Applying a Circuit-Based Component Model to a Distributed OO Application

Matheus Costa Leite

Carlos José Pereira de Lucena

Luiz Fernando Chagas Rodrigues

João Alfredo Pinto de Magalhães

email: {matheus,lucena,lfrodrigues,magalha}@inf.puc-rio.br

PUC-RioInf.MCC38/2004 October, 2004

Abstract: In this paper, we motivate and propose a novel component-oriented model, called Object Circuits (OCs), by applying it to a typical web application: Webclipper. The former aims to take component-orientation to another level, by incorporating ideas from a field that is component-oriented by excellence: electronic circuits. The latter is an application for clipping news, i.e. the process of gathering, formatting and distributing content matching the interest of a group of users. Webclipper has been originally modeled and developed using traditional OO design. It is a real world, distributed OO application with enough complexity and generality to be useful for evaluating our approach. Furthermore, we have chosen an application that is not intuitively associable to OCs to illustrate its true generality. We have ported the application to our paradigm and the results are presented in the paper.

Keywords: component model, circuit, object circuit, software component, Software IC.

Resumo: Neste trabalho, nós motivamos e propomos um novo modelo orientado a componentes, chamado Circuitos de Objetos (COs), aplicando-o para uma aplicação web típica: Webclipper. O modelo visa levar orientação a componentes a outro patamar, incorporando idéias de um campo que é orientado a componentes por excelência: circuitos eletrônicos. Quanto a aplicação, trata-se de um serviço de clipping de notícias, i.e. o processo de extração, formatação e distribuição de conteúdo encontrado para determinados grupos de usuários.O Webclipper foi originalmente modelado e desenvolvido usando design OO tradicional. É uma aplicação OO distribuída, que possui complexidade e generalidade suficiente para testar o modelo. Além disso, a aplicação escolhida não é intuitivamente associável a circuitos de objetos, o que ilustra a generalidade do modelo. Nós transportamos a aplicação para o nosso paradigma e os resultados estão apresentados neste artigo.

Palavras-chave: modelo de componentes, circuito, circuito objeto, componente de software, CI de Software

Introduction

A topic of recurrent interest in Software Engineering is that of software components (Szyperski, 1998). The general goals of component-orientation are well known and happen to be the same of Software Engineering: building software that is more modularized, reusable and flexible.

Despite the efforts to leverage the component paradigm in the software field, the fact is that it has not yet flourished. This becomes clear if we compare the actual stage of software development to other areas where components are based on industrial standards and there is a market where clients can safely choose between implementations from different component vendors (McIlroy, 1968; Cox, 1990).

A field that has successfully embraced the component approach is electronics. Today, an engineer can design an electronic circuit employing only off-the-shelf ICs, and the market for electronic components is strong. This explains why hardware ICs are often used as a metaphor for explaining software components. The strong link between the two was made evident by Cox (Cox, 1986), who coined the term *Software IC* to designate the software counterpart of a hardware IC.

Unfortunately, the hardware metaphor is often used inadequately or superficially. For example, take the case of Object-orientation. A common mistake is to compare a class to an IC (Page-Jones, 2000) since both have an interface that encapsulates internal logic and that clients use to access a set of functionalities. However, a closer inspection reveals the weakness of this comparison. A hardware IC publishes its inputs and outputs through its pins, having no external dependencies. On the other hand, a class hides its outputs – i.e., calls it makes to functions in other classes. This creates external dependencies and contributes to increased system coupling (Ólafsson/Bryan, 1997).

A drastic consequence of hiding external dependencies is the *hyperspaghetti phenomenon* (Webster, 1995), where nearly all classes know each other, and a minor change to a class will affect the system as a whole. Only a disciplined designer with solid background on OO techniques, such as *Design Patterns* (Gamma, 1995), can hope to keep dependencies within a manageable level. This contrasts with hardware composition, where even the poorest designed circuit is still an assembly of interchangeable parts and the replacement of a single IC has little or no impact on the rest of the system.

In this paper, we discuss a component model that is closely related to the hardware metaphor. We have empowered software ICs with structured data packets called *objects*. Hence, the model's name: *Object Circuits* (Leite, 2003). Our approach tries to incorporate features that are common to electronic circuits, such as parallelism, autonomy and dynamic configurability.

Finally, we chose *Webclipper* (Magalhães, 2003) – a real world, distributed OO application – to evaluate our model. Webclipper is not an application intuitively associable to OCs, which illustrates the true generality of our approach.

This paper is structured as follows. Section 1 introduces some fundamental concepts. An overview of the Webclipper application is given in Section 2, followed by a detailed discussion of the OC model in Section 3. Section 4 is an introduction to OCRE, a runtime environment for OC instances, with special emphasis on synchronization issues. Section 5 discusses how Webclipper circuitry operates at system level. In Section 6, we enumerate the benefits of our approach illustrated by the application. Section 7 discusses some related work. Section 8 contains our final considerations, and Section 9 is a list of research directions.

1 Object Circuits in a Nutshell

An *object circuit* is an assembly of connected components and where *objects* flow through propagation paths. Objects are immutable tree-like pieces of information where each tree node is also an object. They are values, thus have no identity tag: two objects representing the logarithmic function or the word "CIRCUIT" are undistinguishable. There is a *null* object denoting empty data, represented by $\perp$.

Objects are instances of their respective *types*. Types allow a lot of important consistency checking, but we will skip the discussion of types in this paper because it is not central to our explanation.

In the realm of OCs, components are called *devices*. Devices are closed boxes capable of some computation, which can be connected to receive and send objects. They are hierarchically organized into composition trees. For instance, in Figure 1, D is the parent of E and F. We use a dot notation to fully qualify a device's name within its hierarchy, so $D.E$ means "device E, whose parent is F". We may also use a preceding plus sign to emphasize the top of the hierarchy, as in *+D.E*.

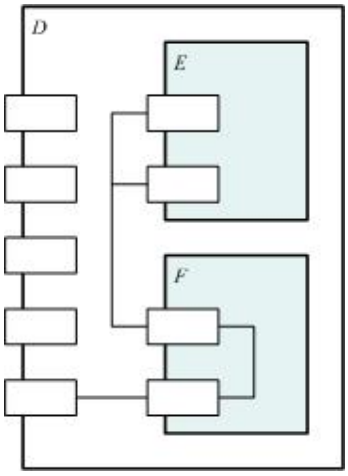

Figure 1. A device *D* composed of *E* and *F*.

Devices are connected through their interfaces, consisting of a set of *pins*. In Figure 1, pins are the rectangles lying at the border of each device. We use brackets to qualify pin names, so $D[P]$ means "pin P, from device D".

A *connection* is a link between two pins, and each pin may be connected to zero or more pins. We denote the connection between pins P and Q as (P,Q). They are bidirectional, so $(P,Q)=(Q,P)$. However, when representing connections, we may graphically wire several pins together, as in Figure 1. This is a simplification to avoid cluttering the diagram. The idea is that a connection exists between two pins if there is a path from one to the other.

In order to ensure encapsulation, a connection may never cross a device's border. For instance, a device can be connected to its parent, but not to its grandparent, for this would cross the parent's border. We use this fact to classify connections based on the hierarchical relationship held by the involved devices. A *child-parent* connection links a device to its parent. A *child-child* connection links distinct devices sharing the same parent. Finally, a *self* connection links a device to itself and comes in two flavors, *inner* or *outer*, depending on which side of the device's border it is located.

A primary function of a device is communication, achieved by sending and receiving objects through their pins. Each pin has a *state* given by an *input* and an *output* object, denoted by, respectively, $in(P)$ and $out(P)$. A pin whose output is non-null is sending, and one whose input is non-null is receiving data.

Object propagation is modeled by a set of formulae relating inputs and outputs. These formulae imitate the propagation of electric potential over a zero resistance path. In this sense, an object is the OC's counterpart of the electric potential. The list of axioms follows:

i. If $in(P) \neq \perp$, (P,Q) is child-parent and P is in the parent, $in(Q) = in(P)$.

ii. If $out(P) \neq \perp$, (P,Q) is child-parent and P is in the child, $out(Q) = out(P)$.

iii. If $out(P) \neq \perp$ and (P,Q) is child-child, $in(Q) = out(P)$.

iv. If $in(P) \neq \perp$ and (P,Q) is inner self, $out(Q) = in(P)$.

v. If $out(P) \neq \perp$ and (P,Q) is outer self, $in(Q) = out(P)$.

An example is given in Figure 2. Initially, suppose $in(P) = o \neq \perp$. We apply the axioms to demonstrate several other inputs and outputs:

- $in(P) \neq \perp$, (P,Q) is child-parent and P is in the parent. By (i), $in(Q) = o$.
- $in(Q) \neq \perp$ and (R,Q) is inner self. By (iv), $out(R) = o$.
- $out(R) \neq \perp$, (R,S) is child-parent and R is the child. By (ii), $out(S) = o$.
- $out(R) \neq \perp$ and both (R,U) and (R,T) are child-child. By (iii), $in(U) = in(T) = o$.

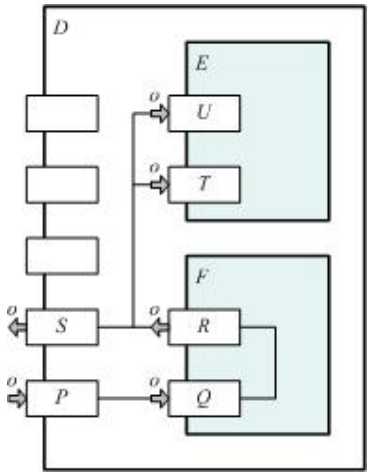

Figure 2 Relationship between inputs and outputs.

1.1 Circuit Dynamics

The dynamics of a circuit is described by a sequence of *states*, starting in an arbitrary initial state. The state of a circuit is given by the state of every device, which in turn comes from

the state of its pins, i.e. their inputs and outputs. Thus, if we monitor a running circuit, we will "see" a sequence of discrete changes to pins inputs and outputs. If a circuit reaches a stable state where no more transitions occur, its execution *terminates*.

The best way to understand OC dynamics is to imagine a device as a closed box capable of controlling the outputs, and sensing the inputs, of its own pins. However, if we "peek into" a composite device, we will realize that its outputs are actually coming from – and its inputs going to – its sub-devices (or even to itself, in the case of inner-self connections). This opens a recursive question about where exactly outputs come from and where they go to.

The answer is: it depends. Our approach only places a set of restrictions over the universe of possible input and output values at every pin at a given moment. Each implementation will provide a mechanism for actually manipulating pin state at the atomic device level, being the values at the composite levels a direct implication.

It is also the responsibility of the implementation to ensure no manipulation will take the circuit to an invalid state, i.e. one that breaks the set of formulae. In such state, the formulae could be used to demonstrate different input or output values for the same pin. For this reason, a manipulation that would lead the circuit to an invalid state is called a *short-circuit*.

1.2 Device Catalogues

The OC model is of little use without a set of general-purpose devices representing powerful abstractions that can be consistently reused to build more complex ones. When reuse comes to play, a concept that arises naturally is that of a *catalogue*. A catalogue is a hierarchical, domain-oriented device collection. The hierarchy of a catalogue is an acyclic graph.

The urge for component catalogues to leverage component-orientation has been pointed out by (McIlroy, 1968). A catalogue not only helps to quickly locate components by domain; it also allows choosing between various implementations of the same functionality, which differ from each other in aspects such as performance and reliability.

In particular, we are interested in *canonical catalogues*, i.e. those whose devices could be recursively composed to derive any other. We have created a canonical catalogue of highly reusable devices, inspired on a compositional, functional language described in (Backus, 1978). We call this the *Backus* catalogue, and it was used to build Webclipper's modules.

It is worth noting the Backus' devices do not exploit some possibilities, such as *duplex* pins, i.e. those used to both receive and send objects. The development of new devices that exploit specific features is an area of ongoing research.

2 The Webclipper Application

Webclipper is a software system for clipping news, i.e. the process of gathering, formatting and distributing content matching the interest of a group of users. It is an instance of *Avestruz* (Magalhães, 2003a; Magalhães, 2003b), a document classification framework, and has been originally modeled and developed using traditional OO design. Webclipper is a real world, distributed application with enough complexity and generality to be useful for evaluating our approach. In this section, we provide an overview of how it works, without dwelling into implementation.

Webclipper's goal is to scan a set of pre-registered sites, find all content matching the interest of its users and send individual reports describing what has been found. The process is repeated on a regular basis to ensure users are always fed with material.

The application starts by visiting the root of each pre-registered site and retrieving its contents. The content is a document, such as an HTML or a PDF file. Once a document is retrieved, the application decides whether it should be processed for interest matching. Also, its references to other locations are extracted and the application decides whether they should be queued for later visiting. Typically, the application stops searching when the visit queue is empty.

A given document should be discarded for interest matching when it has already been matched by a previous scan. Webclipper is only interested in fresh content, i.e. undetected by earlier runs. Hence, the first time a document is ever found is a signal to store its *hash code* into a database, so that subsequent occurrences can be detected and discarded for the purpose of interest matching.

Discarding references is a different problem altogether. In the first place, references previously found by this same scan must be discarded to avoid search cycles. In the second place, a cut criterion discards references too deep within the search tree. Finally, Webclipper must discard all references pointing to resources located outside the registered sites.

A document selected for interest matching is confronted against the interest of all users. Interest can be expressed by rules, such as "documents containing the words *object* and *circuit*". Although interest rules can get much more interesting and complex, it is beyond the scope of

this paper to provide a thorough explanation. Interest is used not only to decide relevance but also to actually extract interesting parts of a document. Thus, a list of "interesting" snippets is extracted from each document; each snippet being associated with a given document and user.

As document snippets are produced, they are transferred to a central database. From time to time, based on user notification preferences, Webclipper will collect snippets from the central database to build individual reports, which are sent to the users through some channel, such as email.

In order to maximize efficiency, several Webclipper instances are run in parallel, each one communicating only with the central database. To avoid different instances reporting the same material, they are configured to scan disjoint sets of pre-registered sites.

3 Webclipper Circuitry

In this section, we detail some of Webclipper's devices, showing how they were built based on the Backus catalogue. We will also give an overview of how these devices are joined to operate together at system level. However, we will omit details in favor of clarity and simplicity. In particular, how multiple flows are synchronized – where synchronization is needed – has been omitted.

3.1 The Matchmaker

The Matchmaker device (Figure 3) is responsible for matching the interest of a list of users against a certain document. It receives a vector of *user* objects and a *document* and outputs a vector containing the snippet corresponding to each user. A typical document is an HTML page or a PDF file. A user with no interest in the provided document produces $\perp$.

The User Object

A *user* is an object representing one of Webclipper's registered users. It encapsulates data that includes the following, among others:

- *interest*: a user-defined *interest* object. It is a function containing information about how to detect and extract content of interest from a given document.
- *report-conf*: an object containing data used to build the user's report. For example, the *template* used to format the report's content is included here.

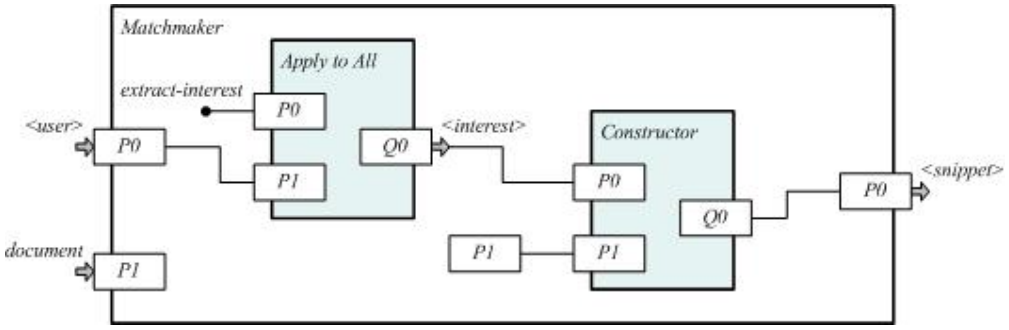

Figure 3 The Matchmaker device.

The Matchmaker is implemented using two Backus sub-devices: *Apply to All* and *Constructor*. The first accepts a function object and a vector and applies the function to each vector element. In our example, it is fed with a constant *extract-interest* function object and a user vector. *extract-interest* takes a user and returns its interest. Thus, *Apply to All* outputs a vector containing the interest function of each user.

The second sub-device works on the opposite way. It accepts a function vector and an object and applies each function to the object. In our example, it is fed with the vector of *interest* functions and the document to be matched. Each *interest* function takes a document object and extracts a snippet (possibly $\perp$) from it. Thus, *Constructor* outputs a snippet vector, which is also the output of Matchmaker.

The Matchmaker device will repeatedly be confronted against documents in order to produce the snippets for all registered users. Subsequently, the snippets created will be used by other devices to assemble the reports.

3.2 The Link Gate

The Link Gate device encapsulates the logic for deciding whether a reference found within a document should be selected for later visiting, or immediately discarded. It is part of a family of devices known as *gates*. A gate with width *n* has pins $P_0, P_1, ..., P_{n-1}$ and $Q_0, Q_1, ..., Q_{n-1}$. It works by either forwarding or blocking the input values of the P_k pins to the respective Q_k pins. If the gate is in forward mode we say it is *open*; otherwise, it is *closed*. Formally, $out(Q_i) = in(P_i)$ if it is open, or else $out(Q_i) = \perp$. The function that decides

whether to open or close the gate and the decision of when to apply such function are particular to a specific gate.

In Figure 4, an instance of a transition-triggered gate with $n = 2$ is shown in two situations. First, we see the gate closed for incoming values x_0 and $\perp$. Next, a transition from $out(P_1) = \perp$ to $out(P_1) = x_1$ triggers the gate, causing it to reevaluate and open.

Figure 4 An open and a closed gate device.

The Link Gate shown in Figure 5 is a specialized gate device built upon daisy-chained sub-gates. It expects a link object at P_0. Daisy-chaining has the effect of a logical conjunction: a link will pass through the device only if it manages to pass through all sub-gates. The first gate closes for links that have been presented to it since the application started. The second gate closes for links whose depth is too high. Finally, the third gate closes to links whose URLs are *external* to their root's URL. A URL is external to another if their top-level domains differ. For example, http://www.foo.com/news/index.html is external to http://www.bar.com. Avoiding external URLs is important to prevent Webclipper from crawling the entire web.

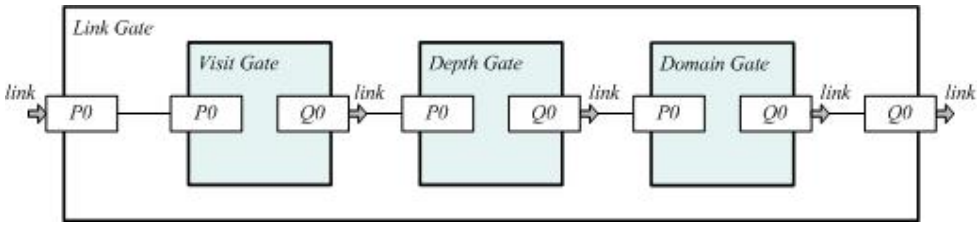

Figure 5 The Link Gate

The Link Object

A *link* is an object that aggregates information about a web resource. Webclipper uses it to decide whether to further process it or not. The application always starts by visiting a list of pre-registered *root* links. A link has the following properties, among others:

- *url*: the resource's address.

- *root*: the initial root link from which Webclipper eventually reached this link.
- *depth*: a value indicating the distance of this link to its root.

4 The Runtime Environment

As part of our research, the OCRE (Object Circuit Runtime Environment) is under continuous development to serve as a platform for developing and testing applications. The Webclipper application we described was implemented with OCRE.

OCRE provides two abstract device models: one for composite devices, and a special atomic model that bridges object circuits to other technologies. The rationale is that the atomic model can be customized to instantiate primitive devices from which composite ones can be derived.

Basically, to instantiate an atomic device the designer specifies how it reacts when a subgroup of its pins suffers an input change. Typically, the reaction includes doing some computation and changing the output of its pins.

4.1 Concurrency and Synchronization

Devices are independent in the sense that they only interact with the environment, and the receivers of an object are unknown to its sender. Allied to the fact that objects are immutable values, thus free from evil side-effects, it is easy to see that device computations may safely be performed in parallel. This raises a need for synchronization techniques.

A possible approach is to have devices listen to a global clock. In this case, concurrency is not maximized as the clock may only advance after the slowest device has finished computing. In a truly asynchronous scenario, a global clock is not available. We now introduce a new data structure we called the *t-buffer*, and discuss how to use it to synchronize object flow in such a scenario.

A *t-buffer*[1] is a group of independent *row* vectors where elements at the same index within each row define a *column*. A column is *complete* at a certain index *i* if all rows have size $n > i$. A t-buffer has several associated operations. First, $add(i, o)$ appends o to the tail of the i-th row. The $get(i)$ operation returns the i-th complete column without removing it. If removal is also needed, $remove(i)$ should be used, causing the index of subsequent elements to shift as well. Both *get* and *remove* are undefined for incomplete columns. Finally, *size()* yields the smallest among the sizes of all rows, which also happens to be the index of the first incomplete column. In Figure 6, a t-buffer with four rows and zero complete columns is represented by an "incomplete" matrix.

$$\begin{bmatrix} x_{00}\ x_{01}\ x_{02} \\ x_{10} \\ \\ x_{30}\ x_{31}\ x_{32}\ x_{33} \end{bmatrix}$$

Figure 6 A t-buffer

OCRE's leaf device has built-in synchronization capabilities enabled by a t-buffer. Each of the buffer's columns holds an atomic data unit; thus, a given column should be processed only if complete. The device contains *n* general purpose pins, P_i, and *n synchronization-in* pins, SI_i. The idea is that $in(P_i)$ is added to row i whenever $in(S_i)$ suffers a non-null transition (one where the old and new values are both non-null), and we say P_i has been *acknowledged*. In this case, the modus operandi of P_i is *discrete*, for it is acknowledged after a discrete transition of SI_i.

If $SI_i = \perp$, then P_i becomes a *continuous* pin. Under this mode, P_i is acknowledged whenever $in(P_i)$ changes, or yet, whenever any other (discrete or continuous) pin is acknowledged. Put another way, if a (discrete or continuous) pin is going to be acknowledged, all other continuous pins are acknowledged as well, independently of whether their inputs have changed.

[1]The name comes from the computer game of *Tetris*.

The difference of discrete and continuous acknowledging is the same observed between digital and analogical electronic devices, respectively. While the former waits for some control signal (e.g., a clock) to denote a given data input is now valid for reading, the latter considers the current input to be always valid, so it must be re-evaluated if any other data input changes.

As pins are acknowledged, the t-buffer rows are filled. Synchronization is achieved by removing the buffer's first column whenever it becomes complete, and processing it as a single, consistent data unit. Figure 7 shows a step-by-step example.

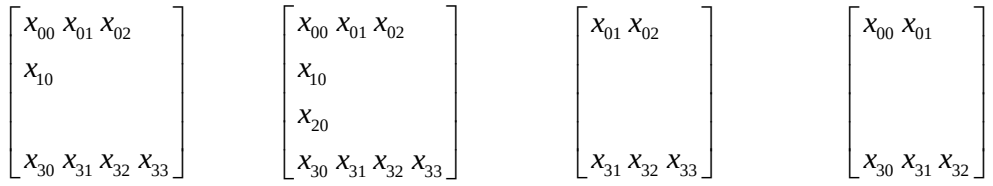

Figure 7 Synchronization with a t-buffer

There is also a *synchronization-out* pin per device, called SO, whose output is a Boolean object. The idea is to mark the end of every computation by toggling *out(SO)*'s value. Typically, the SO pin of a device D will be connected to several SI_k of devices dependent of D's result.

The t-buffer based synchronization mechanism allows maximum control over device operation because it is fine-grained to the level of a single pin. If needed, a device could even be configured to operate listening to a global clock signal by interconnecting all SI pins directly to the clock. Moreover, t-buffers seamlessly solve the problem arising when a device receives input faster than its ability to handle data. All incoming data is automatically buffered and complete columns are processed as the device becomes ready. This is similar to the *Producers and Consumers* problem found in the literature (Burns, 1993).

OCRE also has an alternative approach to synchronize flow that does not rely on explicit signals; rather, it is based on implicitly synchronized device *groups*. The rationale is that, if at a certain time t, two or more devices from a group receive some data, process it, and then make changes to their outputs, all changes will be committed atomically. This is equivalent to achieving synchronization in an electronic circuit by using devices with similar timing properties. In such a situation, the time elapsed between the devices' outputs fall within some acceptable tolerance – thus being, for practical effects, simultaneous events.

5 Joining the Pieces

We proceed by showing how to integrate the Matchmaker and the Link Gate devices at system level. Figure 8 shows a diagram representing two of Webclipper's subsystems, *Matchmaker* and *Scanner*. They are responsible, respectively, for matching interest and scanning the web. The clouds denote connections to other subsystems.

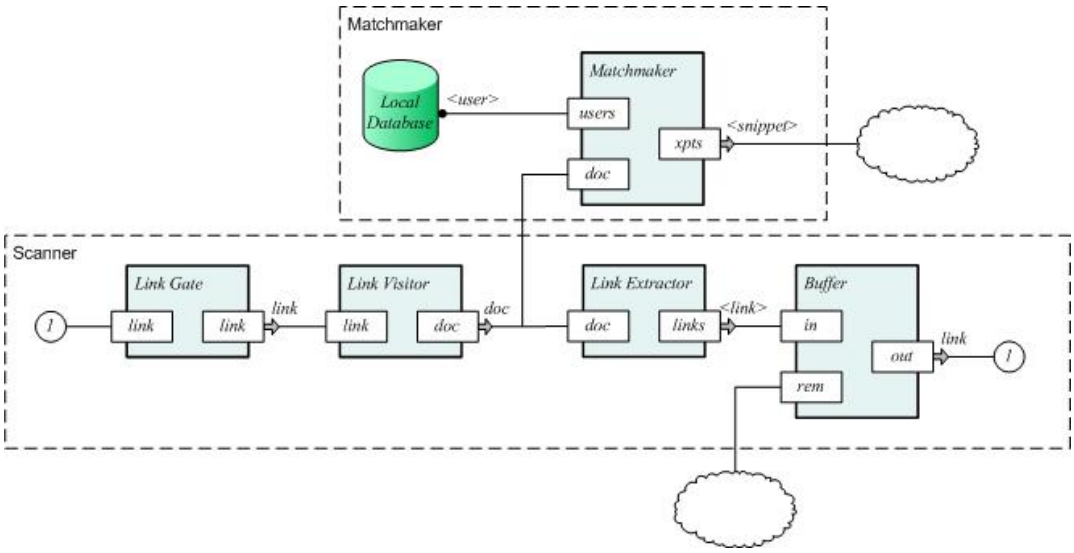

Figure 8 The Matchmaker and Scanner subsystems.

The Link Gate produces link objects that are consumed by the Link Visitor. The Link Visitor visits links and produces document objects representing the resource located at the link's URL. These document objects are sent to the Matchmaker and the Link Extractor. The Matchmaker also receives a vector of users from a local database. Typically, this vector remains unchanged for the extent of a full scan.

The Link Extractor is responsible for searching references inside documents. The found references are wrapped around link objects that later will be used to retro-feed the Link Gate. The produced list of links is presented to the Buffer device, which appends its elements to an internal FIFO buffer. The Buffer contains an *in* pin that, upon a non-null transition, removes the oldest buffered element and sends it to an *out* pin. In the diagram, *out* pin is connected to a small circle labeled "1". This is "syntactic sugar" denoting that all other circles with an identical label represent the same point. There is such a circle connected to the Link Gate device, meaning it will receive links from the Buffer.

The diagram shows another subsystem controlling the Buffer's *rem* pin. When *rem* suffers a non-null transition, the whole *Scanner* and *Matchmaker* subsystems perform a search

cycle: a link is sent to the Link Gate, which may forward it to the Link Visitor, which in turn visits the link and produces a document. The document is sent both to the Matchmaker and Link Extractor devices. The Matchmaker will confront the document against user interest, and the Link Extractor will extract references and send to the Buffer, thus completing the cycle.

6 Benefits of Our Approach

The use of OCs to design Webclipper has produced a very modularized, loosely coupled and parallelized application. We feel particularly comfortable with this statement because we have not designed a new application from scratch; rather, we have ported an existing OO application to the OC realm – and when both versions are put side by side, the differences are clear.

For example, concerns (Dijkstra, 1976) that are not central to Webclipper's goals were originally spread over several modules, creating code that was very hard to maintain. OO alone is not capable of correctly dealing with this issue; actually, an elegant solution can only be achieved by extending its framework (and introducing complexity), as in Aspect-orientation (Kiczales, 1997).

In contrast, the OC-based Webclipper has logging components that listen to objects passing through connections. No external solution was required and the logging concern is now centralized. If needed, logging could be selectively turned off at runtime by disabling logging devices.

The original Webclipper also strived to keep sub-systems as parallel as possible due to performance and reliability issues. The multi-thread logic behind it took considerable effort to design and code, and in the end, the system was only modestly parallel.

This contrasts with OCRE, where even the tiniest device is an independent logical processor. Applications are always parallel, and this is not a designer's choice. Device's computations are allocated to physical processors at runtime, so even when a device depends on the result of others to continue, OCRE manages system resources on the background to keep all processors busy. Load balancing comes for free.

A side effect of implicit parallelism is that the new Webclipper is considerably smaller than its ancestor since its complex multi-thread logic is now managed automatically. Coupled with the fact that our approach has a very tiny framework and intuitive concepts, the result is that the new Webclipper also is much easier to understand and maintain.

7 Related Work

Nearly all component-oriented approaches have links with the hardware metaphor. Sometimes they are implicit, other times the authors make them evident. In the latter case, the metaphor is often mistakenly used or is there mainly for didactical purposes. The OC model pioneers by consistently defining a software equivalent of an electronic circuit. This said, we now present a small survey of component approaches that explicitly claim to be based upon the hardware metaphor.

Fusebox (Quarto-vonTivadar, 2003) is a standard framework for building web-based applications with an associated methodology, called FliP. In Fusebox, each part of the application is an independent circuit. Should a circuit fail, the rest of the application keeps functioning. Mostly, the metaphor is used didactically and helps in building more reliable web applications.

The *Autonomous Component Architecture* (ACA) (Tyan/Hou, 2002) has hierarchically structured components that can be connected to each other through a set of *ports*, akin to OC's pins. However, communication is unidirectional as ports have both an *input wire* and an *output wire*. In ACA, a port implements a service that other components may use. Requesting a port's service and later receiving a response is a process that takes place in an independent context – a functional style that differs from the nature of an electronic circuit.

A style similar to ACA can be found in the Simple Interprocess Messaging for Linux (SIMPL) approach. SIMPL introduces components based on the *send-receive-reply* messaging mechanism of UNIX processes, which are compared to hardware ICs. The communication model is synchronous: a *sender* process sending a message to a *receiver* blocks until it gets a response back. On the other hand, the receiver remains halted until a message destined for it arrives. SIMPL authors' argue that by forcing synchronization at each message pass, the resulting software exhibits a more predictable and replicable behavior, which is an important feature of hardware ICs.

8 Conclusions

In this paper, we have introduced Object Circuits as a novel software modeling paradigm. It aims to take component-orientation to another level by incorporating ideas from a field that is component-oriented by excellence: electronic circuits.

Throughout this paper, we have motivated our approach by describing how we ported Webclipper – a real world, distributed OO application – to the OC realm. We have also shown that the result ended up being more modularized, loosely coupled and parallelized than its ancestor. Successfully porting Webclipper was crucial to strengthen the arguments in favor of our approach because it is not an application intuitively associable to the circuit metaphor.

9 Future Work

Our research is divided into two branches: refining and extending the OC model, and implementing development tools.

From the first branch, we emphasize the research concerning object structure and specification of object types, exception handling, device contracts, textual representation languages, dynamically modifiable circuits, design patterns, etc.

From the second, our efforts concentrate on the development of OCRE, to eventually turn it into a full-fledged software development environment with visual and textual editors, testing and debugging facilities, etc. We are also interested in developing new catalogues of general-purpose devices to serve as basis for other applications.

10 References

Burns, A.; Davies, G., 1993. Concurrent Programming, Addison-Wesley.

Kiczales, G. et al, 1997. Aspect-Oriented Programming. Proceeding of ECOOP'97, Lecture Notes on Computer Science, volume 1241, Springer, June 1997.

Quarto-vonTivadar, J., 2003. Discovering Fusebox 4 with ColdFusion. Techspedition Press.

Backus, J., 1978. Can Programming Be Liberated From the Von Neumann Style? A Functional Style and Its Algebra of Programs. In *Communications of the ACM*, 21(8), 613-641.

Cox, B., 1986. Object-oriented Programming, An Evolutionary Approach. Addison-Wesley, Second Edition, 1986.

Cox, B., 1990. Planning the Software Industrial Revolution. In: IEEE Software, Vol 7, p25-33.

Leite, M., 2003. A Model of Computation for Object Circuits – MSc Dissertation, CS Department, PUC-Rio.

McIlroy, M., 1968. Mass-Produced Software Components, Software Engineering Concepts and Techniques (1968 NATO Conference on Software Engineering), Van Nostrand Reinhold, 1976, pp. 88-98.

Magalhães, J., 2003. Um Framework Multi-Agentes para Busca e Flexibilização de Algoritmos de Classificação de Documentos – MSc Dissertation, CS Department, PUC-Rio .

Magalhães, J.; Lucena, C., 2003. Using an Agent-Based Framework and Separation of Concerns for the Generation of Document Classification Tools. In *AAAI Spring Symposium/AMKM-Agent Mediated Knowledge Management* (to appear LNCS - Springer).

Dijkstra, E., 1976. A Discipline of Programming. Prentice Hall, Englewood Cliffs, NJ.

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J., 1995. Design Patterns. Addison-Wesley Professional; 1st edition (January 15, 1995).

Ólafsson, Á.; Bryan, D., 1997. On the Need for Required Interfaces of Components. In: Special Issues in Object-Oriented Programming, pp 159-165, Verlag Heidelberg.

Page-Jones, M., 2000. Fundamentals of Object-Oriented Design in UML, Addison-Wesley.

SIMPL Project Homepage, https://sourceforge.net/projects/simpl/.

Szyperski, C., 1998. Component Software - Beyond Object-Oriented Programming, Addison-Wesley / ACM Press.

Tyan, H.; Hou, C, 2002. Design, Realization, and Evaluation of a Component-based, Compositional Network Simulation Environment. In: 2002 SCS Western Multiconference on Computer Simulation -- Communication Networks and Distributed Systems Modeling and Simulation Conference.

Webster, B., 1995. Pitfalls of Object-Oriented Development, M&T Books, New York, 1995

Matheus Leite received his MSc in Software Engineering in 2003 and currently is a researcher at the Laboratory of Software Engineering, PUC-Rio, Rio de Janeiro, Brazil.

Luiz Fernando Rodrigues received his CS Bachelor degree in 2004 and currently is an M. Sc student at the Laboratory of Software Engineering, PUC-Rio, Rio de Janeiro, Brazil.

Carlos Lucena is a full Professor of the Department of Computer Science and the coordinator of the Laboratory of Software Engineering, PUC-Rio, Rio de Janeiro, Brazil.

João Magalhães received his MSc in Software Engineering in 2002 and currently is a researcher at the Laboratory of Software Engineering, PUC-Rio, Rio de Janeiro, Brazil.