one branch, from the top node until the component object cards. This will allow a better comprehension of how to work with the Fractalide paradigm.

This main sub-component of this application is the top four nodes. The corresponding graph is shown: 6.3. It's not implemented as a card, as this sub-component must be displayed in a new window and not send actions up a level up. This can be easily transform later. The top node is the QTk window, which receives the entire construction. The node just below is a left-right layout manager, to handle the two panels of the application. The left one is the card *editPanel* and the right is the card *graphPanel*. The hierarchy shown in the overview is in fact inside these cards. There is an *IIP* to create the *lr* card, one to create the *editPanel* and one to create the *graphPanel*. These two last IIPs serve to integrate the panels into the main application, by setting the glue, background and other parameters. The two panels are clearly coupled together as they send IPs to each other. Here is the actions they share :

```
% ---
% Communication between the two
graphPanel() action#displayGraph -> in editPanel()
graphPanel() action#displayLink -> in editPanel()
graphPanel() action#displayComp -> in editPanel()
editPanel() action#newComp -> in graphPanel()
```

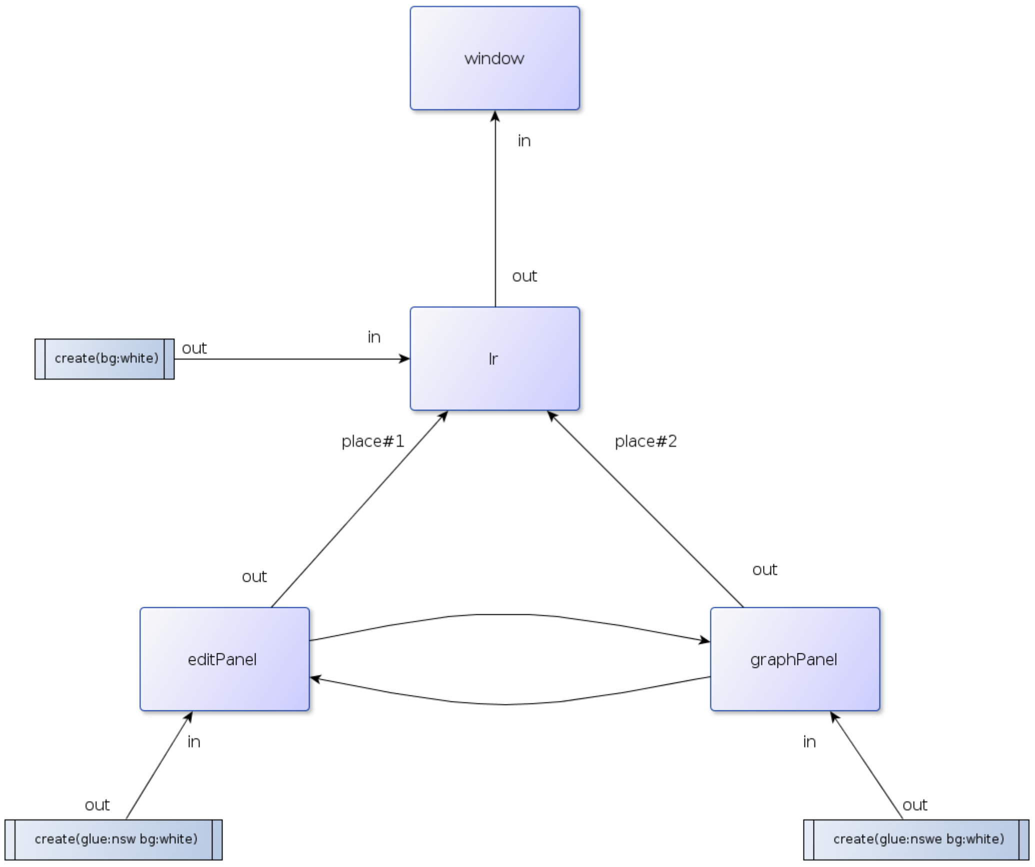


Figure 6.3: The main component

The actions *display...* are used to display a specific item in the *editPanel*. They are generated by a click on the corresponding place on the *graphPanel* : on the background for the *displayGraph*, on a component object for the *displayComp*, on a link object for the *displayLink*. The *graphPanel* can receive the *newComp* action, which adds a new component on the graph.

The graphPanel is a card, represented in figure 6.4. This is a more complex card. The idea is that this card is mainly a TK canvas, which displays component objects and links objects. It's controllable with the mouse. Clicking on the background will display the graph options in the edit panel. Perform a drag and drop on the background will move all items on the canvas, to move the graph. Make a single left click on a component and the object's options will display it in the edit panel. Perform a drag and drop gesture on the component and will the component will move the component object and all the connected link objects on the canvas. The right click on the component object will be used to create new link. Finally, a single left click on a link object displays it in the edit panel.
The mouse behavior is very important in this application and in this card. This specific card logic handles the component's *mouseLogic*. This component receive all the interested actions relative to the mouse from the other cards. The component that manages the mouse or manage the keyboard are very similar state machines. The kind of actions generated by the mouse and the keyboard are *ButtonPress*, *ButtonRelease*, *Motion*, *KeyPress*, *KeyRelease*. The order of these actions are important. For example if the user press the mouse button then move, it's a drag and drop. If the actions *ButtonPress* and *ButtonRelease* follow each other, without *Motion* between, it is a simple

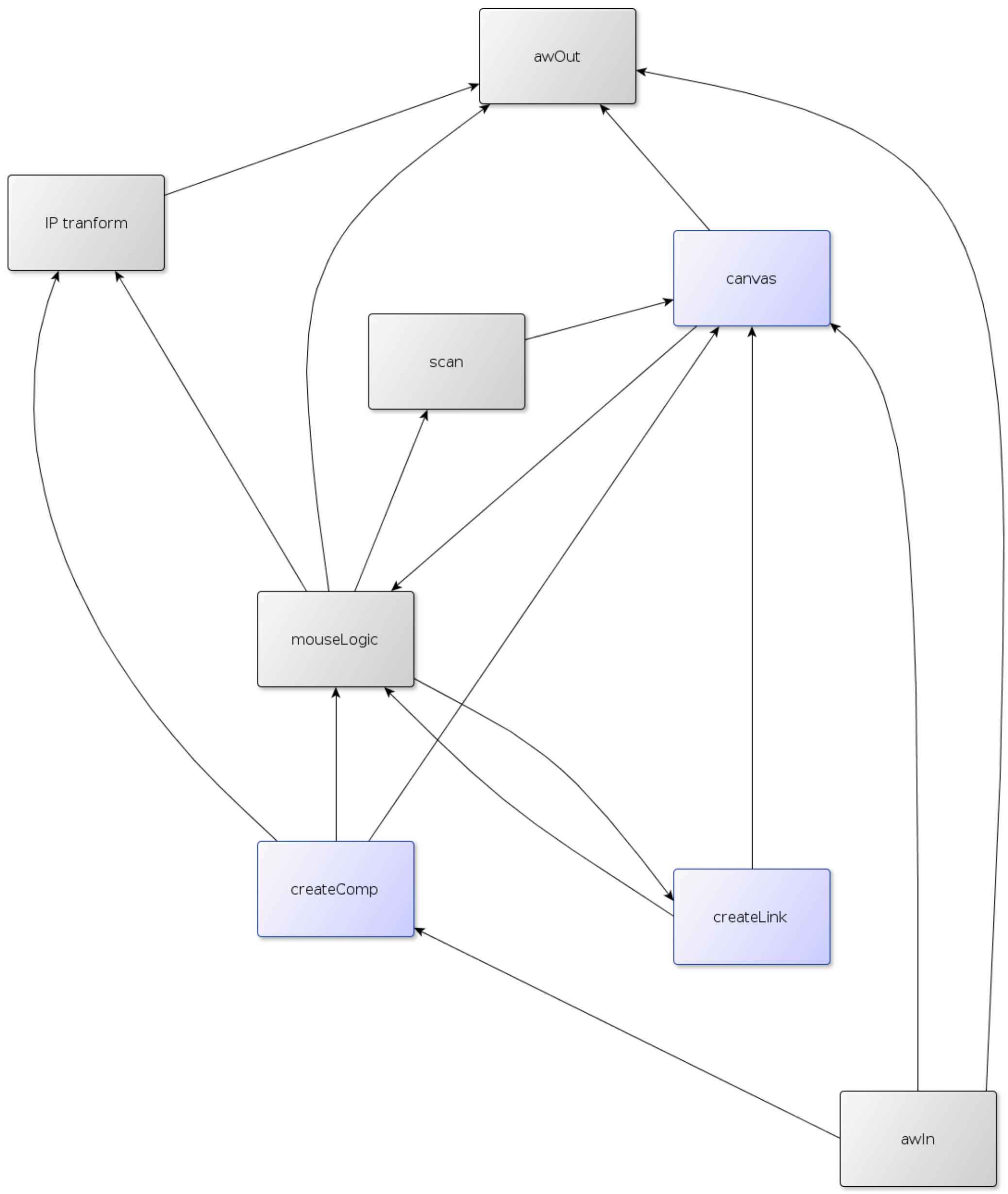


Figure 6.4: The graph panel

click. As example, the code for a drag and drop component is given in figure 6.5.

In the *graphpanel* case, this component is a little more complex. The *canvas* component sends several actions related to the mouse events on the background to *mouseLogic* . The *createComp* send two specials actions : *inObject* and *outObject*. In fact, in QTk, if a click is performed on an item on an canvas, both the item and the canvas will generate a mouse event : *ButtonPress*. This is a problem in Fractalide's logic, so these two events are used to ignore the canvas mouse events when the mouse is over an object on the canvas. The *createLink* send the actions *beginLink* and *endLink* to *mouseLogic*. With all these messages, *mouseLogic* can send the correct messages to one the the three output ports : *drag*, *link*, *click*. The *drag* port send all the IP necessary for the drag and drop logic. *mouseLogic* sends to the *drag* port the *Motion* action, for the *scan* component. This component manages the IPs to send the scanning actions to the canvas.
The *link* port is used to send the motion action to the link that is actually created.

```
component(name:Name type:mouseLogic
          inPorts(input)
          outPorts(move)
          procedure(proc{$ Ins Out Comp}
                        % IP is here an action
                        IP = {Ins.input.get}
                      in
                        % Case on the action type
                        case {Label IP}
                        of 'ButtonPress' then
                          if IP.button == 1 then
                            % Change the state
                            Comp.state.click := true
                            % Save the positions to calculate the delta
                            Comp.state.lastX := IP.x
                            Comp.state.lastY := IP.y
                          end
                        [] 'ButtonRelease' then
                          if IP.button == 1 then
                          % Change the state
                            Comp.state.click := false
                          end
                        [] 'Motion' then
                          if Comp.state.click then DX DY in
                            % We are in a drag and drop gesture
                            % Send the delta
                            DX = IP.x-Comp.state.lastX
                            DY = IP.y-Comp.state.lastY
                            {Out.move move(DX DY)}
                            % Save the positions to calculate the delta
                            Comp.state.lastX := IP.x
                            Comp.state.lastY := IP.y
                          end
                        end
                      end)
          state(click:false x:0 y:0)
         )
```

Figure 6.5: drag and drop component

The *click* output port is used to send a single click out. The implementation of this component is like a small state machine, that retains the actual state : whether it is in the component state, or if in the drag and drop state, etc.

All components and links are created and managed by *createComp* and *createLink* components. The *createComp* component is in fact a kind of database of the component images displayed on the canvas. All events generated by this variable number of cards go out of *createComp*. They must be managed inside other components because they are dynamically created, they are the data inside the IP. It's the same for *createLink*.

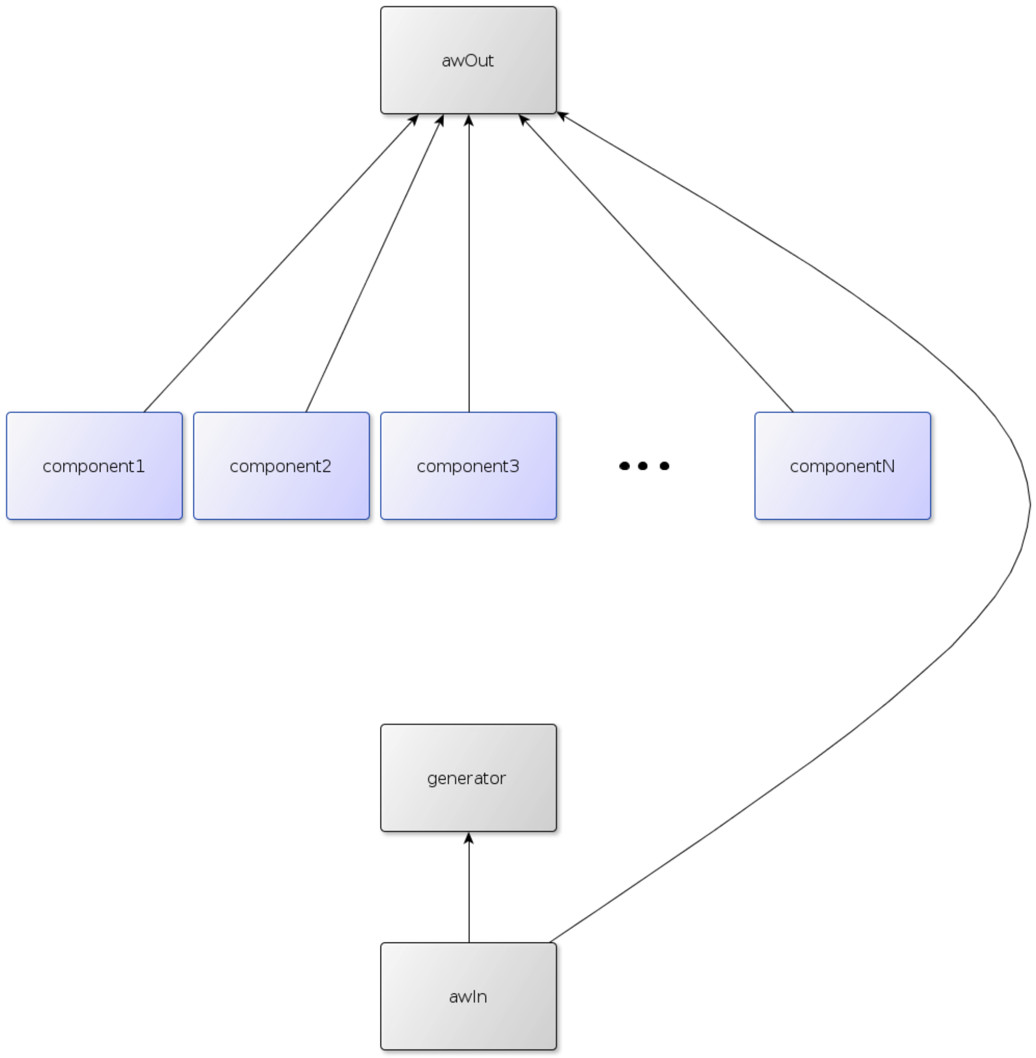


Figure 6.6: The createComp card

Look a little deeper in *createComp* and *createLink* card, that generate the items on the canvas. The figure 6.6 shows the logic of the component one. In fact, this component is not a sub-component but a Oz component, this graph is for clearer explanation. When this card receives the message *newComp* from the exterior, a "generator" creates at run-time a new card of type *editor/component*. This card is bound to the output port of the *createComp* component and later sent to the canvas, *mouseLogic*, and others. It's the same logic for *createLink*.

The three lines from figure 6.7 come from *createComp* and show the creation step. The new component is created, then the ports are bound. The *createComp* has a

```
C = {SubComp.new Name "editor/component"
                      "./components/editor/component.fbp"}
% Bind the port of the new component and start it
{C bind(out Comp.entryPoint input)}
{C bind('ERROR' Comp.entryPoint error)}
```

Figure 6.7: dynamic creation of components

special input port *error* to manage the error flow of the subcomponent.

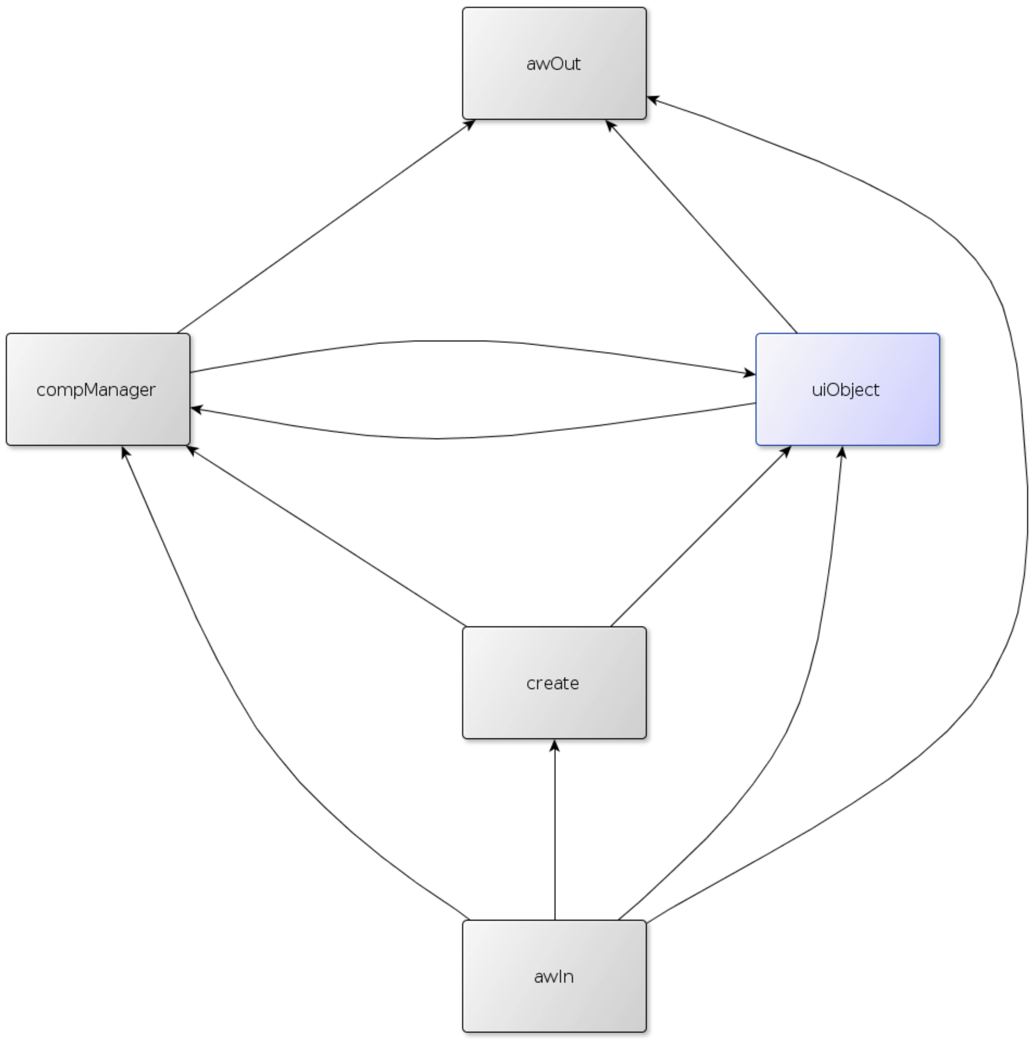


Figure 6.8: The component card

The component *editor/component* the card shown in figure 6.8, is simpler. There are two main components : *compManager* and *uiObject*. The *compManager deals with the real component, the component represented by the circle. The /uiObject* component pertains to the UI; circle drawing, mouse handling, etc. The *create* component serves to create the two other with initial values. *compManager* and *uiObject* exchanges messages. The *uiObject* sends actions *displayObj* and *createLink*. The first is use to display the component in the edit panel. When the manager receives it, it sends a new action *displayComp* outside. This action contains the entry point of the global subcomponent and information about the real component. The second is used when a link is created from/to this component. Again, the information about the sub-component and the component are added to the action. The component send a single action type to the *uiObject*. This action is used to modify the image, allowing to the display of specific images for specific types of component. Also, the *awIn* card sends information from the exterior to these two components. The action *changeName* is sent to these two, as it affects the real component and the ui. *compManager* receives an action to control the real component like *start*, *stop*, *changeComponent*, *setOptions*, . . .

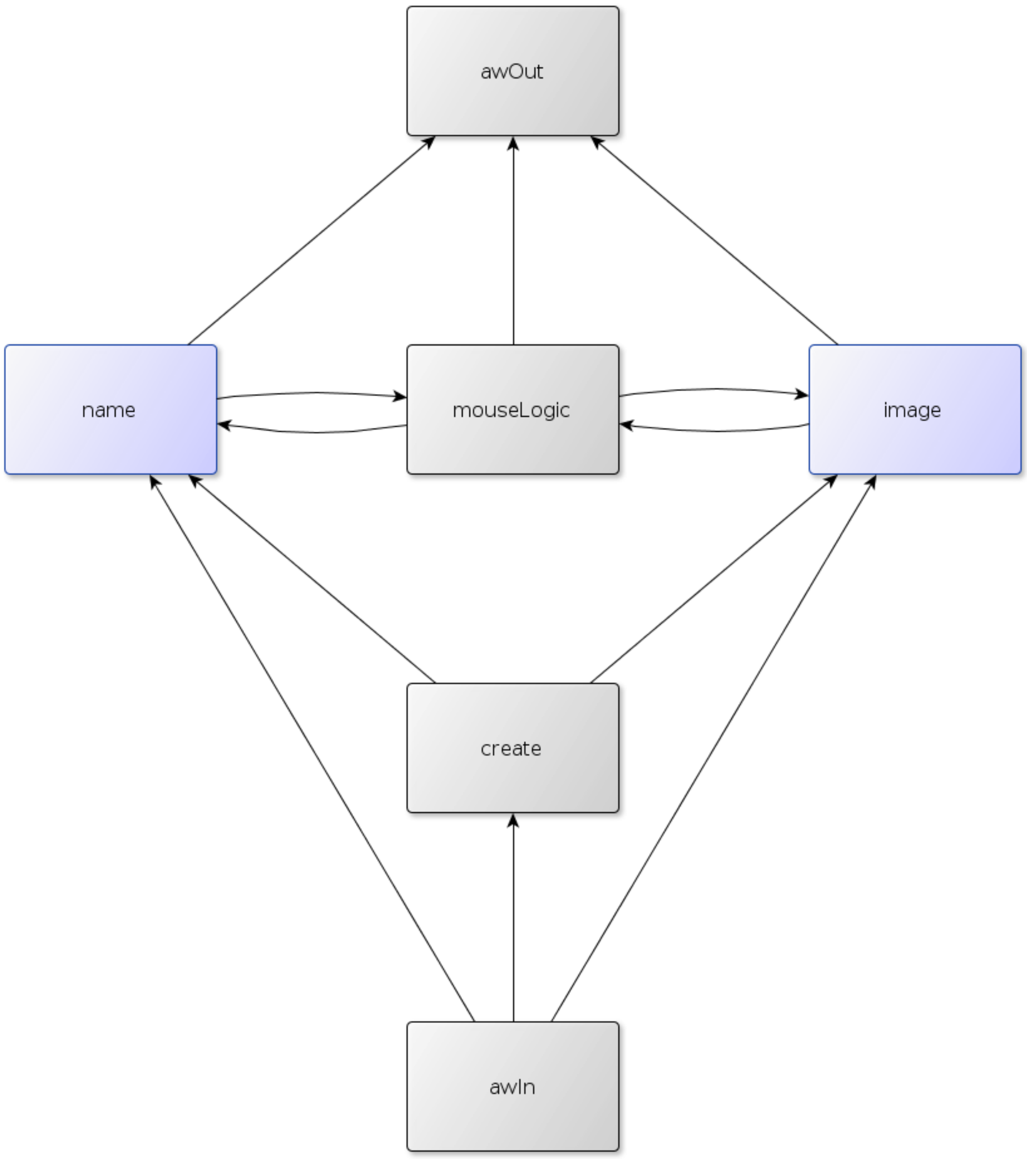


Figure 6.9: The uiObject card

The uiObject card, figure 6.9, has two QTk cards, a creator, a *mouseLogic*, and the two input and output components. The *creator* is used to initialize the two QTk cards. These two cards cards communicate with the *mouseLogic* component, to generate actions like *inObj*, *createLink*, . . . These components are at the bottom of the hierarchy of component objects, it's these items that the user sees, so they manage all the basic actions generated from the user. The implementation of the mouse component is very similar to the one that already exists.

A pattern appears when examining all these cards. The input and output components are always use to filter what is going in and going out. Then there is a component by behavior of the card. If it displays something, there is a card for it, if it manages an object/state, there is a card for it, if there is interaction with the mouse, there is a card for it, etc. Between these cards, it's possible to add smaller components, that are not card, they appear to manage specific actions. For example the *scan* component that transforms a *drag and drop* action in a *scan* action understood by the canvas. Also, many time there is a *create* component. When there is more than one card in the graph, a single *create* action must create several cards, a new component is needed for that.

6.3 The links

The link objects are similar to component objects, but more complex. They bind two real components together. This object draws three items on the canvas : the link and the two names (corresponding to the input and output port names). A single click on a link make it display in the edit panel. Also, the link is related to two real components, and have the output port name and the input port name to manage and display.
The complex part is the position of the line. The two points of the line are managed by the position of the two components objects, they are strongly coupled. If the line is in creation and only one side is attached to a component object, the mouse position on the canvas is the position of the other side. For that, the *link* card has two additional input ports : *moveBegin* and *moveEnd*. These two ports are used to move the begin and end point of the line. When a line is created, it binds the output port of the first component to the *moveBegin* input port, and the output port of the second component to *moveEnd*. They receive *move* actions, that permit to adjust the coordinates.

6.4 The edit panel

The top node of the edit panel is a placeholder. The three children are created at the beginning, and the displayed one at a time.
The *graphEdit* panel is an easy card. It's displayed when the *displayGraph* action is generated. It simply displays a button that generates the action *newComp*. This action leaves to finally arrive in the *graphPanel*. This panel must allow for more actions : global start/stop button, add an external ports, save/load the graph. These actions are the minimum to have a complete FBP graph editor. They will be implemented in future versions.

The *compEdit* and *linkEdit* are more complex. First, they receive parameters in the *display* action. The difference with *graphEdit* is they don't always edit the same object.
The *compEdit* receives the real component to display and the entry point of the component object. The former is used to get the following information: the name, the type and the options. The latter is used to edit the components. For example, if the name is changed, the panel sends an action *nameChanged*. These actions are the dotted links in the overview figure 6.2.

The *linkEdit* card has the same logic. It receives data from the link. The two real components linked together and the entry point of the links. The first ones allow the retrieval of the lists of input and output ports to propose to the user. The entry points are used to signal the selection of one of these port to the link object.
The *linkEdit* card will be explain in more precisely. As the *compEdit* has the same logic and the *graphEdit* is simple, it will be sufficient to understand how they work.

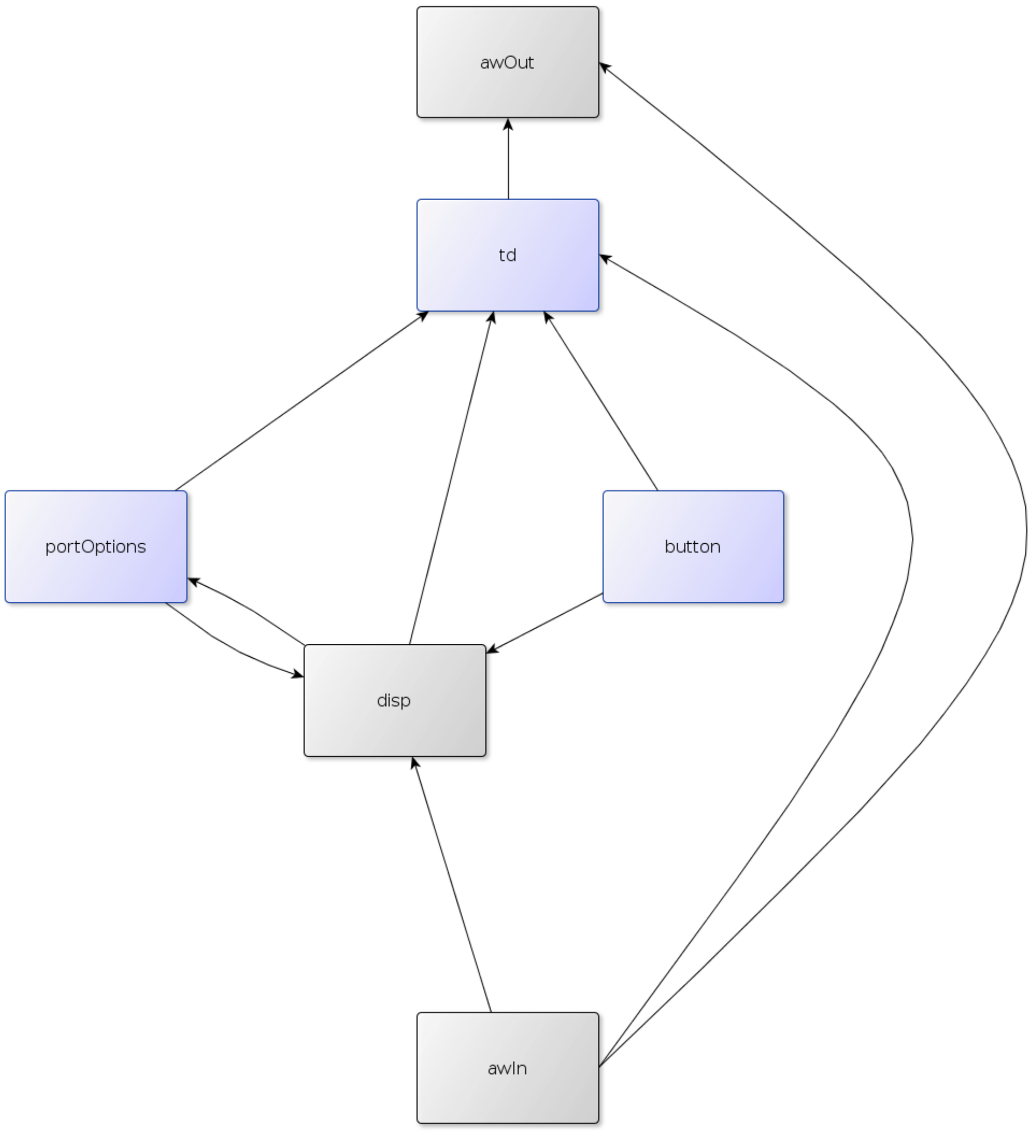


Figure 6.10: The linkEdit card

The *button* and *portOptions* are the displayed cards. The *disp* component is not a card, and it receives the *displayLink* action. It's a state manager. The state, here, is the two real components and the entry point. When the delete button is pressed or when a port is changed, information is given to *disp*, and then an action is send to the link object.

It's the same *button* and *portOptions* for all the links, they are created only once. When a link is selected, the *disp* component sends the available ports to *portOptions*. The card changes the information they display, but do not recreate themselves.

Chapter 7

Implementation

When all is said and done Fractalide is nothing more than a FBP graph which implements a FBP framework. It allows one to build graphs with components, subcomponents and bounded buffers between them. Recall, Fractalide wants components which are autonomous, robust and with hot-swapping features.

The *port object* concept, as presented in the CTM [8], fits our needs quite well. Port objects are conceptually close to Erlang processes. They have a single entry point which receives messages. They can also handle a state which evolves with messages. Having a single entry point is easier to distribute components, but implies to handle the different input ports in the state. A component had the following structure :

```
declare
% Create the entry point
Stream Point = {NewPort Stream}
thread
   {FoldL Stream
    fun{$ State Msg}
        % Treat the messages
    end
    % initial state
    component(name:Name type:Type ...)
   }
end
% The procedure for the user
P = proc{$ Msg}
        {Send Point Msg}
    end
```

The entry point is created with the `{NewPort}` Oz function. All messages are read in a thread. Each loop receives its current component state and returns a new one. The entry point encapsulates a procedure that sends the messages easily and allows the possibility of filtering some messages.

7.1 Component

The state is encoded as a record, which had the structure in figure 7.1.

```
component(name:_ type:_ description:""
          inPorts:'in'() outPorts:out('ERROR':nil)
          procedure:nil threads:nil
          options:opt() state:{NewDictionary}
          entryPoint:_ parentEntryPoint:nil
          run:true
         )
```

Figure 7.1: state : basic structure

The three first features *name*, *type*, *description* are used for component management and not execution. The name and the type are atoms. The description is a virtual string giving a global description of the component.
The *inPorts* and *outPorts* features are nested records containing all information about the ports. They will be explained a bit later.
The *procedure* feature is the main procedure. It's executed when the starting requirements are met. The *threads* field is the current execution thread. It allows to say if the previous execution is over and to stop it.
The *options* field is a record that represents the IIP. The *state* field is a dictionary which is the component internal state.
The *entryPoint* feature is the procedure that gives access to the stream. If the component is run inside a sub-component, *parentEntryPoint* is the parent entry point. If not, this value is nil.
The *run* feature is a Boolean. The component executes itself only if this variable is true.
The insides of *inPorts* and *outPorts* are a little more complex. Here is the structure :

```
inPorts:'in'(
           name:port(q:AQueue)
           name2:arrayPort(qs:queues(a:AQueue b:AnOtherQueue)
                           connect:connect(a:2 b:1)
                           size:10
                          )
           )
outPorts:out(name:nil
             name2:arrayPort(1:nil
                             2:[comp#port comp2#port2])
             ...)
```

Input ports are saved in a nested record. Features are the names, and values are other records containing information. For a simple input port, the only data saved is the *bounded buffer*. An array port contains two more features : *size* and *connect*. The size is the maximum number of IPs. As the queues are dynamically created, it must be saved. The *connect* feature is a record saving how many components are connected to the selected port. This allows dynamic port creation and deletion.

Regarding output ports, the features are also the name. A simple array port is a list. An array output port is a set of simple output port. Each element of the list is a tuple which represents a link. The first argument is the destination entry point, the second argument is the input port name. So it's possible to do `{Port.1 send(Port.2 msg Ack)}`.

As explained in "Declaration of a component", the user inputs information when he creates components. This data is used to build the initial state. It's mainly a record transformations, internal state creation and a bounded buffer creation. Messages understood by component are describe in section "Use of a component". Many of them do very simple modifications to the state record. More complex ones create new array port or put messages in buffer.
The failure model is implemented in two steps. First the *ERROR* output port is put in the initial state. Every components has this port. The second step is to catch exceptions thrown during messages handling. For example, the main procedure is in a *try/catch* block. When an exception is caught, the error message is sent on the *ERROR* output port.

7.2 Sub-component

The sub-component implementation had the same structure, but a different state. The record looks like:

```
subcomponent(
   name:Name
   type:Type
   graph:graph(...)
   inPorts:inPorts('in':[Comp#'in')
   outPorts:outPorts('ERROR':<List>
                  out:[Comp#outputPortName Comp2#outputPortName])
   parentEntryPoint:nil
   )
```

The main feature here is the graph, which will be explained in a bit. The graph is a record representing a *.fbp* file. The sub-component doesn't have an execution, but transfers the messages from its own ports to internal ports. The entry points of internal components are inside the graph record.
The inPorts and outPorts features are sets of list. The feature is an external port name, the value is a list of tuples. The first argument is the internal component entry point, the second is the port name. When a message is sent to an external port, it's relayed to all the internal ports. The same applies for the output ports. An important point here is that sub-components don't exist in the final execution graph. They are only used to build graphs in a more simplistic manner. So when a bind action is sent to an external output port, the same bind action is sent to the corresponding internal output ports. The internal components are directly linked with the outside components.
The sub-component failure model is managed as follows. When a sub-component is created, the *ERROR* external output port is added. It's automaticaly linked with every *ERROR* internal output ports. If the external port is bound, every internal port will be bound too. This allows the global management of errors in the sub-component.

7.3 Graph

There is a special library which is used to tranform *.fbp* files in graph records. This record is used for launching a graph, or inside sub-component. Due to the autonomy of components, this library doesn't do a lot. When the graph file is read, it creates all the

corresponding components and make links. It's at this level, entry points are known, where all information is saved. Though it's possible to create a graph without this library, creating the components one by one and make links manually. It's important to understand there is no “scheduler”. This library doesn't manage execution, only the layout.
Here is the record corresponding to an *.fbp* graph file :

```
graph(
    inLinks:[ 'in'#aw#'in' ]
    outLinks:[out#td#out  out#awOut#out]
    nodes:nodes(
                '0GenOPT':node(comp:Comp)
               aw:node(comp:Comp)
               awOut(comp:Comp)
               td:node(comp:Comp)
               )
    )
```

There are three fields: inLinks, outLinks, nodes. The first two represent the external links. The last one is the set of all components inside the graph. There is no information about the links, they are inside components.
The content of *inLinks* and *outLinks* is a list of tuples. These tuples are the traduction of the fbp DSL for external links. The first argument is the external link name . The second is the internal component name and the third is the internal port name. From this example, we may conclude there are these three lines:

```
in => in aw()
awOut() out => out
td() out => out
```

When this record is created, the components inside are also created and the links are made. External ports are created only when this graph is used by a sub-component.

7.4 Bounded buffer

To make them simple, the bounded buffers are implemented with strong assumptions : the world is synchronous (no losses) and there is no bad nodes, which try to crash the system. It must be improved for a widely deployed system, but works well for the current version.
The base is the Queue object proposed in the CTM [8], figure 7.2. It uses two streams to build the queue.

To according with the needs of FBP, the buffer add an acknowledgment system. The IP is send with an undefined variable. When the *put* procedure is called and there is place in the buffer, the IP is put inside and the *Ack* variable is bounded. If there is no place in the buffer, the IP is saved in a FIFO list with the *Ack* variable. When an IP is remove from the buffer, an IP is get back from the list, if any, and put in the buffer, the acknowledgment variable is also bounded. The *get* procedure will block if there is no IP in the buffer.
This simple system fulfills all FBP needs regarding bounded buffers. When an IP is send, the component must wait for the acknowledgment before it may continue. This allows for an intentional operation block if needed.

```
fun {NewQueue}
   Given GivePort = {NewPort Given}
   Taken TakePort = {NewPort Taken}
   proc {Match Xs Ys}
      case Xs # Ys
      of (X|Xr) # (Y|Yr) then
         X = Y
         {Match Xr Yr}
      [] nil | nil then skip
      end
   end
in
   thread {Match Given Taken} end
   queue(put:proc{$ X} {Send GivePort X} end
         get:proc{$ X} {Send TakePort X} end)
end
```

Figure 7.2: CTM implementation of Queue

7.5 Semantic

The semantic of the graph is the semantic of the execution of each individual node. The semantic of the execution was already discussed in a previous section.
The semantic of the components is easy to show correct. The state before and after each message can be different or the same, and the treatment of a message begin only when the last is over.

```
State1 -> message 1 -> State2 -> message 2 -> ...
```

It's possible that a message starts the execution of a component (IP receive, options set, etc). The execution is started with the previous state. The same state is used for the whole execution. The start of the execution changes the state (the *threads* feature) but the end of an execution doesn't.

```
State1 -> message1 -> State2 -> message 2 -> State3 ->
start execution with State3 -> State4 ->  message 3 ->
State5 -> end execution with state3 -> State5
```

Except the internal state which is a dictionary, all the features are immutable variables, so there is no side effect. The dictionary is suppose to be change only inside the procedure, so there is no problem. If it's access/edit from the exterior, that can break the consistency of the procedure. If it's totally replaced by a new dictionary, so another address in the memory, the procedure will end normally. The next execution of the procedure will have the new dictionary, the old one will be dropped.

Chapter 8

Related work

Fractalide is one in many visual programming tools, that all have specificity. A huge list can be found on [3]. This section take some of them and situate Fractalide by comparison.

8.1 Erlang

A lot of features implemented in Fractalide are inspired from Erlang/OTP, like a kind of selective receive, a failure model and hot-swapping. These basic features offer a lot of tools to do distribute programs and are well tested and used.
The selective repeat is found in the basic of FBP. As Ulf Wiger explains [11] , multiple asynchronous mail-boxes is a kind of selective receive, as we can pick messages from some of them. This is a major reason why the input ports in FBP are asynchronous. It's sometimes more difficult to reason about the components, as they are no more deterministic, but that avoid number of state explosion in some case due to synchronous [10]. This way to do selective receive is less tested than the Erlang's one, but the experience of FBP seems to show it works.
As explained, Fractalide had also a failure model. It's possible to handle all errors at run-time, with a special output port. The hierarchy in FBP of sub-components add a more fine-grained management of the errors. It's possible to do, very easily, a global management for the all system, or very specific one for a part of the system. Also, the components are as independent as Erlang processes, so there is few risk for the all program.
The last feature inspired from Erlang is the hot-swapping capabilities. That push the dynamism of Fractalide a step further by allowing changing a component behavior without change the entry point which is shared. This feature is very well implemented as the procedure can be change, but also all the structure of the component. That allows to do introspection and intercession at run-time.
The major concept added to the Erlang messages passing system is the deep notion of hierarchy. As showed during this paper, it allows fine-grained control of the system at many levels.

8.2 FBP implementation

There exist today many FBP implementations. Paul Morrison works on JavaFBP, there exists CppFBp, GoFBP, etc. The Linux pipes are already a kind of FBP. This section will only show one of two alternatives.

8.2.1 NoFlo

It is surely the most well known FBP implementation, that were funded by Kickstarter. The interface is called FlowHub. The FlowHub interface authorize also to edit a running graph.
The main difference is that NoFlo implements “FBP in the web”, as Fractalide implements “HyperCard on FBP”. Thus, a lot of different choice were made.
To fit more with the actual web, NoFlo has for language javascript/coffeescript with html/css for the user interface. The choice of javascript make some concept of FBP more complex, like the blocking bounded buffer. Waiting an IP if the buffer is empty is hard to achieve. For a long time, NoFlo use synchronized input ports to deal with this problem. They have now asynchronous components.
Another difference is the graph execution is controlled by a scheduler. There is an underlying layer that control all the graph. In Fractalide, each component is autonomous, there is no need of a graph manager for the execution. For sure, if you want to edit the layout of the graph, a graph manager will be necessary to have all the entry points and the existing links. But this graph manager will not manage the execution. The NoFlo scheduler is a dynamic one. It’s so possible to make new links and add/remove component at run-time. They achieve dynamism in this way. The component doesn’t have hot-swapping feature, either behavioral nor structural.

8.2.2 Yahoo! Pipes

Yahoo! Pipes is an very specific goal of the FBP paradigm. It allows to easily work on basic web data : RSS feed and csv format. It provides a set of components to get these data from web-url, do some operations on it (filter, union, sort, etc) and finally give an output. The output are in the same format than the input. It’s easy to deploy the final result on the internet.
The main drawback is that there is no easy way to create new components. It’s possible to do sub-component, but not component. It limits the utilization for simple programs, authorized by Yahoo!. The interface is very intuitive and serve very well the aim of Pipes.

8.3 HyperCard fork of today

It’s hard to compare HyperCard fork and Fractalide for many reasons. The first one is that Fractalide doesn’t provide yet the whole HyperCard environment. There is the back-end layer to run HyperCard like programs, but not yet the front-end.
An existing HyperCard like program is Livecode. It’s a well finished program, working fine and that allow to make programs for many systems (Linux, Windows, MacOS, etc). They stick much more to the HyperCard first implementation. They have a English-like scripting language, that allows the user to do custom behavior for each card. They have also keep the basic hierarchy of HyperCard : items -> cards -> stack -> main stack -> ... . They add a notion of *grouped items* which is equivalent to the background capability of HyperCard.

8.4 Visual tools

8.4.1 LabView

LabView is a complete environment to make scientific programs with a visual programming tool. It possible to simply create the front-end display, and then connect the components on the back-end. It uses the data-flow concept, but is not a FBP implementation. It focus on scientific data manipulation and displaying. The goal of this visual language is to be at the same level of classical language. There is while loop, sequential, very low-level function like increment, equals, etc. This can lead to the same problem than other languages : an exponential cost of development for bigger application.The final graph is build with much more fine-grained component. In FBP, and so in Fractalide, there is no possibility to do a "while loop". Making loop in the graph is not a problem, but this concept is very attach to imperative programming. FBP try to do more coarse-grained programming, moving out from the von Neumann model .
But LabView is not build in the goal to make very big application. It is a finite and well used product. The interface is complete, and they propose many basic components. Also, they provide a similar concept of "sub-component", that allows huge re-usability. The community proposes many new components.

8.5 OOP

P. Morrison speaks a lot in his book about the relation between OOP and FBP. The card concept in Fractalide is very close of an object. It handles a state which is encapsulate. It's not edited by calling getters and setters methods but by sending messages. The advantage is there is no concurrency problem. If two components send a message to the same card, the card will proceed the actions one after the other. The encapsulation of data is stronger, as every change of the state happens inside the card.
The *create* component that we found often in the card is a kind of constructor, and we can imagine that *delete* component will appear for more complex cards, and will be like a destructor. The API is managed by the input actions wrapper card. The public methods are the caught actions. The non handle action are like *super()* call in Java.
There is no notion of inheritance. That is more due to functional paradigm. It's possible to do composition to extend the capabilities of a card. The concept of interface and polymorphism can be done, but not in a verified manner as Java do. It's possible to implement two cards that understand and send the same actions, handling a different state, but there is no verification at compile time.

Chapter 9

Conclusion and future work

Fractalide picks up and dusts off two age old technologies, that of HyperCard and Flow-Based Programming and combines them in such a way as to bring about a visual programming environment that perhaps results in a new paradigm concoction, but nonetheless the concepts that form its foundation are all well tested, documented and used in production. Fractalide is just one of many such solutions aiming to ease the crippling burden of development and maintainance costs. This was a tough challenge made easier by standing on the shoulders of giants.

This solution is by far incomplete and production usage comes with strong warning labels. Though the implementation is ready for early adopters to start contributing to. We've chosen the AGPL3 license as the *GPL has the strongest community building features. Initial experiments demonstrated this level of abstraction to be good at creating high-level applications. The paradigms used works well with actuators and sensors (GUI interface in our case). Hierarchical message passing, present in HyperCard, allows for fine-grained control of its behaviors. Fractalide adds the possibility to build customized and flexible hierarchies that are no longer hard-coded as is the case with the legacy HyperCard. Given we have swapped out HyperTalk, Fractalide is perhaps less intuitive for non-programmers, but the FBP backend chosen is by far a more powerful fit for the needs of application development.

Moreover FBP allows non-wizards to be aware of the logic made aparent by the simple graph layout, which allows for flow visualization from component to component in an intuitive and graceful manner. FBP pushes Fractalide a step further with many of its built-in properties, such as reusability, component sharing, improved maintainability, concurrency and side-effect free components.

Fractalide is a good back-end for a HyperCard like program. It adds a degree of dynamism to the system core allowing it hotswap on the fly, in an easy and consistent manner. As the implementation is minimalist, thus less bugs and easier to maintain. Lastly we've found FBP to be a natural fit for hierarchical scenarios.

Our FBP implementation is a complete. Some notions of "classical" FBP were removed, like the explicit dropping of IPs. In fact, only the concepts needed for a correctly functioning implementation were kept, making the code simpler. Components are autonomous, there is no over arching scheduler. Which allows for great degree of autonomy and dynamism. Including Erlang failure handling system allows for the creation of robust systems in a distributed world.

It would appear our HyperCard/FBP combination creates finer-grained graphs than classic FBP. This is not due to our implementation but because we work with smaller basic components, that of GUI items. Working with the GUI widgets demands this level of abstraction, though the logic of FBP components may be at a much coarser-grained level if need be.

Fractalide's future depends on its budgeoning community. It'll be a lost cause to have reusable, shareable components when users don't have the capability to create new components. Fractalide must provide all the tools to support the card creation, thus in turn nurturing a living system. Components will be used over long periods of time by many people, so a version control manager similar to Git must be provided. Also, a Github like platform should exist allowing people to form sub-communities surrounding particular cards and components of interest.

9.1 Future work

This paper is a proof of concept. It requires much more work to build a complete and viable system. This section will go over different features that can lead the Fractalide project further.

- HyperCard like interface

  The first point will be to develop a HyperCard like interface. Many of the future ideas are on the Github repository. The main-screen, replacing the main stack, will be like the starting page of a web-browser. The downloaded cards, implementing an application, will be displayed in the center. By doing a double-click, it will launch the application. A button "edit" will be available to switch in the card editor and edit the interface and logic at run-time.

  The components sharing must be develop in parallel. The power of FBP is in the re-usability. A user must be able to find an existing application easily, and also existing components. That must allow to design graph easily, without implementing a component in Oz. Yahoo! Pipes shows us it possible to build intuitive and reusable components.

- Implementation

  There still many to do, but here are few important issues that will be resolved. It's a sum-up, better explained on the Github issues section of the repository.

  - Bounded buffers

    As shown, the bounded buffers are great for a local execution, but bad for a distribute one. They must work like TCP connection, to be robust in an asynchronous world.

  - QTk cards

    There still unimplemented QTk cards, the port of the QTk library must be finished. Moreover, there is a bad design choice. For the moment, the creation can block the component if the *create* action doesn't finish in a QTK window card.

  - Creation of components

    For the moment, each component calls the component library at creation. This should be reversed : the component library must call the desired components. Also, the component name and type must not be handle by the component functor, but by the graph library which loads the component. It already does it for the sub-component.

- Security

  The components entry points are shared with full access to the methods. It is not possible to control which owners can change the state or just send messages. As the entry point is a procedure sending messages on the port, it's not complex to create another procedure that filter these messages, and only allow some of them. Many components just need to send IPs.

- Machine-human interface

  The Fractalide paradigm, hierarchical message passing, can work with many systems which have actuators and sensors. Fractalide focus on UI, but other must work. Our bodies work like that: messages are send by sensors (senses) and go up in the human hierarchy. Many of them go first in the spine, and then in the brain. It permits to have reflex for some actions, without going in the brain. Robots will stick very well also. If a camera see something, it will send the information to a "global manager", a kind of brain. The same for a wheel. But if the robot have four wheels, it will surely exist a "wheels manager" which will coordinate them, and send only some messages to the top level.

- Data-base

  An un-implemented feature of HyperCard is the data-base. As we work on a graph, it can be interesting to look the concept of DB graph [4]. Don't build a classic data-base as a new layer, but use the existing components to make the queries and fetch the data. Such system will allow to transform any applications in a data-base, and get the HyperCard feature.

- Network back-end

  The first network back-end should be to run a single graph in a distributed way. The same authority will be able to run his application in a distribute way, without any modification of the graph. This will allow to run intensive CPU applications for the average user, with all the benefits concurrency gives.
  Then, "HyperCard on the web" will be envisaged. A single graph will be manage by different authorities. An IP will go from component to component managed by different people. Hosting a component will be like hosting a web page. People will be allow to use the power and the knowledge of the server. Recent research in Content Centric Network show it's possible to drop the point-to-point paradigm. It makes more easy to distribute data over the network, which is mainly the utilization of Internet. It's known as Named Defined Network (NDN). It allows to transform the network in a memory. The next step is Named Function Networking (NFN) [9]. It makes a computation on the network and returns the result, just giving a name and not an address. As components are just a collection of functions, lets elevate the abstraction a bit to component. It can propose a new style of networking : the Named Component Networking. It will allow people to deploy their application easily on the network, without any problems due to point-to-point constraints.

Appendix A

Appendix

A.1 Calculator card

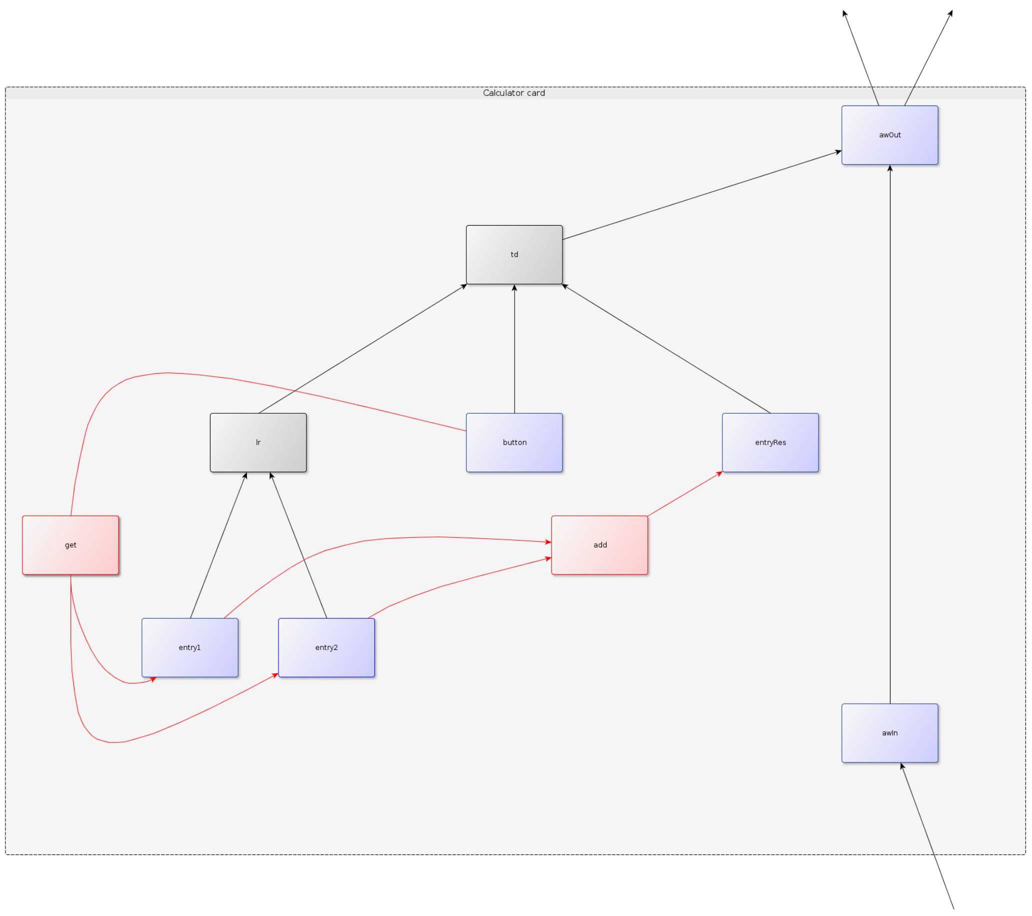


Figure A.1: port object The all calculator graph. In black the ui links, in red the logic links.

A.2 Install Fractalide

All the project is hosted on https://github.com/fractalide/fractalide.
The fractalide libraries are fully compatible with Mozart2, but one component is de-

veloped for Mozart1.4. So for the moment, Mozart1.4 must be used to run Fractalide. The following commands are necessary to compile the project :

```
git clone git://github.com/fractalide/fractallang.git
cd fractallang
make && make editor
```

For starting the editor, the command is the following

```
ozengine launcher.ozf components/editor/editor.fbp
```

Bibliography

[1] Facebook. Flux application architecture. `http://facebook.github.io/react/docs/flux-overview.html`, 2014 (accessed june 1, 2014).

[2] Danny Goodman. *The Complete HyperCard Handbook*. Bantam Books, 1988.

[3] Erick Hosick. Visual programming languages - snapshots. `http://blog.interfacevision.com/design/design-visual-progarmming-languages-snapshots/`, 2014 (accessed june 1, 2014).

[4] HyperGraphDB. Distributed dataflow. `http://hypergraphdb.org/learn?page=DataFlow&project=hypergraphdb`, 2014.

[5] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004.

[6] Leander Kahney. Hypercard: What could have been. `http://archive.wired.com/gadgets/mac/commentary/cultofmac/2002/08/54370`, 2002 (accessed june 1, 2014).

[7] John P Morrison. Flow-based programming. 2004.

[8] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, MA, USA, 2004.

[9] Christian Tschudin and Manolis Sifalakis. Named functions and cached computations. `http://www.ccnx.org/wp-content/uploads/2013/07/CCNDeployment_3_Tschudin.pdf`, 2013.

[10] Ulf Wiger. Structured network programming. `http://ulf.wiger.net/weblog/wp-content/uploads/2008/02/structured_netw_programming.pdf`, 2005.

[11] Ulf Wiger. What is erlang-style concurrency? `http://ulf.wiger.net/weblog/2008/02/06/what-is-erlang-style-concurrency/`, 2008 (accessed june 1, 2014).