Introducing Object Circuits

Matheus Costa Leite **Carlos J. Lucena**

e-mail: {matheus, lucena}@inf.puc-rio.br

PUC-RioInf.MCC27/02 Outubro, 2002

Abstract: In this paper, we introduce the concept of Object Circuits, a programming technique which addresses traditional object-oriented programming through the electric circuit's metaphor. We motivate the discussion by studying the usefulness of Object Circuits on the specific domain of modeling & simulation, and conclude by generalizing its applicability to other research areas.

Keywords: circuit, component, connection-programming, modeling, object-oriented programming, simulation.

Resumo: Neste artigo, introduzimos o conceito de Object Circuits, uma técnica de programação que aborda a programação orientada a objetos tradicional através da metáfora de circuitos elétricos. Nós motivamos nossa discussão estudando a utilidade de Object Circuits sob o domínio específico de modelagem e simulação, e concluímos generalizando sua aplicabilidade em outras áreas de pesquisa.

Palavras-chave: circuito, componente, programação orientada a conexões, modelagem, programação orientada a objetos, simulação.

1 INTRODUCTION

Object Circuits is a paradigm for software construction that combines the expressiveness of traditional object-oriented programming and the established semantics of traditional electric circuits. An incomplete list of proeminent features is given below:

- Intrinsically concurrent model.
- Support for dynamic evolution.
- Very loosecoupling between parts.
- High flexibility.
- Natural use of visual development environments.

The rest of this paper is organized as follows. Section 2 gives an overview of Object Circuits. Section 3 analyzes its use in the simulation domain. Section 4 throroughly explains several Object Circuit examples. Section 5 provides a brief discussion of related work. Section 6 presents a list of our ongoing and future work. Finally, section 7 summarizes the topics covered in this paper.

2 OBJECT CIRCUITS IN A NUTSHELL

The key idea behind Object Circuits is the establishment of an analogy between electric circuits and object-oriented programs. Thus, an Object Circuit is a circuit where objects flow through transmission paths. Figure 1 shows an Object Circuit and its composing elements. The first element is the *device*. A device is a software component that constitutes an independent unit of deployment, with asynchronous execution and a well-defined interface. In Object Circuits, this interface consists of a set of *pins*, as in a hardware IC. Device A, for example, has a set of pins ranging from P0 to P5. A pin works as sink (input) or source (output), and constitutes the primary channel through which a device receives and sends data.

We generalize the concept of a pin and define a *connection point*, or just point. Points are connectable entities that, along with *wires* – abstractions for connecting two points – are the blocks for constructing transmission paths.

A set of interconnected points and wires is called a *connection graph*. A connection graph is a complex channel through which data flows – in our case, in the form of object flux. It is said to be *active* if its flux is non-empty at a certain moment. The main property of a connection graph is that the flux from a connected source point is automatically propagated to all other reachable points, as seen in the left circuit from Figure 2. A.P4 (we will use the device's name followed by a dot to fully qualify pin names) is a source for flux f, which travels the connection graph to reach B.P1 and C.P1.

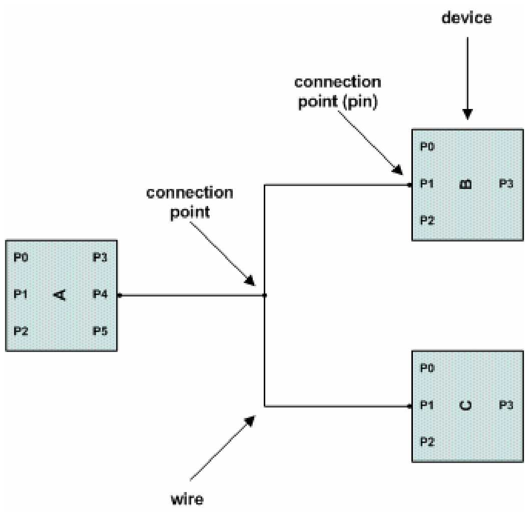

Figure 1 The main elements of an Object Circuit

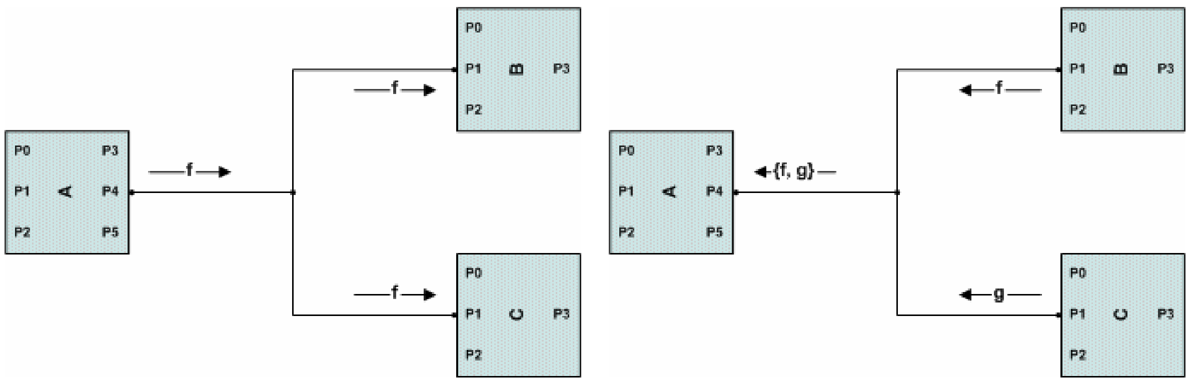

Figure 2 The traversal of object flux over a connection graph

It is possible that a graph has several flux sources simultaneously. In such situations, it behaves as a mathematical set, capable of grouping several fluxes together. This is shown in the right circuit from Figure 2. B.P1 and C.P1 output f and g, respectively. These fluxes are grouped into a single {f,g} set that reaches the input pin A.P4.

2.1 Composition

As a component, a device behaves as a black box which receives and sends data from and to the outside. From a user's point of view, as long as a device's interface is known, its internal mechanisms are irrelevant.

The black-box approach allows one to group devices together to build more complex ones. In Object Circuits, one builds such *composed* devices by creating connections, and wrapping devices around a new interface – i.e. a new set of pins. Figure 3, for example, shows a composed device built on top of devices A, B and C.

Composition is recursive, allowing multiple layer device hierarchies. At the leaf-level, there are the atomic devices, or Integrated Object Circuits (IOC). The rationale is that, as with electronic ICs, one may not "open" an IOC and expect to encounter reusable parts inside. Generally, an IOC is built on top of a lower level abstraction, in the same way machine code is used to write the basic constructs of a high level language. Alternatively, one may have an IOC that wraps an entire program or service, acting as a bridge between different technologies.

2.2 Concurrency Unfolded

Concurrent programs are hard to build. In this section, we shall argue that visual paradigms are better for capturing the essence of a concurrent program, and that the linear structure associated with traditional textual programming contributes to the difficulty of creating them.

A textual language induces a sequential way of thinking, and hence, of coding. A text line after another, an instruction after another, and so on. Even though there may exist several execution threads in a concurrent program, any single thread is ultimately tied to a sequential mechanism, meaning that the next instruction cannot be executed before the current one is finished. This model tends to hide the natural concurrency that exists among blocks of code, and contributes to slower algorithms.

Typically, a paradigm in which programs are *drawn* rather than *written* has an advantage in concurrency design. The reason is that the elements that compose the program are visually distributed in two or more dimensions rather than just one, resulting in a clear separation of logically interdependent blocks of code. For instance, take the Object Circuit approach. In a circuit, each computation node works independently from the rest. Sequential execution can still be achieved, as we will

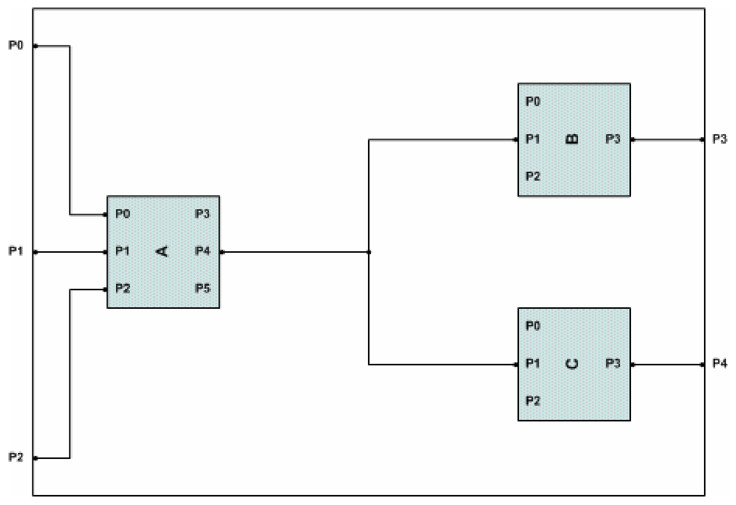

Figure 3 Internal view of a composed device

see, by means of special synchronization devices; nevertheless, concurrent execution is the norm. This approach has a natural elegance, as it induces the programmer to think concurrently and develop concurrent programs without adding unnecessary complexity.

Consider the factorial function pseudocode. Instructions (8) and (9) execute sequentially; however, a quick examination shows the result of (8) has no effect – nor is needed by – instruction (9). This means they could be executed concurrently, halving the time spent in a single loop step. But the program's linear structure hides this instruction independence. Although one may rewrite the program so that (8) and (9) execute in parallel, this is often very difficult due to the extra complexity involved.

Now, looking at the Object Circuit implementation of the factorial function in Figure 6, we see that, at the i-th step, the MULT IOC takes the current value of i to update the factorial variable f, while SUM updates the i counter – all at the same time. This corresponds to executing instructions (8) and (9) from the pseudocode in parallel. However, the ObjectCircuit approach achieved this result adding no extra complexity to the algorythm.

2.3 Dynamic Evolution

One of the main goals of component-oriented software development is to support dynamic evolution: the ability

```
 1   function factorial( n )
 2   {
 3          i ? 1
 4          f ? 1
 5
 6          while ( i = n )
 7          {
 8                 f ? f * i
 9                 i ? i + 1
10          }
11
12          output f
13   }
```

to alter a running system by seamlessly adding or removing components.

Dynamic evolution usually takes a considerable effort to be implemented in more traditional programming paradigms; nevertheless, the most basic electric circuits make use of it. If this was not the case, one would have to turn off the house's power switch before changing a single burnt light. This "magic" is only possible because an electric circuit is governed by a rigid set of laws – in this case, physic laws – that continuously guarantee its integrity, no matter the structural modifications one does.

The strength of this comparison makes us believe that the circuit foundations upon which OC rests is ideal for building systems capable of dynamic evolution. But finding a suitable set of laws is only half of the problem, as they deal with *what* to check for, but fail to say *how* to introduce modifications. As a matter of fact, one needs to envision a mechanism to carry on modifications, which, along with a set of integrity laws, forms a complete dynamic evolution solution.

A naïve approach to solve the how-to question is to let system users (e.g. a human, other systems, etc.) take care of modifying a running circuit as they find convenient. Unfortunately, this might not be enough for some domains. We need a more generic and robust solution, one that allows circuits themselves to carry on structural changes. Again, this idea is not exactly new. If we look to the side and think of biological circuits – for instance, the cells that make up our body – we shall realize that Nature has been adopting this approach for a couple million years.

Such self-referent mechanisms [HOFSTADTER 99] are to circuits what *reflection* [DEMERS XX] is to object-oriented languages: the ability of a system to talk about itself, which is often regarded as a powerful and important feature.

Currently, we have some preliminary solutions that incorporate dynamic evolution into the Object Circuits paradigm, but their discussion is not within the scope of this introductory paper.

3 SIMULATION AND OBJECT CIRCUITS

Simulation is the art of imitation. Not rarely, it turns out to be too costly, impractical or even impossible to study a real system in action. If one adds the requisites of having the system under total control and being able to reproduce the experiment at will, the task becomes even harder. For this reason, researchers from many areas find shelter under simulation techniques, which provide ways to capture within a *executable model* all the relevant features of a system under study.

From a historical perspective, object-orientation and simulation have a lot in common, as some key concepts from the former – including classes and objects, inheritance, and dynamic binding – were first introduced in a simulation language from the early 60's, Simula I [BIRTWISTLE 79]. In fact, the results obtained from the simulation domain caused Simula to be rewritten later as Simula 67: a full scale, general purpose programming language, which influentiated many modern object-oriented languages.

On the other hand, circuit theory is also intimately linked with simulation, in the sense that both deal with highly concurrent, communicating parts in a dynamic environment. Hence, the union of object-orientation and circuits is perfect: the former, as a powerful and expressive tool for modeling the world; and the later, as a platform where concurrent systems have a natural and elegant description.

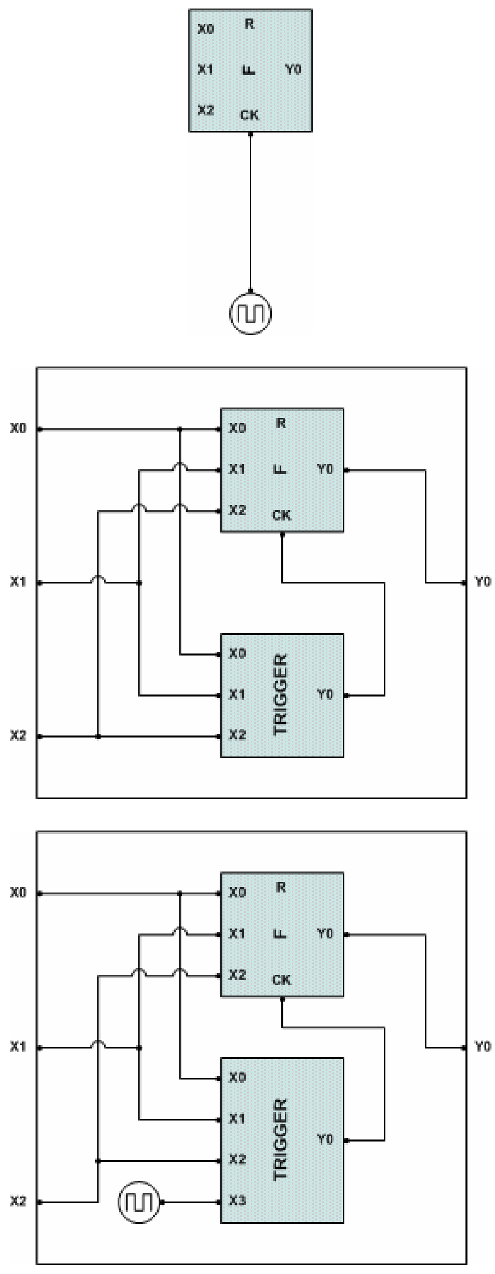

Figure 4 Time Stepped, Discrete Event and Hybrid modes. TS uses a clock device to mark time passage; DE uses the trigger's signal; and the Hybrid model is a combination of both

In the rest of this section, we discuss some aspects of simulation and the role that Object Circuits play in them. Also, the examples section includes two simulation problems modeled using Object Circuit: the game of Life and the Heatbug World.

3.1 Timing Modes

A *time stepped* (TS) simulation is one where the system state is updated at a certain constant rate. One can think of a global clock mechanism whose job is to periodically signal the passage of a time unit. Alternatively, *discrete event* (DE) simulations keep the state unaltered until some event causes a transition, leading the system to a new state.

Both TS and DE have their strengths and weaknesses. TS is usually simpler and cheaper to implement; however, if the system's state is likely to remain unaltered for long periods, refreshing the whole system too often may become wasteful. In these cases, a DE simulation is more adequate. Nevertheless, efficient event handling mechanisms usually have higher implementation costs.

Several simulation tools are tied to a particular timing mode. The generality of OCs, however, allows one to build models that seamlessly integrate TS and DE. This is possible because OCs make no global distinction between timing modes; rather, these are emulated by the very devices that compose the circuit. To understand how this is achieved, we shall explain how DE is implemented, for in our approach, a TS simulation is a subcase of DE where time advancement is itself a discrete event.

An *event* is defined as a flux change on a connection graph. We further subdivide events into three categories: *flux up*, *flux down* and *flux change*. Respectively, they correspond to the cases where an inactive graph becomes active, an active graph becomes inactive, and an active graph changes its flux (but remaining active). A device connected to a graph "listens" to its events, i.e. it is able to sense flux changes and react accordingly.

The leftmost circuit in Figure 4 shows two devices: F, which represents a generic function Y0 = F(X0, X1, X2), and a clock. As the clock device generates signals, they are sensed by the CK pin, causing F to recompute and refresh the Y0 output. This illustrates how the TS mode is achieved. Note that, although a global timeline may exist, it is not necessary, as each clock device defines an independent, local timeline.

The central circuit shows how to implement DE using a version of F that is driven by flux changes from the input pins. For this, we use F itself, and substitute the clock by a trigger device, whose function is to generate a signal on Y0 every time a change at X0, X1 or X2 occurs. This signal is sensed by the CK pin, causing F to reevaluate.

Finally, the rightmost circuit illustrates how to combine both timing modes. We have modified the previous circuit as to include a clock as well. The result is that F is refreshed either when a clock signal is issued, or when there is a flux change at one of its input lines.

3.2 Models of Computation

A model of computation is an abstract representation of a computer machine. Conversely, every algorithm is based on an underlying model of computation. With a model in hand, one can ignore details of implementation while studying the intrinsic execution time or memory space of an algorithm.

Some of the most popular models include the sequential, functional, relational, concurrent and distributed paradigms. However, it is the nature of the problem that will dictate which one is more appropriate, as models can vary greatly in performance.

Simulation tools are not only concerned about implementing a given model of computation: complex problems can be often subdivided in smaller problems which are more easily solved by different models. Thus, a primordial question is how to simulate heterogenous systems which operate under various models.

A possible approach, as adopted in [BHATTACHARYYA 02], is to explicitly implement several models and provide mechanisms to integrate them in a multi-model simulation. Object Circuits takes a quite different approach; it tries to implement a single model that is generic enough to embrace a variety of simpler ones. Although not perfect – as some models cannot be easily derived from it – it has the advantadge of getting heterogeneous systems integration at no cost.

Our intent in the remaining part of this section is to provide a brief discussion on how to implement a particular model of computation using Object Circuits. For this, we have chosen to describe the dataflow model.

3.2.1 DATAFLOWS

A dataflow is an asynchronous, distributed model of computation. In a dataflow program, a computation starts at a node as soon as all needed data becomes available at its inputs.

We implement dataflows with the aid of synchronization devices, named SYNC, as shown in Figure 5. Such devices hold the incoming flux until all input pins are active. When this happens, it simply lets all incoming flux pass through the output pins.

As we see on the left circuit, SYNC is itself implemented as an Object Circuit, composed of a flux detector, a switch, and an AND logic gate. The flux detector outputs boolean values stating whether the respective input pin is active. The logic gate uses these signals to discover whether all input pins are acive. Moreover, The AND output controls the switch selection pins, S_i. The switch device works as follows: whenever S_i has a true value, Y_i outputs the same as X_i. If S_i is false, then Y_i remains inactive.

The SYNC device behaves like a flux “barrier” – in this example, an AND-barrier. Nevertheless, since it was built on top of other devices, it could be easily modified to behave as an OR or XOR-barrier, or even a barrier of more complex logic.

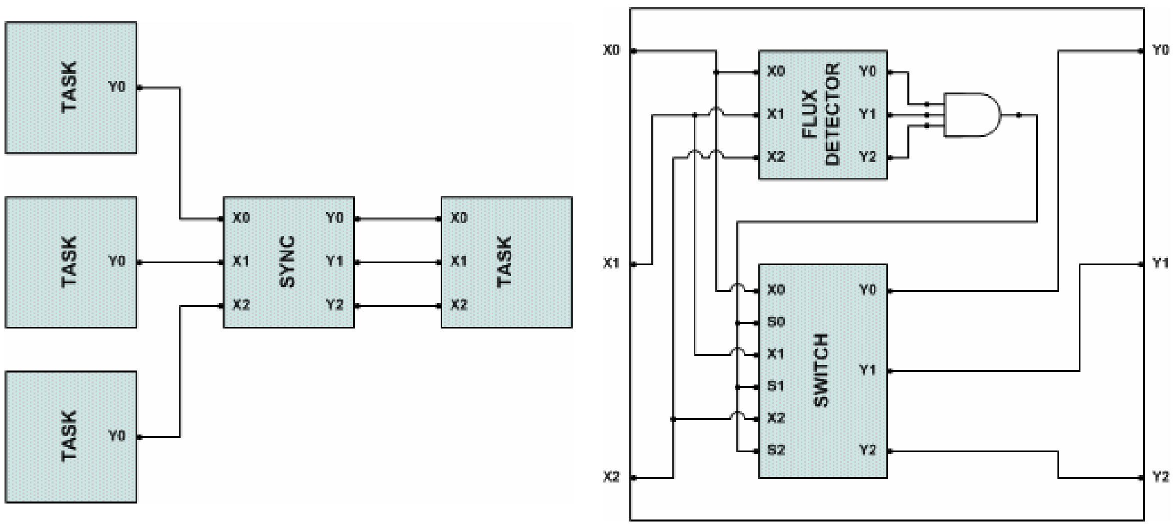

Figure 5 Modeling a dataflow. SYNC lets flux pass from the input to the output lines, as long as all input lines are active. To the right, SYNC's underlying Object Circuit implementation using a flux detector, a switch and an AND logic gate.

4 SAMPLE CIRCUITS

In this section, we provide several examples which provide a more detailed discussion about the mechanics of Object Circuits. We start with two very simple circuits, which compute elementar mathematical functions, and then continue with more concrete – albeit simple – applications.

4.1 The Factorial Function

In this first example, we shall build an OC capable of interactively computing the factorial function as clock signals (flux change events) are issued. Figure 6 shows the complete circuit. It contains two IOCs: an adder, labeled SUM; and a multiplier, labeled MULT. The small circle with a "+" denotes an object power source, which outputs a constant value – in this example, an object representing the integer 1.

Both IOCs have pins labeled X0, X1, CK and C. X0 and X1 are input pins; they provide the necessary arguments for the respective IOC functions. CK denotes a clock pin. In our example, a clock signal makes the respective IOC to read its input pins and recalculate its function, outputing the new value. Finally, the pin labeled C denotes a clear pin. A clear signal causes the IOC to restore its initial state.

The two pins labeled Y are the adder and the multiplier outputs. At any given moment, their fluxes corresponds to the value computed the last time a clock signal was issued.

The next circuit in Figure 6 shows the initial state. We suppose an external circuit is generating clock and clear signals, which are represented by flux changes between Boolean objects, denoted by F (false) and T (true). The initial values at the output pins were preset – they were previously "stuffed" into the IOCs as to fit this particular example. Typically, the first values outputed by a source are one's choice, and not the result of a computation.

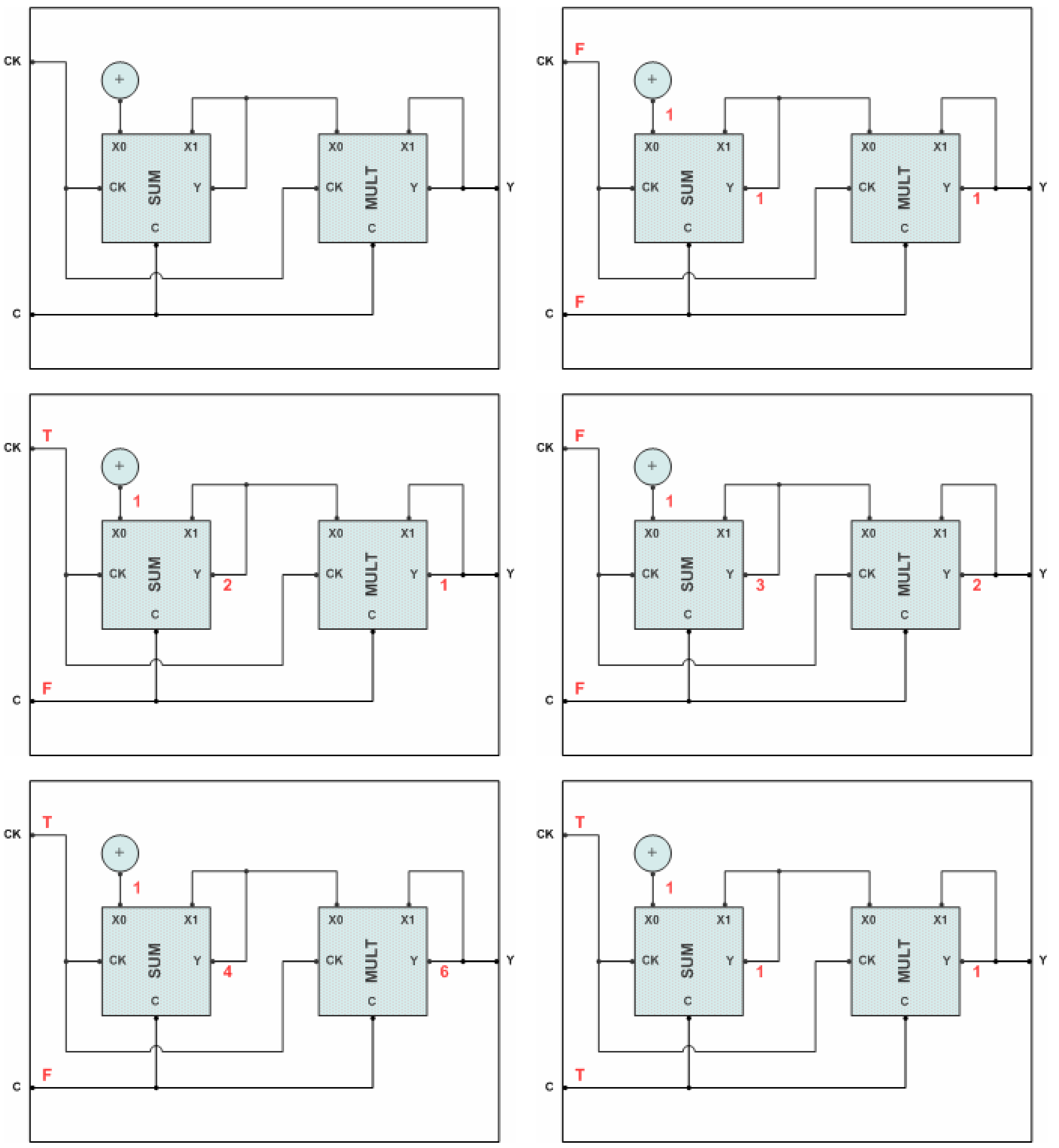

Figure 6 The Factorial Object Circuit and several stages of computation. The upper right circuit is at the initial state; the next four circuits show the computation of 1!,, 2! and 3!, respectively. The last circuit shows a clear signal and the transition back to the initial state.

This feature allows us to model the transient state of a system.

Next, we see the arrival of a clock signal. It propagates to the CK pins of both IOCs, whose underlying functions are immediately reevaluated. The adder evaluates X0 and X1 at 1, yielding 2. The multiplier, on its turn, also evaluates X0 and X1 at 1, yielding 1. Since the current and the newly calculated values match, there is no noticeable external change. But internally, this step corresponds to the computation of 1!, which can be read at the outer circuit's F pin.

It is easy to see that the i-th clock transition corresponds to the computation of i!. The next two circuits show the computation of 2! And 3!, respectively. Finally, the last circuit depicts a clear signal, which brings the whole circuit back to its initial state.

4.2 The Fibonacci Function

We shall now describe an implementation of the Fibonacci function, f(n) = f(n-1) + f(n-2), with f(0) = f(1) = 1. This example is in many ways similar to the previous, but illustrates the use of a new device: the delay. Figure 7 shows the completed circuit and several stages of computation.

In the Fibonacci function, the value computed for f(n) at a certain step will be used for the two subsequent steps, where it becomes, respectively, f(n-1) and f(n-2). We "transform" f(n) into f(n-1) by hooking the adder's output pin Y to its X1 input. Moreover, we transform f(n) into f(n-2) by connecting Y to a delay device, which in turn is connected to the X0 input. A careful examination will show that the flux at X1 is equal to the flux at Y one clock transition earlier, while the flux at X0 is equal to the flux at Y two clock transitions earlier.

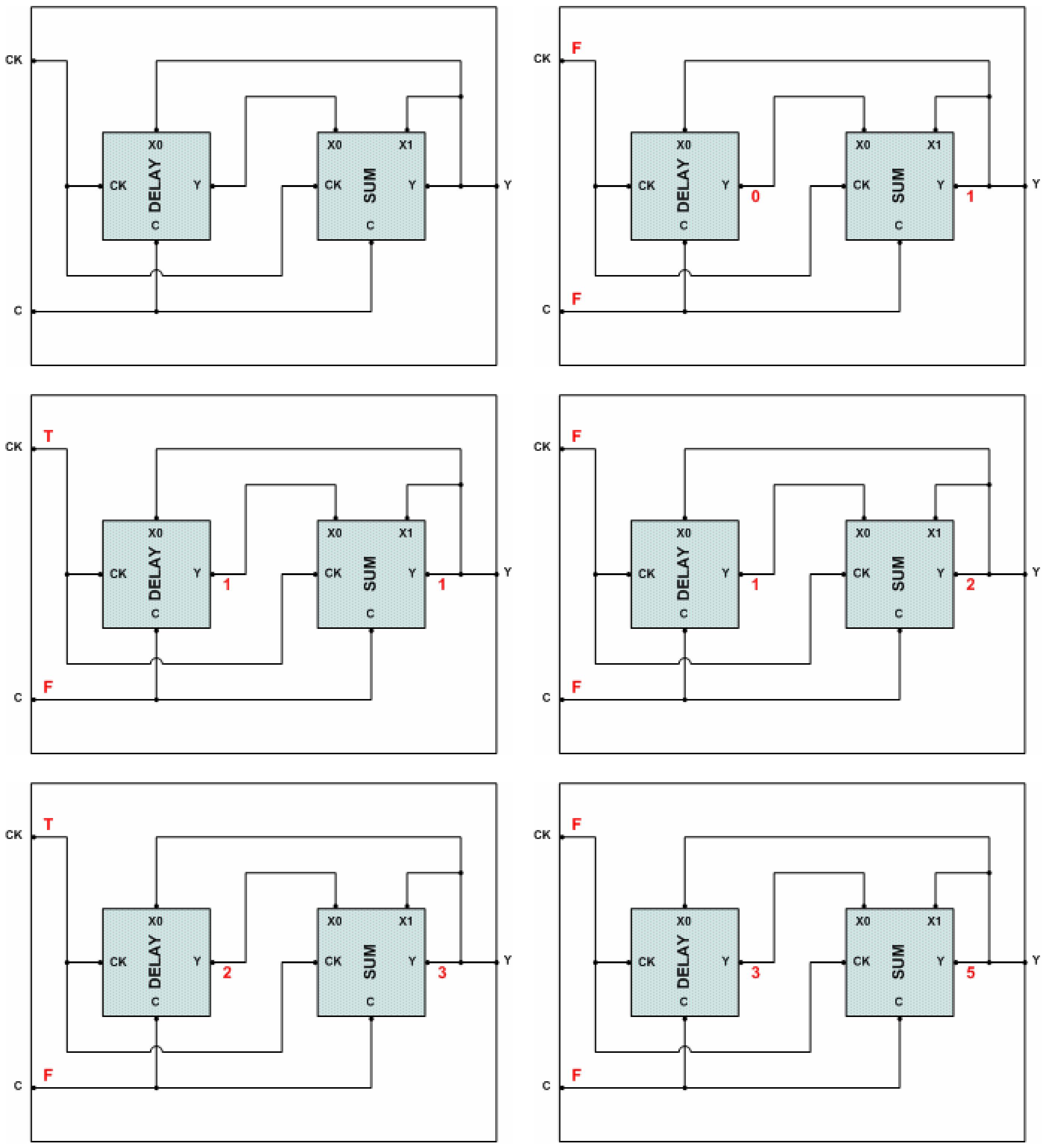

Figure 7 the Fibonacci Object Circuit and several stages of computation. The upper right circuit is at the initial state; the next four circuits show, respectively, the computation of f(1), f(2), f(3) and f(4).

Conway’s Game of Life

The Game of Life [BERLEKAMP 82] was created by the mathematician John Conway in 1970, and became a classical example of cellular automata. The game is run by placing a number of cells on a two-dimensional square grid. A cell can be alive or dead, and on each step, its state is recomputed following a set of rules that takes in account the cell’s current state and that of its surrounding neighbors.

A brief description of Life’s rules is given below:

- A cell with fewer than two or more than three living neighbors becomes or remains dead.
- A cell with two living neighbors maintains its current state.
- A cell with three living neighbors becomes or remains alive.

We implemented a cell as shown in Figure 8. There are eight input pins to which the neighbor cells connect to. The circuit works as follows: upon a clock signal, the SUM circuit evaluates the number of neighbor cells whose state is alive. This value, along with the cell’s current state, is passed to the F circuit, which then calculates the new state. The pseudocode for F is given below:

```
function $F_{n+1}$( $F_n$ , L )
{
        if ( L = 1 ) or ( L = 4 )
        {
                output 0
        }

        if ( L = 2 )
        {
                output $F_n$
        }

        output 1
}
```

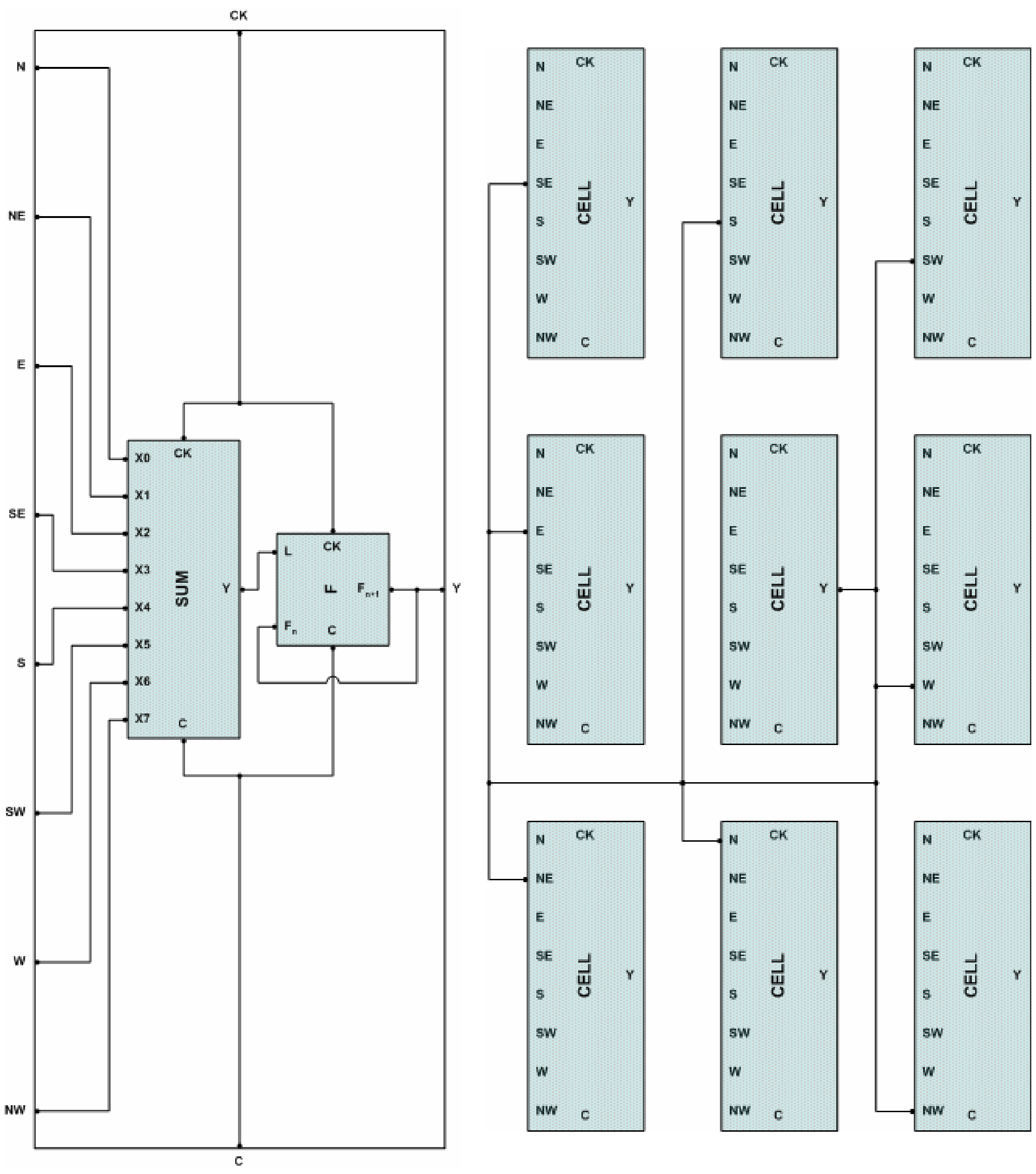

Figure 8 the internal view of an object circuit representing a Life cell. The SUM circuit adds up the number of living neighbors, while F calculates the cell's next state. To the left, a 3x3 cell grid. The wires shown connect the central cell's output pin, Y, to the corresponding input pin of each neighbor. For simplicity, all other connections have been omitted.

4.3 The Heatbugs World

In this example, we simulate the behavior of several interacting heat seeking bugs. This problem has been described by [MINAR 96], and we reproduce it here using the Object Circuits approach. In order to facilitate understanding, we have chosen to use the Java laguage to model the objects used in this example.

Consider a closed two-dimensional region with a certain property, heat, defined for each point. A number of heatbugs fly within this region, trying to stay at places where the temperature is amenate (each heatbug has an individual optimal temperature). However, the only heat sources are the heatbugs themselves; typically, they output a small amount of heat that gradually disperses over the environment. Thus, a place's final temperature depends of the distribution of heatbugs in the surroundings and of the heat each of them outputs.

SWARM [MINAR 96], a multi-agent simulation tool, uses the Heatbug World as an introductory example. Figure 9 shows a heatbug environment screenshot at a given moment. Although bugs are essentialy reactive agents driven only by their heat sensors, we can observe an emergent behavior arising from the system: heatbugs stay in bands if they need more heat, or alone when it is too hot.

We now detail the Object Circuit Heatbug implementation. Details have been ommited to better expose key concepts. Figure 10 shows the heatbug circuit. The central device, BUG BRAIN, controls the bug's wings and internal heat source. It senses the place's amount of heat from input pin H and, based on some internal cognition mechanism, orders its wings to execute a force (given by their vertical and horizontal projections, dF0 and dF1), and its heat source to output an amount of heat (given by dH).

The next two devices, labeled POS, accumulate force information to update the bug's position with respect to the force direction. Then, the ENCODER device takes the two-dimensional coordinates,

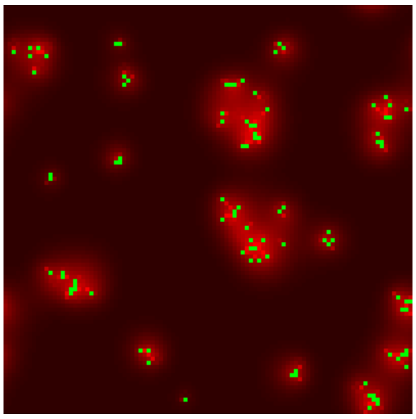

Figure 9 The Heatbug world. Bugs are represented by green dots, while the heat they produce are red regions

together with the heat amount dH, and pack them into a single object array. This array can be read from the outer circuit's Y pin. The input pin HFUN is a little trickier. It illustrates the use of objects as entities which also possess behavior and not merely data. The HFUN reads objects representing a heat function. With this function, the EVAL device can calculate the heat the bug is sensing at a given moment, using as function arguments its current coordinates, readily available from the POS devices. This heat amount goes to the output and is sensed by the BUG BRAIN, which we have already explained.

In order to preserve EVAL's reusability, it needs to handle a generic function, not tied to the heatbug problem. We have defined the interface `Function` that models such generic function. It contains a single method, `evaluate`, which accepts an array of `Object` arguments, and returns the value the function admits when they are applied.

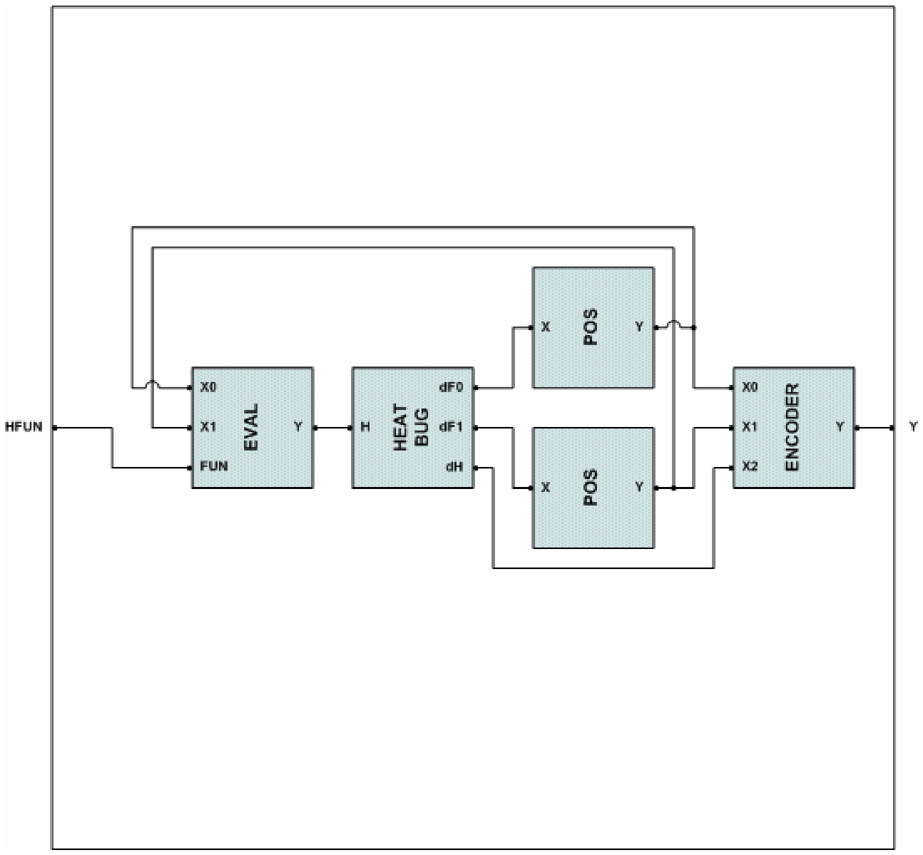

Figure 10 The Heatbug Object Circuit.

```
public interface Function
{
	Object evaluate( Object[] args );
}
```

```
class HeatFunction
	implements Function
{
	public Object evaluate( Object[] args )
	{
		Float x = ( Integer ) args[ 0 ];
		Float y = ( Integer ) args[ 1 ];

		return computeHeat( x, y );
	}

	private Float computeHeat( Float x, Float y )
	{
		// Calculates heat at position (x,y)
		...
	}
```

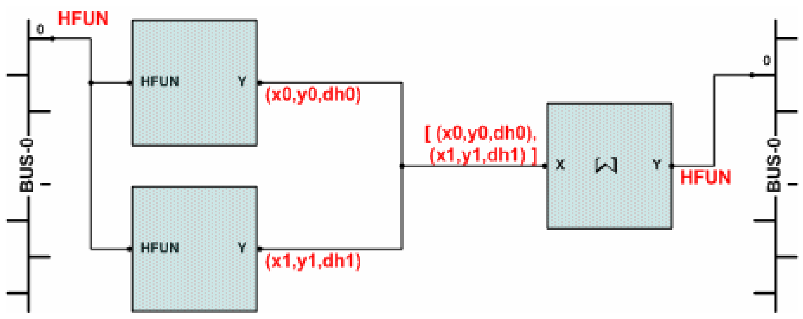

Figure 11 The Heatbug environment.

Next, we built the `HeatFunction` class, an implementation of `Function` whose job is to calculate the amount of heat present at a given coordinate pair.

In our example, HFUN senses a function object which is ultimately an instance of `HeatFunction`. But for all practical purposes, EVAL only knows it is handling a generic `Function` object, with a well defined interface.

We are now ready to show the rest of the system. Figure 11 shows an environment with two heatbugs (to the left). At a given moment, they write to Y an array containing their position and internal heat. The arrays from all existing bugs are sensed by the integrator device, labeled S. This device uses this information to build a `HeatFunction` object, which is written to the Y pin. Note that this pin is connected to the HFUN pins of both heatbugs through line 0 from BUS-0, thus closing the cycle.

5 RELATED WORK

5.1 Visual Circuit Board

Visual Circuit Board (VCB) is a component oriented software development tool which also exploits the metaphor of electric circuits. It introduces the concept of *datatrons*, a tree-based data structure named after their electronic counterpart, electrons. Datatrons are the impulse that travels transmission paths, carrying information exchanged by *parts*, the reusable software components that compose the circuit.

Besides the electric circuit analogy, which unavoidably involves abstract concepts such as components, connections and transmission paths, VCB has little in common with Object Circuits. Its approach induces a sequential, single-threaded execution mechanism. In a nutshell, VCB works as follows:

- A part creates and sends a datatron to one or more connected parts.
- The recipients process the datatron, possibly modifying it, or repeating this very algorythm in the meanwhile.
- The datatron is sent back to the original sender.

Despite some minor modifications, the steps above depict exactly what a procedural program does. This approach misses perhaps what is the most important circuit feature, so important that it is the very meaning of the word “circuit” – the fact that it is a closed loop, with no beginning nor end. The way VCB is designed, one needs a starting point, which is responsible for sending the initial datatron.

5.2 Ptolemy

The Ptolemy project [BHATTACHARYYA 02] is a simulation framework whose focus is on the modeling of heterogeneous systems. For this, Ptolemy defines a range of models of computation which rule the interaction between components. Thus, it is possible to build a hybrid model which implements several models of computation without adding extra complexity. While this is surely a powerful feature, it does not come without a couple of drawbacks.

For example, there is the learning curve problem. Ptolemy has many available models of computation, and a beginner may end up confused about which one to use under a certain circumstance. In this aspect, the single, generic model approach adopted by Object Circuits has a clear advantage.

Furthermore, a dynamic evolution as proposed by Object Circuits is not allowed in Ptolemy. The reason is that each model of computation would require individual treatment, causing any solution to be prohibitively difficult to implement.

6 ONGOING & FUTURE WORK

We are currently in the process of refining, improving and formalizing the Object Circuit theory. In parallel, we are building a preliminar implementation using the Java language.

Future work on Object Circuits includes the design of mechanisms to support circuit introspection and dynamic evolution, as discussed in Section 2.3. Also, we are interested in studying how other advanced techniques from software engineering (e.g. design patterns and contracts) and circuit design (e.g. fault tolerance and redundancy) can be dealt within to the Object Circuit world. Also, the build of visual development tools could greatly enhance the usability of Object Circuits as a programming paradigm.

Although this paper concentrates on explaining Object Circuit basics and its use as a simulation tool, we believe that a refined, thorought Object Circuit theory might expand its applicability to other areas as well. For instance, we foresee that an efficient dynamic evolution mechanism will enable the modeling of more complex and interesting Multi-agent systems. Moreover, the construction of comprehensive device libraries, along with a visual development environment, suffices for turning Object Circuits into a full fledged concurrent progra mming language. As our work progresses, we expect not only to find new uses for Object Circuits, but also to discover the very places where it excels related approaches.

7 CONCLUSION

In this paper, we have introduced the Object Circuit concept, a programming technique based on the well established semantics of electronic circuits and object-oriented programming.

We have shown that its main characteristics are a highly concurrent model, support for dynamic evolution, very loose coupling between parts, high flexibility and natural use of visual development environments. Our discussion has been motivated by studying the usefulness of Object Circuit as a tool for simulation, through a number of clarifying examples. Finally, we have concluded by envisioning future uses for a refined Object Circuit theory, beyond the simulation domain.

8 REFERENCES

[BHATTACHARYYA 02] S. Bhattacharyya, E. Cheong, J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, B. Vogel, W. Williams, Y. Xiong, H. Zheng. **Heterogeneous Concurrent Modeling and Design in Java**. Memorandum UCB/ERL M02/23, EECS, University of California, Berkeley, August 2002.

[BERLEKAMP 82] Berlekamp, Conway, and Guy. **Winning Ways (for your Mathematical Plays)**. Academic Press, August 1982.

[DEMERS XX] F. Demers, J. Malenfant. Reflection in logic, functional and object-oriented programming: a Short Comparative Study.

[HOFSTADTER 99] D. Hofstadter. **Göedel, Escher, Bach: an Eternal Golden Braid**. Basic Books, 20th anniversary edition, January 1999.

[BIRTWISTLE 79] G.M. Birtwistle. **SIMULA Begin**. Van Nostrand Reinhold, June 1979.

[KOKSAL 99] P. Koksal, I. Cingil, A. Dogac. **A Component-based Workflow System with Dynamic Modifications**. In Proceedings of the Next Generation Information Technologies and Systems (NGITS'99), Israel, 1999.

[MINAR 96] N. Minar, R. Burkhart, C. Langton, M. Askenazi. The Swarm Simulation System, A Toolkit for Building Multi-Agent Simulations. June 1996.

[VCB] Visual Circuit Board Homepage.

http://www.jcon.org/projects/vcb/

Matheus Costa Leite is a researcher from the TecComm-LES group at the Computer Science Department, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil.

Carlos J. Lucena is a full professor at the Computer Science Department, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil.