Published in Critical Issues in User Interface Engineering, P. Palanque & D. Benyon Eds., Springer-Verlag: London Publ., ISBN 3-540-19964-0, 1995, pp. 191-209.

Agent-Based Architecture Modelling for Interactive Systems

Coutaz, J., Nigay, L. & Salber, D.
LGI-IMAG

BP 53, 38041 Grenoble Cedex 9, France
email: {Joelle.Coutaz, Laurence.Nigay, Daniel.Salber}@imag.fr
tel. +33 76 51 48 54

Abstract
Architectural modelling is becoming a central problem for large, complex systems. With the advent of new technologies and user-centred concerns, the user interface portion of interactive systems is becoming increasingly large and complex. In this article, we discuss agent-based architectural styles and show how sound tradeoffs between conflicting requirements and properties can be done using the PAC-Amodeus conceptual model.

1. Introduction

A software architecture is an organisation of computational elements and the description of their interactions. Although specialists still argue about a precise definition of this concept, it is widely recognised that software architecture designs can no longer simply emerge from craft skills. The increasing complexity and size of software systems require sound engineering principles and frameworks to formally structure the design process into multiple but consistent perspectives. Possible perspectives on software architectures include the functional partitions of the system, the structural view in terms of components and connectors, the dynamic coordination model [Abowd 1994]. All of these representations should be the result of an explicit design rationale. They should be analyzable, communicable, and maintainable.

Architectural styles such as pipes and filters, object-oriented organisations and layered structures, can be used as generic vehicles to express architectural solutions. An architectural style makes recurring organizational patterns explicit [Garlan 1993]. A style defines a vocabulary of design elements, imposes configuration constraints on these elements, determines a semantic interpretation that gives meaning to the system description, and enables analysis of the system properties. As a result, architectural styles not only lead to different design solutions but to designs with significantly different software properties [Shaw 1995].

In the domain of user interface, a number of architectural styles have emerged. Seeheim, Arch, MVC, PAC, and others can be considered as canonical references. They primarily provide a framework for performing functional partitioning and allocation of function to structural components based on system and user-centred properties. One lingering problem in user interface software design is that off-the-shelf tools such as interaction toolkits, application skeletons, and user interface builders do not make explicit the link between the facilities they provide and the underlying architectural framework they convey. The software architecture is lost in the resulting code or difficult to extract:

- Interaction toolkits such as Motif [OSF 1989] do not embed the architectural principles that software designers need to build interactive systems effectively. For example, the "call-back procedure" paradigm made popular by X Window [Scheifler 1986], does not enforce the distinction between task domain concepts and presentation specific issues. Thus, without an adequate software framework, the resulting interactive system may be an incredible mixture of concerns.

- In order to exploit object-oriented application frameworks like MacApp [Schmucker 1986], the programmer needs to reverse-engineer the architecture of the existing code. This task can be made easier if the underlying organisation of the environment were made explicit to programmers. A good example is the MVC model that structures the Smalltalk programming environment in a systematic way.

- User interface generators, which they alleviate the programming task, tend to provide a false sense of confidence that software architecture is no longer an issue. We call this attitude "the ABS syndrom": a false sense of security. It is true that some implicit architecture is embedded in the code generated by such tools. However, the software designer must understand the functional coverage of the generated code in order to devise what needs to be developed by hand. In addition, the developer has to discover how to integrate and coordinate the hand-coded portion with the generated code in a way that supports the design requirements. Without an architectural framework to structure the problem, it is difficult to properly achieve this task.

Software tools for the construction of user interfaces will not eliminate architectural issues as long as the construction of user interfaces requires programming. Clearly, developers and maintainers of interactive systems need to rely on canonical models for identifying software components, for organising their interconnections, for reasoning about them and for maintaining them in a productive way.

The literature shows a wide variety of architecture styles for interactive systems revealing distinct goals, usages, and scientific beliefs. The purpose of an architectural description may be to support assessment against specific properties as described in the SAAM method [Kazman 1994]. Another goal may be to communicate a design solution to another development team. Then the description should be unambiguous, with possibly clear reference to the implementation tools, and no opportunity for misinterpretations. In a nutshell, an architecture design expresses what is important[1]. And what is important depends on the purpose.

In this chapter, we present agent-based architectural styles for the purpose of assessing software designs. In Section 2, we discuss the concept of agent as well as the general properties that agent-based models tend to support. In Section 3 we present our own work in this area: the principles of the PAC-Amodeus model. In 4, we show how PAC-Amodeus can be used in practice to reconcile conceptual design with software properties and requirements.

2. Agent-Based Models

Agent-based models structure an interactive system as a collection of specialised computational units called agents. An *agent* has a state, possesses an expertise, and is capable of initiating and reacting to events. Agents that communicate directly with the user are sometimes called *interactors*. An interactor provides users with a perceptual representation of its internal state. The terms interactor and agent are sometimes used indifferently even if there is no direct interaction with the

[1]Len Bass from SEI Carnegie Mellon University, should be thanked for this remark

user. Interactors are also coined as *interaction objects*. An object is a generic term that covers a computational element with a local state. It can either be viewed as a concept or as the technical structure that underpins the object-oriented programming paradigm. In the following discussion, we will consider an object as a generic concept.

Our view of the concept of agent is one perspective of the more general definition used in distributed Artificial Intelligence (A.I.). In A.I., agents may be cognitive or reactive depending mainly on their reasoning and knowledge representation capabilities [Demazeau 1991]. A cognitive agent is enriched with inference and decision making mechanisms to satisfy goals. At the opposite, a reactive agent has a limited computational capacity to process stimuli. It has no goal per se but a competence coded (or specified) explicitly by the human designer. In current interactive systems, agents are reactive. In the following discussion, we will not make the distinction between cognitive and reactive agents although current models developed for the software design of user interfaces have considered reactive agents only.

Having presented the vocabulary of the agent-based style, we need to discuss the advantage of this modelling technique and compare several of the approaches using this style as a basis.

2.1. Benefits from the agent-based style

Agent models stress a highly parallel modular organisation and distribute the state of the interaction among a collection of co-operating units. Modularity, parallelism and distribution are convenient mechanisms for supporting the iterative design of user interfaces, for implementing physically distributed applications, and for handling multi-thread dialogues:

- An agent defines the unit for functional modularity. It is thus possible to modify its internal behaviour without endangering the rest of the system.

- An agent defines the unit for processing. It is thus possible to execute it on a processor different from the processor where it was created. It is also possible to use instances of a class of agents to present a concept on distinct workstations. This property is essential for implementing groupware.

- An agent can be associated to one thread of the user's activity. Since a state is locally maintained by the agent, the interaction between the user and the agent can be suspended and resumed at the user's will. When a thread of activity is too complex or too rich to be represented by a single agent, it is then possible to use a collection of co-operating agents.

In addition to satisfying requirements for better user interfaces, agent models can easily be implemented in terms of object-oriented languages: an object class defines a category of (reactive) agents where class operators and attributes respectively model the instruction set and the state of an agent category, and where an event class denotes a method. An object and an agent are both highly specialised processing units, and both decide about their own state: a state is not manipulated by others but results from processing triggered by others. The sub-classing mechanism provided by object-oriented languages can be usefully exploited to modify a user interface without changing the existing code.

A number of agent-based models and tools have been developed along these lines. MVC [Goldberg 1984], PAC [Coutaz 1987], ALV [Hill 1992], the CNUCE [Paterno' 1994] and York [Duke 1993, Duke 1994] models, are typical agent-based styles. Interviews [Linton 1986] and Aïda [Ilog 1989] are examples of toolkits based on such a model, whereas Serpent [Bass 1991] and Sassafras [Hill 1986] are run time kernels and user interface generators organised as a multi-agent structure. All of these models and tools push forward the separation of concerns advocated by seminal Seeheim [Pfaff 1985]. They generalise the distinction between concepts and presentation techniques by applying the separation at every level of abstraction. In order to do so, they distribute the separation of concerns among co-operating agents. They differ however in the way they perform the distribution and the co-operation. This is the topic of the next section.

We propose to compare agent-based models according to their functional structure and according to the MSM dimensions [Coutaz 1993].

2.2. Comparative analysis based on agent structures

In MVC (Model, View Controller), an agent is modelled along three functional perspectives: the Model, the View, and the Controller. A Model defines the abstract competence of the agent (i.e., its functional core). The View defines the perceivable behaviour of the agent for output. The Controller denotes the perceivable behaviour of the agent for inputs. The View and the Controller cover the user interface of the agent, that is, its overall perceivable behaviour with regard to the user.

PAC (Presentation, Abstraction, Control) conveys similar ideas: the facets of an agent are used to express different but complementary and strongly coupled computational perspectives. A PAC agent has a Presentation (i.e., its perceivable input and output behaviour), an Abstraction (i.e., its functional core), and a Control to express dependencies. The Control of an agent is in charge of communicating with other agents as well as expressing dependencies between the Abstract and Presentation facets of the agent. In the PAC style, no agent Abstraction is authorized to communicate directly with its corresponding Presentation and vice versa. In PAC, dependencies of any sort are conveyed via Controls. Controls serve as the glue mechanism to express coordination, formalism transformations that sit between abstract and concrete perspectives.

ALV (Abstraction, Link, View) has a similar decomposition to PAC except that a Link has a more restricted role than the PAC Control. An ALV Link is in charge of expressing constraints between the View and the Abstraction whereas PAC Controls may be used to express relationships with other agents. Whereas PAC does not prescribe any form of implementation (i.e., whether an agent should be built from three objects in an object-oriented development environment like MVC, or simply as a single C module), ALV implies the decoupling of the agent into three distinct pieces.

A CNUCE agent possesses four perspectives inherited from the computer graphics modelling techniques: the Collection and the Abstraction facets define the functional core aspect of the agent for input and output respectively; the Measure and the Presentation cover the user interface side of the agent for input and output respectively. In addition, two triggers express the conditions under which input and output may occur.

A York agent has two perspectives : the presentation and its internal behaviour. This decomposition is similar to the structure chosen for InterViews.

In summary, MVC and CNUCE decouple input techniques from outputs, whereas ALV, PAC, and York concentrate them in the notion of Presentation or View. Contrary to PAC and ALV, MVC has no explicit notion of arbitrator for expressing the relationships and the co-ordination between agents. In the Lisbon model [Duce 9191], agents do not have perspectives but are organised as specialised categories to model the user interface portion of an interactive system.

2.2. Comparative analysis along the MSM dimensions

The MSM (Multi-Sensori-Motor) framework has been devised to support reasoning about current and future interactive systems [Coutaz 1993]. The framework is comprised of 6 dimensions. The first two dimensions deal with the notion of physical devices: the number and the nature of these devices (i.e., input vs output). The other four dimensions are used to characterise the degree of built-in cognitive sophistication of the system: levels of abstraction, context, fusion/fission, and parallelism.

2.2.1. Input and output

All of the models considered in our discussion stress the fact that an agent may communicate with multiple agents. Agents have multi-channel capabilities. However, not all of the agent-based models make explicit the communication channels their agents support. For example, MVC does not say how communication occurs between the agents of an interactive system. The question is left opened until implementation.

In PAC, on the other hand, the Control facet of an agent supports communication in two ways: first, it is as an explicit bridge between the two facets it serves (the Abstraction and the Presentation may use different formalisms as well as distinct time basis); second, it is used as the switchboard of the agent: it receives inputs and outputs from other agents. At the opposite of CNUCE, however, PAC does not make explicit how inputs and outputs are processed and dispatched within the agent.

In CNUCE and York, input and output channels are clearly expressed. For example, the Measure of an agent is modelled as the local process specialised in processing inputs from lower level agents. The Collection has a similar role for processing inputs from higher levels of abstraction. The Presentation and the Abstraction are the output channels for lower levels and higher levels of abstraction respectively.

2.2.2. Levels of abstraction

The notion of level of abstraction expresses the degree of transformation that the interpretation and rendering functions perform on information. The sequence of input transformations forms the interpretation function whereas in the other direction, internal information (e.g., system state) is transformed to be made perceivable to the user by a sequence of output transformations called the rendering function. The notion of level of abstraction also covers the variety of representations that the system supports, ranging from raw data to symbolic forms.

Information acquired by agents is transformed by a population of agents before reaching the functional core of the system. In the other direction, agents concretise information from to functional core into perceivable behaviour. The successive steps of such input and output transformations define levels of abstraction. In the conceptual architectural models considered in our discussion, abstracting and concretising are performed by agents organised into levels of abstraction.

Within CNUCE, levels of abstraction are defined in two ways: within an agent and in the form of a composition mechanism.

- A CNUCE agent stresses a clear distinction between the rendering and the interpretation functions. These functions operate along a 2 step process. Each step defines a level of abstraction within the agent. For rendering, information from higher levels of abstraction is received by the Collection and delivered to lower levels by the Presentation. For interpreting, the Measure receives information from lower levels whereas the Abstraction communicates with higher levels.

- CNUCE agents can be composed into more powerful, abstract agents with composition operators such as the interleaving operator [Paterno' 1994]. By doing so, an agent is defined as a hierarchy of agents composed of a parent and a set of interleaved child agents.

PAC and MVC adopt a similar approach for modelling levels of abstraction. Like CNUCE, they do not make explicit the nature of these levels. The Lisbon model, on the other hand, introduces classes of agents, i.e., Conceptual objects (CO’s), Interaction Objects (IO’s), and Transformer Objects (TO’s) as an indication of what these levels might be.

In the Lisbon model, CO’s are “the user interface accessible representation of an object the functional core wishes to make visible” [Duce 1991]. In the user interface portion of an interactive system, CO’s are the proxies of the task domain concepts manipulated by the functional core. IO’s support the translation process between CO’s inputs and outputs and low level input and output devices. As in CNUCE, “Composite IO’s may be formed from single IO’s to handle arbitrarily complex threads of dialogue” [Duce 1991]. TO’s provide the basic mechanism for managing the relations between different objects of the user interface. Examples of such relations include constraint management to maintain consistency between IO’s, context switching between dialogue threads, integrity mapping between CO’s and IO’s, etc. TO's are in charge of similar functions as PAC Controls.

2.2.3. Context

The capacity of a system to interpret and render information may vary dynamically with respect to “contextual variables”. Contextual variables are like cognitive filters. They form a set of internal state parameters used by the representational processes to control the interpretation/rendering function. Agents, which are the processes of the user interface, are good candidates for being the owners of local contexts.

2.2.4. Fusion and fission

Fusion refers to the combination of several chunks of information to form new chunks. Fission refers to the decomposition phenomenon. Fusion and fission are part of the abstracting and materialization phenomena.

The composition mechanism within York, MVC, PAC, and CNUCE was primarily introduced for the expression of levels of abstraction. One side effect of this mechanism is that composition provides a sound foundation for fusion and fission. Fusion and fission are part of the internal processing capabilities of an agent (which owns contexts to perform its computation). A more detailed discussion about fission and fusion can be found in [Coutaz 1993]

2.2.5. Parallelism

Representation and usage of time is a complex issue. In our discussion, we are concerned with the role of time within the interpretation and rendering functions. How does time relate to levels of abstraction and contexts? How does it interfere with fusion and fission? Parallelism at the user interface may appear at multiple grains: at the physical level (multiple I/O devices may be used simultaneously), at the task level (the user or the system may execute multiple tasks in parallel), and at the task cluster levels (the user or the system may carry multiple types of activities at the same time or in an interleaved way).

An agent is a processing unit that can operate in parallel with other agents. It results from agent-based architectures that multiple interpretation and rendering activities can take place simultaneously. Therefore, agents can be defined to represent parallelism at the appropriate grain of concurrency. It is important to note however that, at the implementation level, an agent is not necessarily an actual process (in the sense of the underlying run-time platform).

Allocation in architectural design is a multiple step activity. It includes the allocation of functions to structures, the allocation of structures to system processes, and the allocation of processes to physical processors. Allocation depends on non functional requirements such as the characteristics of the run-time platform. Clearly, implementing an agent as a Unix process would be inefficient, but at the conceptual level of design, one should view an agent as an entity capable of initiating actions in parallel to other agents.

Having presented general issues about agent-based styles, we now describe our own approach: PAC-Amodeus.

Published in Critical Issues in User Interface Engineering, P. Palanque & D. Benyon Eds., Springer-Verlag: London Publ., ISBN 3-540-19964-0, 1995, pp. 191-209.

3. PAC-Amodeus: the principles

PAC-Amodeus uses the Arch model [Arch 1992] as the foundation for the functional partitioning of an interactive system and populates the key element of this organisation, i.e., the Dialogue Controller, with PAC agents.

Basically, Arch is a revisited Seeheim that provides hooks for reasoning about software engineering properties such as code re-usability, portability, and modifiability. PAC-Amodeus incorporates the two adaptor components of Arch, the Interface with the Functional Core and the Presentation Techniques Component, to insulate the keystone component (i.e., the Dialogue Controller) from modifications in its unavoidable neighbors: the Functional Core and the Low Level Interaction Component. A detailed discussion on the advantages of these adaptors will be presented in Section 4.

The Arch model however, does not provide any guidance about the decomposition of the Dialogue Controller nor does it indicate how the MSM features (such as parallelism) can be supported within the architecture. PAC, on the other hand, stresses the recursive decomposition of an interactive system, in terms of agents, but does not pay attention to engineering issues such as the existence of implementation tools. PAC-Amodeus gathers the best of the two worlds. Figure 1 shows the resulting structure.

Figure 1: The PAC-Amodeus functional components.

As shown in Figure 1, PAC-Amodeus adopts the same overall structure than Arch but goes one step further in two ways:

(1) PAC-Amodeus makes explicit the boundaries of the Presentation Techniques and the Low Level Interaction components using the notions of physical device and interaction language. A *physical device* is an artefact of the system that acquires (input device) or delivers (output device) information. Examples of devices include the keyboard, mouse, microphone and screen.

An *interaction language* defines a set of well-formed expressions (i.e., a conventional assembly of symbols) that convey meaning. The generation of a symbol, or a set of symbols, results from actions on physical devices. Examples of interaction languages include pseudo-natural language and direct manipulation.

In PAC-Amodeus, the overall functional partitioning of an interactive system should be defined according to the following rule: the Low Level Interaction Component should be device dependent and language dependent; the Presentation Techniques Component is device independent but still language dependent; the other components of the interactive system, including the Dialogue Controller, should be both device and language independent.

(2) PAC-Amodeus decomposes the Dialogue Controller into a set of cooperative PAC agents. The set is derived for a particular system using the heuristic rules presented in Section 4. These rules, which consider the external specifications of the system as the driving rationale, offer a bottom-up approach to the process of functional decomposition. Other rules, based on a task description, support a top-down partitioning process [Paterno' 1994]. Both approaches are valid. If the external specifications are consistent with the task model, bottom-up and top-down reasoning should lead to the same set of cooperative agents.

Although agents provide a sound approach to the refinement of an Arch Dialogue Controller, we need to clarify how they are connected to the functional neighbours of the Dialogue Controller. As shown in Figure 2 an agent is related to the Interface with the Functional Core (IFC) and to the Presentation Techniques Component via its Abstraction and Presentation facets. The Abstraction is connected to one or multiple entities of the IFC. Depending on the case at hand, the connector is implemented as a procedure call, as a pointer, or as any other protocol suitable for the system requirements. Similarly, the Presentation facet of an agent is connected to one or multiple entities of the Presentation Techniques Component.

In summary, the hierarchical organisation of the Dialogue Controller in terms of PAC agents is motivated by the necessity of modelling computation at various levels of abstraction, the necessity of expressing the fusion and fission phenomena as well as parallelism at various levels of granularity. In addition to the hierarchical information flow, (and in contrast with the original PAC model [Coutaz 1987]), communication can also occur horizontally at various levels of abstraction through the Abstraction and Presentation facets of the PAC agents.

Figure 2: A PAC agent of the Dialogue Controller. Dashed lines represent possible relationships with other agents. Dimmed arrows show the possible links with the surrounding components of the Dialogue Controller.

PAC-Amodeus focusses on the refinement of the Dialogue Controller but leaves open the analysis of the other four functional components. Actually, these components are also concerned by the MSM dimensions (e.g., parallelism, context, levels of abstraction, fusion and fission). As a result, the agent-style applies equally well to all of the components of the Arch. We have chosen not to do so for several reasons: either these components are fully covered by dedicated implementation tools (therefore, the architecture is already there) or they are partially covered by the development environment (then, it is difficult to provide systematic rules at the conceptual level that fit every situation). It is however possible to model an interactive system in a systematic way using an agent style to prove user-centered properties as in the interactor theory [Duke 1993, Duke 1994]. If, on the other hand, the goal is to formulate an architecture with the perspective of an implementation using off-the-shelf tools, then the hybrid PAC-Amodeus model, which combines agents and gross functional components, is a good candidate to start with.

4. PAC-Amodeus in Practice

Conceptual models like PAC-Amodeus are ideal views of a complex world. They are very attractive as style guides in a book but are difficult to relate to the practical activity of design. This section shows how PAC-Amodeus can be used to reason about a particular design when confronted with practical software requirements. First, we present possible usages of the PAC-Amodeus adaptors, then we comment on the lingering problem of allocating functions to structural components. In 4.3, we close the discussion with our set of heuristic rules for refining the Dialogue Controller in terms of PAC agents.

4.1. Interpreting the PAC-Amodeus Adaptors

PAC-Amodeus reuses the two adaptors of the Arch model: the Interface with the Functional Core and the Presentation Techniques Component.

4.1.1. The Interface with the Functional Core: Temporal Strategies and Data Exchanged

The Interface with the Functional Core (IFC) serves as a mediator between the Dialogue Controller and the domain-specific concepts implemented in the Functional Core. It is designed to absorb the effects of changes in its direct neighbors. As any boundary, it implements a protocol. A protocol is characterized by temporal strategies and by the nature of data exchanged [Coutaz 1991].

A *temporal strategy* defines the coordinating rules for transfering information between two communicating entities such as the Functional Core and the Dialogue Controller. The coordination may be fully synchronous or fully asynchronous or may alternate between the two extremes. Synchronous coordination implies that the sender waits for the receiver before its own processing can resume. In the context considered here, synchronous coordination models the mutual control of the Functional Core and the Dialogue Controller: either one has the initiative, but the initiator is directly controlled by its partner. Asynchronous coordination allows communicating entities to exchange information without waiting for each other. With such a communication scheme, the Functional Core and the Dialogue Controller are two equal partners sharing a common enterprise: that of accomplishing a task with the user.

Using a synchronous scheme, either the Functional Core controls the interaction in a query-answer style, or the Functional Core is controlled by the user interface. Most legacy functional cores, which perform the well-known Pascal "writeln's" and "readln's", comply to the query-answer style. On the other hand, implementing a user interface on top of X Window forces to model the

Functional Core as a passive server of the user interface. Possible conflicting requirements may emerge from this situation, for example, upgrading the user interface of a legacy system using X Window. If so, the Functional Core and the user interface should be run by distinct processes and the Interface with the Functional Core should be used to synchronise these processes.

From the user's point of view, synchronous coordination results in single threads of dialogue or, at best, in interleaved threads. If mono-threading is not consistent with the user's requirements, then asynchronous coordination should be adopted. Asynchronous coordination supports multiple concurrent threads of dialogue: the user may issue multiple commands simultaneously or in an interleaved way while the Functional Core may have its own processing going on. Functional Cores that implement dynamic systems such as a robot or a nuclear plant, and provision for end-user's control over the on-going process, call for asynchronous coordination. In turn, asynchronous coordination leads to allocate the Functional Core to a dedicated process.

Exchange of data between the Functional Core and the user interface is performed through the IFC in terms of domain objects. A domain object is an entity that the designer of the Functional Core wishes to make perceivable to, and manipulable by the user (cf. the CO's of the Lisbon Model). Ideally, it is supposed to match the user’s mental representation of a particular domain concept. It may be the case however that the Functional Core, driven by software or hardware considerations, implements a domain concept in a way that is not adequate for the user.

Semantic enhancement [Bass 1991] may be performed in the IFC by defining domain objects that reorganize the information modeled by the Functional Core. Reorganizing may take the form of aggregating data structures of the Functional Core into a single domain object or, conversely, segmenting a concept into multiple domain objects. It may also take the form of an extension by adding attributes and operators, which can then be exploited by the other components of the user interface. Note that the CO's of the Lisbon Model fit in the Interface with the Functional Core.

4.1.2. The Presentation Techniques Component

The Presentation Techniques Component (PTC) acts as a mediator between the Dialogue Controller and the Low Level Interaction Component (LLIC). Because it is device independent, it is generally viewed as a logical LLIC. As shown in Figure 3, we propose to refine the PTC into two layers of abstraction: the Extension Layer and the Interaction Toolkit Adaptor. The Interaction Toolkit Adaptor defines a virtual toolkit used for the expression of presentation objects. This expression is then mapped into the formalism of the actual interaction toolkit used for a particular implementation. Switching to a different toolkit requires rewriting the mapping rules, but the expression of the presentation objects remains unchanged.

Figure 3: Refinement of the Presentation Techniques Component.

Interaction objects are generally constructed from entities made available in interaction toolkits. In general, interaction toolkits such as the X Intrinsics [OSF 1989], provide an abstraction mechanism for defining new interaction objects. However, it is not always possible to build new interaction objects from the predefined building blocks of the toolkit. For example, in an earlier version of the X Intrinsics, widgets (i.e., interaction objects) would occupy rectangular areas only. Under such conditions, the notion of a wall in a floor plan drawing editor could not be implemented as a diagonal line widget. Instead, a presentation object "wall" would have to be defined as a new presentation object outside the toolkit.

The wall example shows that the Presentation Techniques Component should, conceptually, be structured into two layers. Specific-interaction objects, which can be built from the building blocks of the toolkit, should belong to the toolkit. Those which cannot be built with the toolkit should be part of the Extension Layer. The Interaction Toolkit Adaptor, such as XVT [Valdez 1989] defines the boundary between these two layers.

Sometimes, the location for implementing a presentation object is not as straighforward as the wall example. If located in the toolkit, then the presentation object becomes a general purpose interaction object and thus, should be implemented according to the programming rules imposed by the toolkit to guarantee reusibility. If located in the extension layer, the private status of the presentation object relaxes the reusability constraints of the underlying platform.

This example is merely a simple illustration of a more general problem: that of identifying the appropriate location for software functionalities.

4.2. Allocating functions to structural components

Experience shows that no single architectural model can satisfy all of the software design factors and criteria such as those reported by McCall [McCall 1977]. Factors and criteria may be conflicting. For example, portability and modifiability may impede efficiency. In PAC-Amodeus, the two adaptor components have been reused from the Arch model to minimize dependencies from the modifications of the Functional Core as well as from user interface toolkits. Conversely, these abstract machines may have an adverse effect on the speed of the run-time end product.

Another example of conflict occurs between the allocation of functionalities and response time. The foundations set up by the Seeheim model stress that domain-specific objects should be confined within the Functional Core and its extension, the IFC. It results from this principle that the semantic quality of feedback for a single user-system transaction may require many round trips between the user interface portion of the system and the Functional Core. This long chain of data transfer between the components of the user interface to reach the Functional Core and vice versa, plus possible process boundaries crossing, may be costly with respect to the system response time. Therefore, it may be inconsistent with the expectation of the end-user.

Domain-knowledge delegation[2], which consists of down-loading Functional Core knowledge into the user interface, is a way to reduce transmission load at critical points [Coutaz 1991], therefore to improve response time when this criteria has been identified as an important requirement. For example, rubber-banding in direct manipulation interfaces, requires high performance at the user interface. In particular, if the Functional Core is implemented as a distinct process running on a distinct processor, it may be judicious to delegate domain knowledge into the user interface portion of the interactive system. By doing so, semantic knowledge is readily available in the user interface

[2]The term "semantic delegation" was previously used by Coutaz to denote "domain-specific delegation".

and can be rendered to the user within the response time constraint. Communication with the Functional Core can be postponed when response time is not critical.

The Slinky metamodel acknowledges the fact that software architectures for interactive systems must be tailored to the requirements and criteria selected for the particular case at hand [Arch 1992]. “The term “Slinky” was selected to emphasize that functionalities can shift from component to component in an architecture depending on: - the goals of the developers, - the weighting of development criteria, -the type of system to be implemented. This concept is loosely represented by the flexible Slinky™ toy.” [Arch 1992, pp. 35].

Thus, the Slinky metamodel is a generic framework from which particular instances of arches can be derived. For example, if efficiency prevails against toolkit portability, then the Interaction Toolkit Adaptor may not be needed. If the Functional Core provides the "appropriate interface" in accordance with the user's requirements, and if the Functional Core will not evolve in the future, then the Interface with the Functional Core can be scaled down to a simple connector (e.g., a set of procedure calls). If high quality semantic feedback and efficiency are important requirements as in semantically rich rubber-banding tasks, then domain-knowlege delegation may be performed. Such decision may result in reducing the relative importance, thus the code size, of the Functional Core.

PAC agents of the Dialogue Controller provide a way to perform domain-knowledge delegation without jeopardizing the basic “separation of concerns” principle. The Abstraction facet of agents can be used to locate domain-dependent information. This information may be a copy of the original information maintained in the Functional Core or a copy from an adapted version maintained in the Interface with the Functional Core, or even the domain-knowledge per se (with no equivalent copy in the Functional Core). Duplicating information is one way of improving local efficiency. However, it introduces an additional complexity for maintaining consistency between the copies in the Dialogue Controller and the Functional Core (or the IFC). If consistency is not guaranteed then honesty [Abowd 1992] is not satisfied. If consistency is difficult to maintain, then the designer may decide to not implement the domain concept in the Functional Core and use a PAC agent instead. Pushing the rationale further, the Functional Core may be empty by allocating and distributing its functional capabilities across a set of PAC agents.

4.3. Heuristic Rules for devising PAC agents

The following rules have been devised for PAC-Amodeus and have been implemented in the form of an expert system, PAC-Expert [Nigay 1994]. As mentioned above, they imply a bottom-up analysis starting from the external specifications of the system. They are organised along four issues: window existence, window content, window links, and hierarchy revision.

4.3.1. Window Existence

Generally speaking, a window is a rendering surface for displaying information on the physical screen. A distinction should be made between main-dialogue-thread windows, which display domain concepts exported from the Functional Core, and convenience windows, such as dialogue boxes and forms used in sub-dialogues to inform users that an abnormal condition has occurred or to offer them the opportunity to enter the parameters of a command.

Rule 1: Model a “main-dialogue-thread window” as an agent.

The Presentation facet of an agent of type "main-dialogue-thread window" manages the window itself (that is the interaction technique "window" offered by the toolkit component), including the title and the windowing commands such as the resize and move functions. In addition, if the agent is a leaf in the PAC hierarchy, its presentation facet also manages the presentation of the concepts it renders to the user.

If a "main-dialogue-thread window" agent is a leaf in the PAC hierarchy, its Abstraction facet maintains an abstract representation of the concepts (i.e., the domain objects) it renders or at least the links between these concepts.

We observe that, in general, the presentation of a hierarchy of concepts is displayed within the window technique associated with the "main-dialogue-thread window" agent. Although general, this property is not always true. It may be the case that the selection of the representation of a sub-concept in the "main-dialogue-thread window" agent opens a new window. In turn, this window, which displays domain objects, is modelled as a child "main-dialogue-thread window" agent.

Rule 2: Use an agent to maintain visual consistency between multiple views.

If multiple views about the same concept are allowed and if each view is modelled as an agent, then a Multiple View parent agent is introduced to express the logical link between the children view agents. Any user action with semantic and visual side effect on a view agent is reported by the view agent to its parent, the Multiple View agent, which in turn broadcasts the update to the other siblings.

4.3.2. Window Content

It is often the case that the user interface presents a list of the classes of concepts from which the user can create instances. For example, a drawing editor includes the classes circle, line, rectangle, and so forth. In general, these classes are gathered into palettes or tear-off menus. Let's call such presentation techniques "tool palettes". Tool palette agents provide a good basis for extensibility, reusability and modifiability. Note that we must make a distinction between tool palette agents, which render classes of concepts, and main-dialogue-thread-window agents, which represent instances of concepts.

Rule 3: Model a tool palette as an agent.

The Abstraction facet of a tool palette agent contains the list of the classes of instantiable concepts. In general, the Presentation facet of a tool palette agent is built from interaction objects offered by the Low Level Interaction Component. It is in charge of the local lexical feedback when user actions occur on the physical representation of the concept classes (e.g., reverse video of the selected icon). These actions are then passed to the Control facet.

The Control facet of a tool palette agent maps user actions to the list maintained in the abstraction facet. It transforms these actions into a message whose level of abstraction is enriched (e.g., a mouse click on the “circle” icon is translated into the message "current editing mode is circle").

Rule 4: Model the editable workspace of a window as an agent.

A window may contain an area where the user can edit concepts. In this case, this area should be modelled as a "workspace" agent. A workspace agent is responsible for interpreting (1) the user actions on the background of the window, (2) the user actions on the physical representations of the editable concepts when these concepts are not managed by any special purpose agent, (3) messages from the child agents when those agents represent editable concepts. In addition, a workspace agent may be in charge of maintaining graphical links to express logical relationships between the editable concepts. In summary, a workspace agent has the competence of a manager of a set of concepts.

At the opposite, a non-editable area of a set of concepts is not modelled as an agent. It is embedded in the "main-dialogue-thread window" agent which displays these concepts.

Rule 5: Model a complex concept as an agent.

A complex concept may be either a compound concept, or it may have a structured perceivable representation, or it may have multiple renditions.

1. A compound concept is built from sub-concepts, and this construction must be made perceivable to the user. In general, a hierarchy of agents in the user interface maps the composition of the compound concept.

2. When involving a set of rendering items, a simple concept may be modelled as an agent. For example, the concept of wall, whose presentation is built up from a line, a hot-spot (to select it), and a pop up menu (to invoke an operation on the wall), can be modelled as an agent.

3. An elementary concept may be presented in multiple locations. In addition, each presentation may be different. One agent is introduced to maintain the consistency between the multiple presentations.

4.3.3. Window Links

Window links fall into three categories: open links, syntactic links, and semantic links.

Rule 6: If the access to a “main-dialogue-thread window” is allowed from another “main-dialogue-thread window”, a parent-child relation is modelled by an agent.

We use the term “open link” to refer to the relation that describes the possibility for the user to open a “main-dialogue-thread window” from another one. This relationship is controlled by a dedicated agent.

Rule 7: An agent is introduced to synthesised actions distributed over multiple agents.

Windows agents are related by a “syntactic link” when a set of user actions distributed over these windows can be synthesised into a higher abstraction (i.e., the fusion phenomenon). For example, to draw a circle, the user selects the “circle” icon in the tool palette agent, then draws the shape in the workspace agent. These distributed actions are synthesised by a cement agent into a higher abstraction (i.e., the command “create circle”). This agent, which maintains a syntactic link between its sub-agents, is called a “cement agent”.

Rule 8: An agent is introduced to maintain a semantic relation between concepts included in distinct windows.

This rule is identical to the previous one except that the competence of the Semantic agent is to maintain a semantic relation and not to synthesise the user's actions.

4.3.4. Hierarchy revision

Once the hierarchy of agents have been devised according to the rules above, we recommend to analyse the hierarchy using the following rules.

We have observed that users of PAC-Amodeus have difficulty in bounding the recursive decomposition of the agent hierarchy. As a pedagogic material, we recommend the following approach: refine the decomposition down to the elementary interaction objects as specified in the external specifications (e.g., a push button, a menu), then prune the hierarchy using the following rule.

Rule 9: If the development tools used at the LLIC or PTC levels can implement an agent in a straightforward way, then turn the agent into an interaction (or presentation) object and make it part of the Presentation facet of the parent agent (i.e., the facet gets connected to the interaction/presentation object).

Rule 10: If a parent agent has one single sibling, if future evolution of the system does not lead to additional siblings, and if information transfer between agents is not for free, then it may be useful to combine the two agents into a single one.

4.3.5. About agent facets

Our heuristic rules imply that not all agents have a Presentation nor an Abstraction facet. Typically, multiple view agents, which maintain some consistency between their sibblings, have no Presentation and no Abstraction. They are pure controllers.

Other agents, like cement agents, which combine information from multiple sources into higher abstractions, have no Presentation but may have an Abstraction in case domain-knowledge is needed to perform data fusion. In addition, the Abstraction facet may be a simple connector to the IFC or, if there is no IFC, to the Functional Core.

Agents that are pure controllers may not be explicit components of the architectural design. As mentioned in the introduction, a design solution expresses what is important. If by chance, the designer can draw upon a dedicated language or tool to express dependencies such as FLO [Ducasse 1993] and Link [Hill 1992], then the design solution may simply show a connection between the dependent agents. Because the controller is automatically generated by a tool (as opposed to program it explicitly in C code), it is no longer an issue, and it disappears from the architectural design solution. Similarly, the fusion algorithm that we have developed to support multimodal interaction [Nigay 1995], is not part of a PAC-Amodeus architecture. It sits behind the scene as a reusable mechanism.

5. Conclusion

Architectural modelling is becoming a central problem for large, complex systems. With the advent of new technologies and user-centred concerns, the user interface portion of interactive systems is becoming increasingly large and complex. Although interface builders tend to alleviate the problem, they are limited in scope and apply to mundane cases for which the user interface is often a second class component. Architecture design of user interfaces needs better support.

In this chapter, we have focussed on agent-based architectural styles and showed how sound tradeoffs between conflicting requirements and properties can be done using the PAC-Amodeus conceptual model. Although we have provided some practical guidelines for the software design of user interfaces, we do not claim exhaustiveness. In particular, we have not discussed how PAC-Amodeus can be extended to groupware [Salber 1995], and we still do not have operational generic heuristics to bridge the gap between architectural design and implementation tasks.

6. Acknowledgements

Published in Critical Issues in User Interface Engineering, P. Palanque & D. Benyon Eds., Springer-Verlag: London Publ., ISBN 3-540-19964-0, 1995, pp. 191-209.

This work has been supported by project ESPRIT BR 7040 Amodeus II. It was influenced by stimulating discussions with Len Bass (SEI, Carnegie-Mellon University), our Amodeus partners, David Duke and Michael Harrison (York University), Giorgio Faconti and Fabio' Paterno (CNUCE, Pisa), and with participants of the Dagstuhl Seminar 9508 on Software Architecture.

7. References

[Abowd 1992] G. Abowd, J. Coutaz, L. Nigay, Structuring the Space of Interactive System Properties, Proceedings of the *IFIP TC2/WG2.7 on Engineering for Human-Computer Interaction*, Ellivuori, Finlande, 1992, pp. 113-128.

[Abowd 1994] G. Abowd, L. Bass, *"Software Architecture: A Tutorial Introduction"*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, 1994.

[Arch 1992] "A Metamodel for the Runtime Architecture of an Interactive System", The UIMS Tool Developers Workshop, *SIGCHI Bulletin*, ACM, 24, 1, 1992, pp. 32-37.

[Bass 1991] L. Bass and J. Coutaz, *Developing Software for the User Interface*, Addison Wesley Publ., 1991.

[Coutaz 1987] J. Coutaz, "PAC, an Object Oriented Model for Dialog Design", in Proc. *Interact'87*, North Holland, 1987, pp. 431-436.

[Coutaz 1991] J. Coutaz, S., Balbo, "Applications: A Dimension Space for User Interface Management Systems", in Proc. *CHI'91 Human Factors in Computing Systems*, ACM Press, 1991, pp. 27-32.

[Coutaz 1993] J. Coutaz, L., Nigay, D., Salber,"The MSM framework: A Design Space for Multi-Sensori-Motor Systems", in Proc. *East/West Human Computer Interaction*, *EWHCI'93* (Moscow, Aug. 3-7, 1993), Springer Verlag Publ., Lecture notes in Computer Science, Vol. 753, 1993, pp. 231-241.

[Demazeau 1991] Y. Demazeau and J-P Müller, "From Reactive to Intentional Agents", *Decentralized Artificial Intelligence*, vol. 2, Demazeau & Muller eds., North Holland, Amsterdam, 1991.

[Ducasse 1993] S. Ducasse, M. Fornarino, "Protocol for managing dependencies between objects by controlling generic function invocation", in *OOPSLA'93 workshop on Object-Oriented Reflection and Metalevel Architectures*, Washington, ACM, 1993.

[Duce 1991] D. Duce, M.R. Gomes, F.R.A. Hopgood, J.L. Lee (eds), *"User Interface Management and Design"*, Eurographics Seminars, Berlin: Springer-Verlag, 1991, pp. 36-49.

[Duke 1993] D. Duke, M. Harrison, "Towards a Theory of Interactors", The Amodeus Project, Esprit Basic Research 7040, Amodeus Project Document, System Modelling/WP6, 1993.

[Duke 1994] D. Duke, M. Harrison, "Folding Human Factors into Rigourous Development", in the Proc. of *Eurographics Workshop "Design, Specification, Verification of Interactive Systems"*, F. Paterno' Ed., 1994, pp. 335-352.

[Garlan 1993] D. Garlan, M. Shaw, "An Introduction to Software Architecture", *Advances in Software Engineering and Knowledge Engineering*, V. Ambriola and G. Tortora Eds., Vol. 1, World Scientific Publ., 1993, pp. 1-39.

[Goldberg 1984] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley Publ., 1984.

[Hill 1986] R. Hill, "Supporting Concurrency, Communication, and Synchronization in Human Computer Interaction- the Sassafras UIMS, ACM Transactions on Graphics, 5 (3), 1986, pp. 179-210

[Hill 1992] R. Hill, "The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications. In proceedings CHI'92, ACM:New York, 1992, pp. 335-342.

[Ilog 1989] Ilog, "Aïda, Environnement de développement d'applications", Manuel de référence Aïda, Version 1.32, Copyright Ilog, mars, 1989.

[Kazman 1994] R. Kazman, L. Bass, G. Abowd, M. Webb, "SAAM: A Method for Analyzing the Properties of Software Architectures", In Proc. of the *16th International Conference on Software Engineering*, ICSE'94, 1994.

[Linton 1986] M. Linton and C. Dunwoody, “Partitioning User Interfaces with Interactive Views”, Computer Systems Laboratory, Stanford University, Stanford, CA 94305-2192, April, 1986.

[McCall 77] J. McCall, *Factors in Software Quality*, General Electric Eds, 1977.

[Nigay 1994] L. Nigay, " Conception et modélisation logicielles des systèmes interactifs : application aux interfaces multimodales". Thèse de doctorat de l'Université Joseph Fourier, 1994.

[Nigay 1995] L. Nigay, J. Coutaz, "A Generic Platform for Addressing the Multimodal Challenge", in Proc. *CHI’95 Human Factors in Computing Systems*, ACM New York, Denver, May 1995, pp.98-105.

[OSF 1989] “OSF/Motif, Programmer's Reference Manual”, Revision 1.0; Open Software Foundation, Eleven Cambridge Center, Cambridge, MA 02142, 1989.

[Paterno' 1994] F. Paterno', A. Leonardi, S. Pangoli, "A Tool Supported Approach to the Refinement of Interactive Systems", in the Proc. of *Eurographics Workshop "Design, Specification, Verification of Interactive Systems"*, F. Paterno' Ed., 1994, pp. 85-96.

[Pfaff 1985] G.E. Pfaff et al., *User Interface Management Systems*, G.E. Pfaff ed., Eurographics Seminars, Springer Verlag, 1985.

[Salber 1995] D. Salber, "De l'interaction individuelle aux systèmes multi-utilisateurs. L'exemple de la Communication Homme-Homme-Médiatisée". Thèse de doctorat de l'Université Joseph Fourier, September, 1995.

[Scheifler 1986] R.W. Scheifler and J. Gettys, “The X Window System”, *ACM Transactions on Graphics*, 5(2), April, 1986, pp. 79-109.

[Schmucker 1986] K. Schmucker, “MacApp: An Application Framework”, *Byte*, 11(8), 1986, pp. 189-193.

[Shaw 1995] M. Shaw, D. Garlan, *Software Architecture. Perspectives on an Emerging Discipline*, Prentice Hall, 1995.

[Valdez 1989] J. Valdez, “XVT, a Virtual Toolkit,” *Byte* ,14(3), 1989.

Keywords: PAC, MVC, Arch, PAC-Amodeus, agent-based architecture, interactor, software architecture modelling, interactive systems.