The Architecture of Federations From Process to Software

(Position Paper)
First Working IFIP Conference on Software Architecture
22-24 February 1999, San Antonio, TX, USA

J. Estublier, Y. Ledru, P.Y. Cunin
L.S.R. Actimart, Bat 8, Av de Vignate
38610 Gieres FRANCE
{jacky|ledru|cunin}@imag.fr

Abstract

We define a federation as a software application built mainly from COTS. A COTS, being designed to be executed stand alone, mainly interacts with users and reacts to changes perceived in its environment. These characteristics make a COTS, to a large extent, similar to a Process Support System (PSS).

In our previous work, we have addressed the issues raised by the architecture and interoperability paradigms required to build federations of process support systems. This position is a preliminary attempt to assess to what extent our contribution applies to software federations.

It is shown, through examples, that the concepts and architecture identified in PSS federations are relevant for software federations; it is our belief (and future work) that our architecture can be used as a reference framework for any federation.

1 Federations

We define a federation as an application built mainly from Components Of The Shelf tools (COTS) which implies that they are (largely) autonomous, and that their source code is not available.

The goal of a federation is to provide collectively a (complex) service while preserving the independence and autonomy of components, and to be open to changes in the composition and distribution (new/changed/removed/moved components).

Indeed, today, many such COTS are available like text editors, spreadsheets, databases, web browsers, workflow, groupware, network services,... Building a software application consists often in building a federation where most components are COTS and only a few ones are application specific.

A COTS is a software component, and as such contains a local state, persistent or not, a number of methods that can change that state and an API giving access to a sub-set of the methods; but, being designed to be used in isolation, it directly interacts with the external world (users and/or common computer resources like network, database or file system). Let us call common space the external world.

The fact the component has direct interaction with the common space has deep consequences on its design: it has to behave in a non-deterministic context. COTS designers usually try to identify a number of “abnormal” behaviors, and to identify convenient reactions to these, in a fixed or customizable (i.e. programmable) way.

For example, suppose a file is edited. The file is in the common space (the file system); a copy of which, in an ad-hoc format, is in the local state (editor main memory). If the file is changed by another component during the editing session, the editor will notice the file has a different state (usually when saving the session) and may either refuse to save, overwrite the file, save under another name, ask the user or other. This kind of behavior is to a large extent arbitrary, it is a policy which can (could) be left to the application builder. This kind of behavior can be said to constitute the *process mode*l of the tool.

If a number of COTS, in the same application, deal with same parts of the common world, it becomes critical for these COTS to behave consistently in “all” circumstances. In other words, the process model of each tool should be compatible or customized, and/or there exists a way for the federation, to define and enforce the whole federation behavior. Let us call this federation behavior the *federation process*.

2 PSSs federations

In our previous work, we have addressed the architectural issues raised by federations of process support systems (PSS) [1]. A PSS is a software component which behavior is defined through an explicit *process model*, in a specific formalism.

In a PSS federation, all components deal with different facets of the same problem: process support. As such, many components share a common knowledge (e.g. activity *FixBug* is under way), and interpret that knowledge in different ways (the SCM tool builds a workspace for the activity, the workflow tool adds an activity in an agenda, the planner starts the tasks and allocates resources, and so on). In other words, PSSs knowledge and behavior overlap; they closely interoperate.

This overlap corresponds to a part of the common space of these components. The process of each component explicitly deals with this common space management; putting together different PSSs creates a high risk of inconsistent global behavior. The *federation process* is the process which defines the way the common space evolves.

The difficulties identified for any federation are magnified in PSS federations. The work we have done was to find the architectures allowing PSS federations to be built. In this work we have identified the following components [1] see Fig 1.

We materialized the common space, in a standard format, managed by a *state server* [2]. Each component is responsible to update, if and when needed, its internal state to match changes in the common space. We identified an *event server,* to which each component can subscribe, in order to receive events when relevant actions are performed on relevant parts of the common space.

Seen from outside the federation executes the federation process, as reflected by the common space changes. The challenge is to impose the federation to really execute that process, without loosing its openness nor the autonomy of its components.

We identified a number of managers, either hardwired or controlled by models, which altogether drive and control the federation:

- a *federation process manager,* which enforces the functional federation behavior by modification of the common space.
- a *connection manager* whose purpose is to establish, if required, direct communication between components who do not know each other. Once established, the protocol and communication is up to the components, which supposes the definition of standard or common interfaces.
- a *control manager* which is in charge, in some circumstances, to find which components(s) are to be called, with which parameters, in which order, and how to deal with errors. It also enforces some consistency and protection rules to the whole federation.

Altogether these managers and models constitute the foundation. The general architecture we propose is as follows.

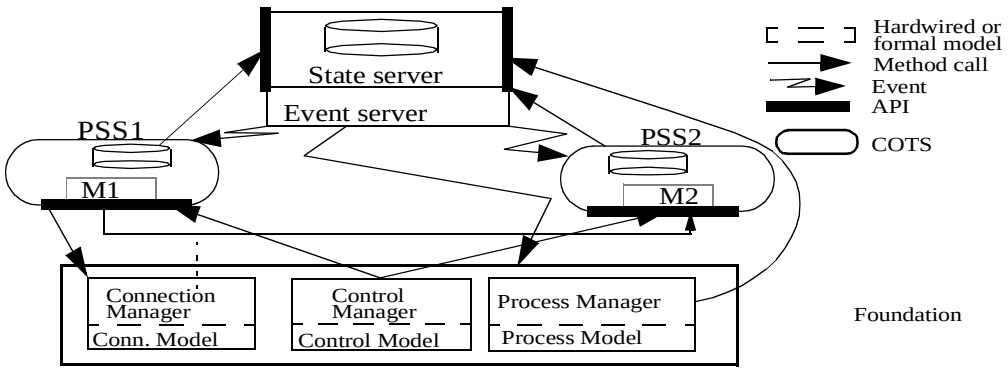

Figure 1: The General PSS Federation architecture

3 Software federation examples

We claim that this general architecture also applies to software federations. We will exemplify using a web browser and editing a composite document on a Windows machine.

3.1 Netscape and helpers

Commercial Web browsers are organized as a software federation. They take advantage of COTS to interpret or visualize the files they get from the web. In Netscape, these components are called “helpers” and have been defined independently. Fig 2 shows the interactions of Netscape with a postscript viewer (Ghostview) and a MPEG player. These interactions are rather simple: typically, Netscape writes a file in the common space (/tmp under UNIX systems) and helpers only read these files.

This federation is a simple instantiation of our general architectural model. The state server corresponds to the UNIX file system; there is no event server; the connection and control managers are included in Netscape itself. The connection model is defined by the table of helpers which defines which helper to call for each type of file; the control manager uses this information to call the right helper with the name of the file. There is almost no process model.

For optimization reasons Netscape also plays the role of helpers because some file types (HTML, GIF) are directly interpreted; these files are not copied into the common state, which shows the difference between the conceptual architecture and its effective implementation.

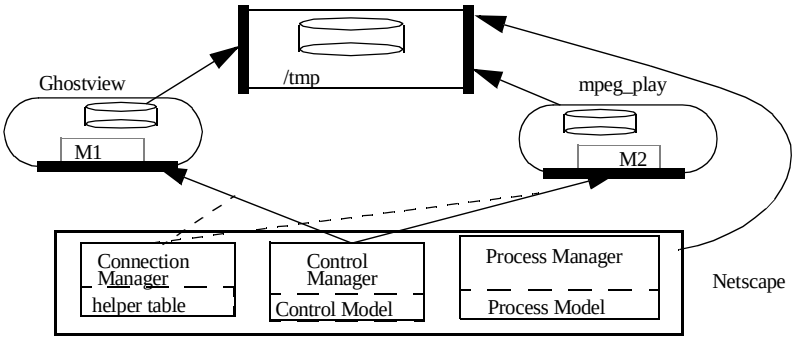

Figure 2: The Netscape Federation

3.2 Composite document (Com/ActiveX)

Let us consider a Word document containing an Excel spreadsheet. This is a federation dedicated to composite document editing: components are the various editors (Word and Excel); they collaborate to a common goal (the document editing) and they ignore each other. The goal of that federation (the federation process) is to provide the end user the illusion the federation is a single editor, whereas a number of autonomous components interoperate to provide that illusion.

In a Windows machine, both tools have the knowledge of the format and semantics of the common state: a “stream” i.e. a complex file containing both Word and Excel data. This example shows that, as soon as close interoperability is required, the common state must be defined a more detailed format and semantics.

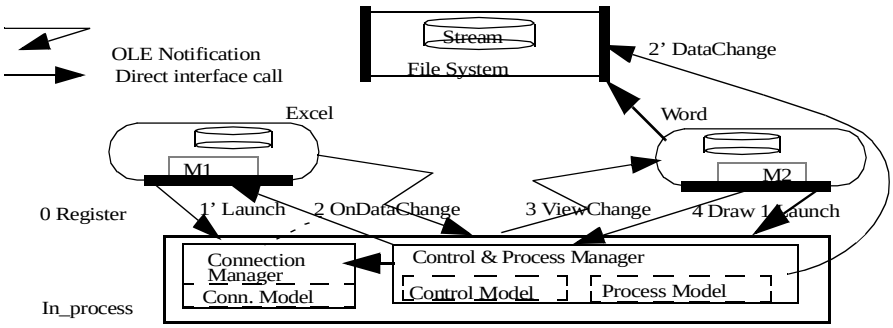

Figure 3: The OLE federation Architecture

In the OLE technology, in (0) components register themselves. In (1), when clicking in the spreadsheet the In_process (control aspect), based on the connection manager information, launchs Excel(1’). In (2), *OnDataChange* is a notification from Excel to the In_process component (Control aspect) which copies the changes into *Stream (2’)*. Then (3) *OnViewChange* is a notification to Word that a part of the spreadsheet bitmap has changed. Word reacts to this notification (4) *Draw*, asking In_process (process aspect) for somebody to (re)draw the changed spreadsheet providing screen location. The In_process component (process manager) knows who can interpret this method (itself).

The OLE technology offers, in a hidden way, the major federation components.

- A *state server*. The common space is a *stream* file containing the global data of both tools, any composite document server is assumed to know its syntax and semantics. The state server is simply the file system.
- An *event server*, using the notification mechanism of COM. In composite document, it is an aspect of the In_process component. There is no subscription, the protocol is predefined.
- A *connection manager* allowing a component to call another one without knowing its name and location (Word does not know that the other component is Excel). The connection server knows how to launch and initialize a component, and how to establish a direct connection between them (exchanging interface addresses).
- *Control and Process manager*. It is in charge of defining the protocol to follow when composite documents are edited (the process of editing). In this example, when it receives the event OnDataChange (2) it copies the data from Excel to the stream and then sends an event to Word (3). When Word asks explicitly for redrawing (4), it decides who will execute it (in the standard case it is In_Process itself).

In this example, using OLE, the foundation is mainly represented by the In_process component, which contains, in a hard-wired way, the common process (it is specialized in composite document editing only), the control process, the notification server (all notifications go through In_process) and the connection process. The process server is based on an agreement, from all components, on the syntax and semantics of Streams.

The fact that OLE/ActiveX is 1) strictly based on standard interfaces and method calls and 2) the fundamental components and their functions are not clearly identified, makes the federation difficult to understand, and the solution proprietary. It is our belief that a better identification of the basic foundation components would have provided an easier and more open way to define and control the federation (Fig 4). Actually, in [3] these mechanisms are expressed in terms closer to the logical architecture of Fig 4 than its implementation described in Fig 3.

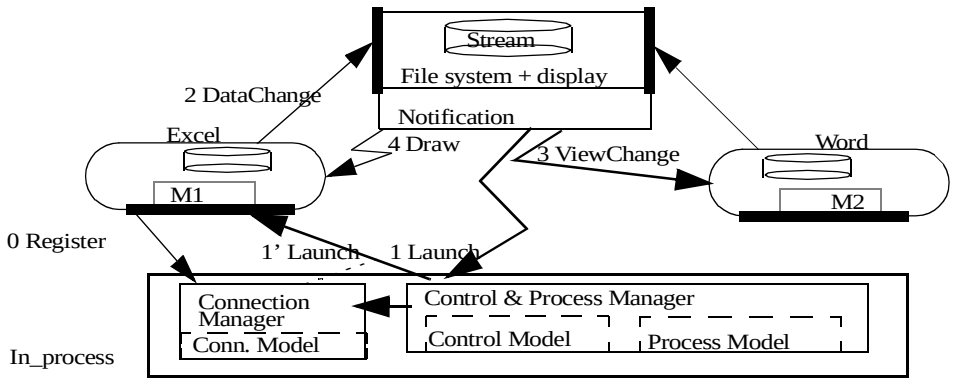

Figure 4: The Logical OLE federation Architecture

4 Conclusion

This early work shows that most of the basic components we have identified in PSS federations, are also present in software federations. Current work aims to see if the following claims are true:

- All federations are instantiations of our general architecture. This paper has verified this claim on two examples. We have informally studied other software architectures (Netscape and plug-ins, CASE tools, e_mail, Corba and others). The two examples presented don't cover all aspects of our generic architecture. In particular, none of these exhibits direct connections between the components and the foundations in both examples only include a very simple process model. Covering other examples would exhibit these aspects: a vast majority of federations rely on direct connections between components (e.g. federations of Corba components); complex process models appear in more specific federations, like federations of CASE or CAD tools, or in enterprise applications such as BAAN or SAP.
- As shown in Fig 4, the identification of the servers and managers of a federation helps in designing and understanding the federation principles. This identification may lead to distinguish between the conceptual and real (efficient) architecture.
- At longer term, can this federation model provide the basis for a general architecture model? In other words, can we see any architecture as a special case of federation?

We expect to be able to report shortly on these future works and experiments.

Acknowledgments

This work has been perfomed within the common laboratory between Dassault Systèmes and LSR/IMAG.

References

[1] Jacky Estublier, Pierre-Yves Cunin, Noureddine Belkhatir. *An architecture for process support interoperability*. ICSP 5, Pages 137-147. 15-17 June 1998 Chicago, Illinois, USA.

[2] D. Heimbigner. “The ProcessWall: a Process State Server Approach to Process Programming”. ACM-SDE, December 1992.

[3] Inside ActiveX and OLE. David Chapell. Microsoft Press. 1996