The AMODEUS Project
ESPRIT Basic ResearchAction 7040

First Version of a Reference Model for the Software Design of Interactive Systems

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

11th June 1994

Amodeus Project Document: System Modelling/WP38

A Reference Model for the Software Design of Interactive Systems

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

Abstract
This document constitutes the first version of the “Amodeus system reference model”. The intention of the reference model is to serve as a sound framework for reasoning about the software design of non conventional interactive systems. The model operates at three levels of refinement. Each level covers a distinct phase of the software development process and serves different purposes. The first level addresses issues that are of interest to both software designers and HCI modellers. It provides designers and modellers with high level concepts (e.g., fusion) and properties motivated by cognitive modelling (e.g., observability). Level 2 of the reference model provides software designers with conceptual models and formalisms (e.g., York, CNUCE and PAC-Amodeus agent-based models) that support the reification process from the task analysis or from the external specifications of a system into system-oriented constructs. Level 3 aims at providing software implementers with reusable software mechanisms that follow from Level 1 properties (e.g., the fusion mechanism).

Keywords: CNUCE, PAC, MVC, Seeheim, Arch, PAC-Amodeus, MSM framework, multi-agent models, interactor, software architecture modelling, interactive systems.

Table of Contents

1. Introduction

This document constitutes the **first version** of the “Amodeus system reference model”. The intention of the Amodeus system reference model is to serve as a sound framework for reasoning about the software design of non conventional interactive systems. In particular, the goal is:

(1) to define the scope and dimensions of the problem of designing interactive systems,
(2) to understand the role of system modelling formalisms in addressing these problems,
(3) to provide a starting point for opening links with other modelling techniques.

The Amodeus system reference model is primarily intended for software designers. Its foundational concepts however can be used by cognitive modellers to characterise the usability of a particular system. Although not scheduled for completion until year 3, the framework already covers the key aspects of innovative systems such as multimodal user interfaces.

The reference model operates at three levels of refinement. Each level covers a distinct phase of the software development process and serves different purposes. The structure of the reference model and its coverage within software life cycles are discussed in Section 2. Sections 3, 4 and 5 present the key concepts of each level of the reference model. Detailed descriptions are provided in Annexes or in Amodeus Working papers.

Annex 1 is a summary of the main models used in software engineering to structure the software development process: the waterfall model, the V life cycle, and the spiral model. In the following discussion, we will make extensive reference to the V model: the V model provides a simple structuring process and makes testing activities explicit.

Annex 2 presents a detailed description of the MSM (Multi-Sensory-Motor) dimensions. This framework proposes a set of dimensions that clarify the software problems of designing innovative interactive systems.

Annex 3 is a state of the art review of software architecture modelling techniques. It will be used to demonstrate our contribution to this domain of research.

2. Structure and Coverage of the Reference Model

As shown in Figure 1, the Amodeus reference model covers three levels of interest ranging from key concepts and properties to technical implementation concerns:

- The first level is intended for both software designers and HCI designers and modellers.

- The second level is primarily intended for software designers.

- The last level, which provides technical mechanisms and solutions, is intended for implementers.

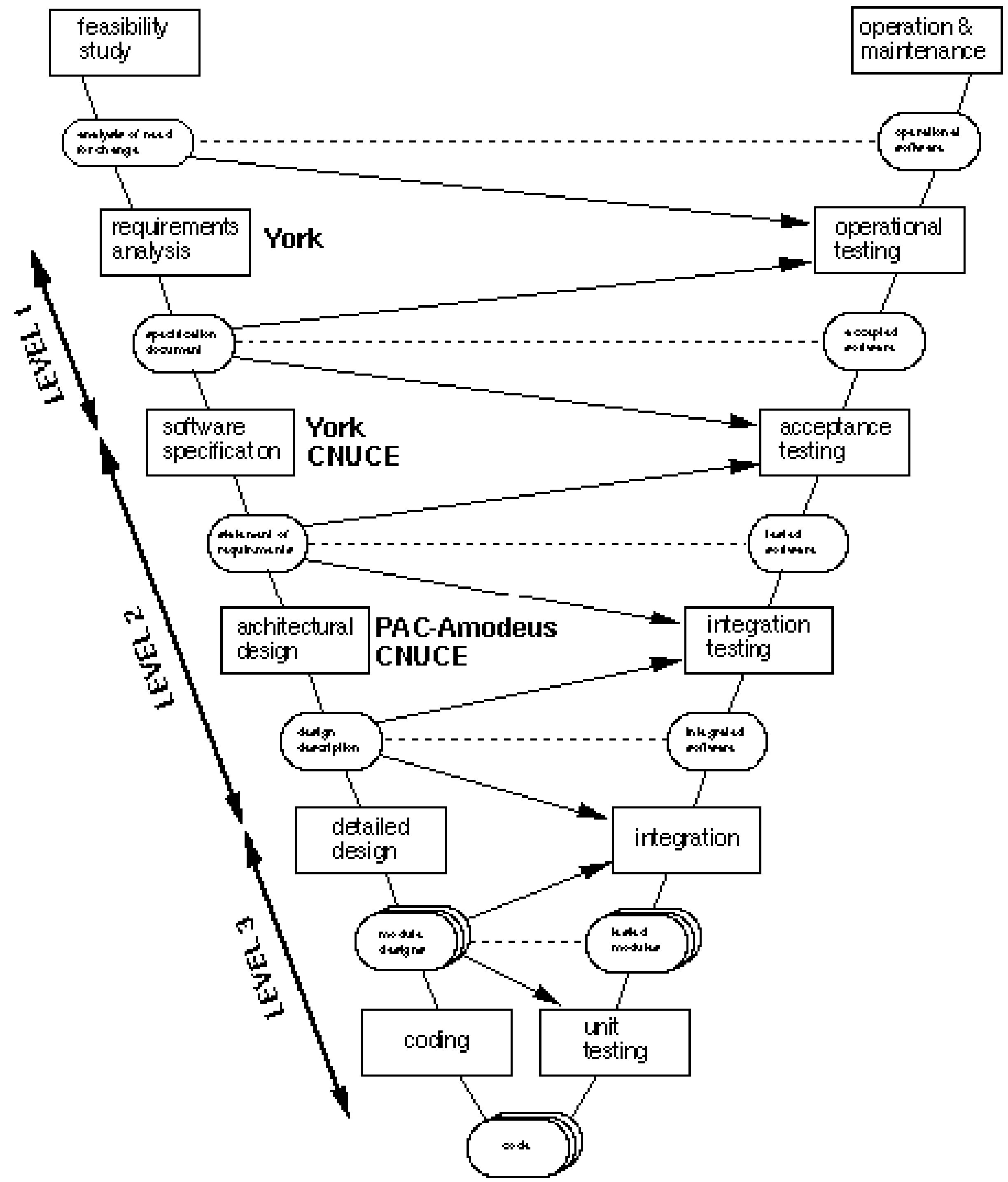


Figure 1: Coverage of the Amodeus reference model for the V life cycle development process. (See Annex 1 for the detailed description of the phases, the relationships and their product.)

2.1. Level 1 : Concepts and properties

The first level of the reference model addresses issues that are of interest to both software designers and HCI modellers. It provides designers and modellers with high level concepts such as the notions of *interaction language* and *parallelism*, as well as properties motivated by cognitive modelling such as the notions of *observability* and *conformance*. These concepts, which cover the knowledge that a system and a user must share to accomplish a task successfully, are both user-centred and relevant to software design. They are useful for characterising a particular design or for making comparisons between alternative designs. They are also useful as usability criteria.

Usability criteria are properties used to assess the usability of a particular software. They are specified in a quality plan which, in turn, is part of the deliverables of the requirements phase (see the V model in Annex 1). For each criterion the quality plan should contain an informal definition showing the relationship of the criteria with a quality factor, the metrics adopted to assess the criteria, the method and/or the tools used for the measurement, the desired value for the metrics, and the range that is expected and acceptable for positive assessment. It is interesting to observe that Software Quality Assurance adopts similar principles to those advocated in usability engineering : "evaluation is an assessment of the conformity between a system's performance and its desired performance" [Whitefield 91, p. 70].

The similarity of approaches and problems between HCI usability engineering and software quality assurance opens the way to their mutual integration. The pending problem is cross-discipline education. As long as "usability" will be loosely defined, the software community will not be able to address the appropriate quality goals for it.

Level 1 of the reference model is an attempt to refine "usability" in terms of more specific criteria such as observability, predictability and honesty. It can be used as a common reference between HCI designers and software designers in the early phases of the development process (e.g., the requirements and specification phases of the V life cycle).

2.2. Level 2 of the Reference Model : conceptual architectural models

Level 2 of the reference model provides software designers with conceptual models and formalisms that support the reification process from the task analysis or from the external specifications of a system into system-oriented constructs. If we refer to the V model, Level 2 can be used as a reference for designing the architectural and detailed specifications of the software.

Although system-oriented, constructs of Level 2, which are based on the notion of *agent* and *interactor*, find their root in Level 1 of the reference model: they are informed by, and motivated by the user-centred issues of human computer interaction. They can be exploited in distinct complementary ways according to the PAC-Amodeus, or the York and CNUCE approaches. PAC-Amodeus aims at supporting the structuring process of a software within the constraints of technical practices and environments. The York and CNUCE approaches do not consider technical details but aim at formally proving properties about the system behaviour:

- The York modelling approach supports HCI and system modellers in the process of producing a precise statement of the intended behaviour of the system. It does so in terms of properties and interactors that are formally described using a Z-based notation. Interactors are used to express the behaviour of a system from its informal, possibly incomplete and ambiguous external specifications. A York formal construct, which combines properties and interactors, allows designers to formally verify that the intended properties are effectively supported by the set of interactors. In other words, a

York model can be used conjointly with HCI designers to help understanding key issues about a particular problem and verify the soundness of an HCI design solution with regards to these issues.

- Once an external specification of the system has been produced and verified through the York model, the system designer may use PAC-Amodeus to structure the interactive system into maintainable, modular components. In particular, PAC-Amodeus shows how to integrate Level 1 foundational concepts of the reference model, such as interleaving, languages and devices, into a software global design and how to conciliate this software global design with the platform and tools that will be used for implementing the system.

- The CNUCE approach provides an architectural model for the specification of an interactive system using the LOTOS process oriented notation. A system specification is expressed as a composition derived from the task analysis, of predefined basic interaction objects (interactors) and of their controlling agents. Specific behavioural properties, embedded in a specification, can be verified by taking a model checking approach. An abstract architectural specification can then be refined into an implementation which keeps the intended properties by means of correctness preserving transformations.

Figure 1 makes explicit the relationship between the York, CNUCE, and PAC-Amodeus approaches within the V development process.

2.3. Level 3 : Software mechanisms

Level 3 of the reference model feeds into implementational issues. The intent is to provide software implementers with a generic class of mechanisms that support properties and features identified in Level 1 (e.g., fusion and fission) and that can be embedded into agent-based architectures such as PAC-Amodeus. In the long run, these mechanisms would be part of the run time systems offered by the UIMS technology. Clearly, Level 3 of the reference model fits into the coding phase of the V life cycle.

At this stage, a particular mechanism is introduced for fusing sound and other forms of input to create high level semantically meaningful constructs (e.g., commands).

In the following sections, we describe each level of the Amodeus reference model in more detail.

3. Level 1 of the Reference Model: Concepts and properties

The concepts and properties we have developed for reasoning about non conventional interactive systems address two different types of concerns: that of the software designer and that of the HCI designer. The pipe-line model shown in Figure 2 is one way of reconciling the two perspectives where the system side has been blown up for the purpose of the discussion. The human side may be expanded in a symmetric way using, for example, ICS or Norman's 7 stage model (goal, intention formation, action specification, action execution, perception, interpretation and evaluation) [Norman 86].

In the next paragraph, we will use the pipe-line model as a framework for presenting some of the concepts of the reference model. A more complete list of concepts is provided in the glossary (SM/WP27). Properties will then be discussed in 3.2. As shown in Paragraph 3.3, they can be used as criteria to assess the usability of a particular system. They can also be used to classify interactive systems (e.g., make a distinction between multimedia and multimodal user interfaces) or compare alternative design solutions. Classification of interactive systems will be addressed in paragraph 3.4. *Italic* will be used to denote concepts or properties whose definition can be found in the glossary (SM/WP27).

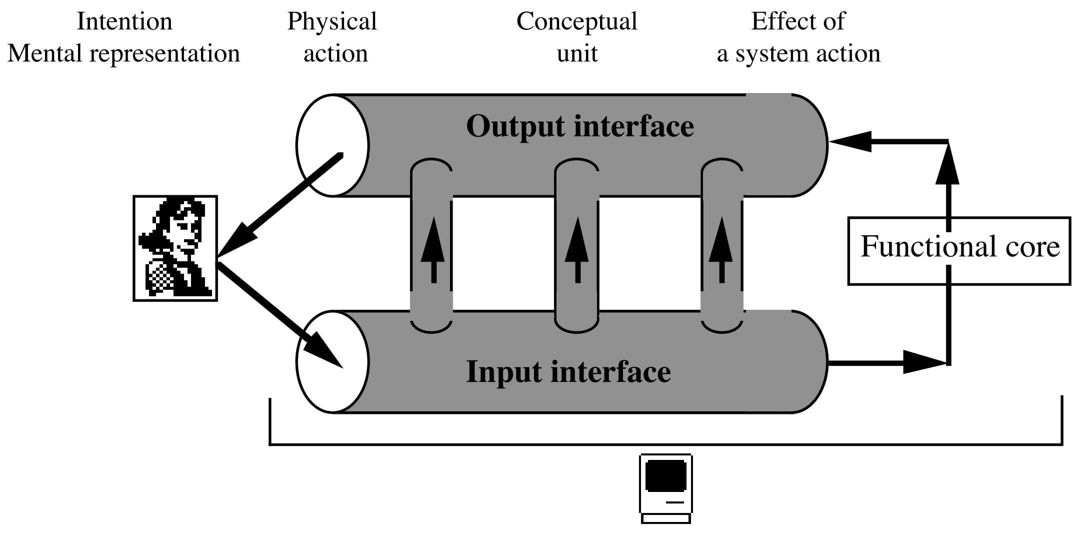


Figure 2: The Pipe-Line Model [Nigay 94].

3.1. Concepts of the reference model

As shown by the pipe-line model, the user translates an intention into *physical actions*. Physical actions that are detected by *physical input devices* are abstracted by the system into conceptual units. In turn, the interpretation of *conceptual units* in the functional core has an effect on the system state. Depending on the communicational intent of the system, the effect is expressed in terms of conceptual units which in turn are rendered as physical actions on *output devices*. The user may perceive the state change, interpret the new state and compare the situation with the current goal, etc.

This simplistic view can be enriched with extra system activities. These activities are illustrated in Figure 2 as shortcuts between the input and output interfaces. For example, an input physical action may have a direct impact on the output interface without being processed as a conceptual unit. The MSM dimensions presented in Annex 2 are other refinements of the system picture provided by the pipe-line model. These dimensions include:

- the notion of *parallelism*. In Figure 2, input and output interfaces may be simultaneously active and each interface may handle multiple flows of data in parallel;

- the notion of *context*. Context is used by the system to direct the transformation of physical actions into conceptual units then into effects, and vice-versa (a refinement of the notion of context is described in Annex 2);

- a generalisation of the notion of *level of abstraction*. Pipe-line makes explicit three levels of abstraction: physical actions, conceptual units and effects. When designing the software components of a particular system, one may need to refine the three level model of pipe-line (Level 2 of the reference model will show how);

- *fusion* and *fission*. These phenomena express the combination or decomposition of physical actions and/or of conceptual units that may happen through the input and output interfaces. (A detailed description, illustrated with examples, of fission and fusion is provided in Annex 2.)

In the description of the pipe-line model, we have used the notions of physical devices and physical actions. More precise definitions follow:

A *physical device* is an artefact or an organ required by a system or a human in order to acquire (input device) or deliver (output device) information. Examples include keyboard, loudspeaker, microphone, ears and mouth.

A *physical action* is an action performed either by the system or the user on a physical device. Examples include highlighting information (system physical actions), pushing a mouse button or uttering a sentence (user physical actions).

Physical devices and interaction languages are the resources and knowledge that the system and the user need to share to accomplish a task successfully.

An *interaction language* is a language used by the user or the system to exchange information. A language defines the set of all possible well-formed expressions, i.e., the conventional assembly of symbols, that convey meaning. The generation of a symbol or a set of symbols, results from a physical action. Examples include pseudo-natural language, direct manipulation language, Bernsen's classes of "output modalities" [Bernsen 93].

If we adopt Hemjslev's terminology [Hemjslev 47], the physical device determines the substance (i.e., the unanalysed raw material) of an expression whereas the interaction language denotes its form or structure. For example, in MATIS (see SM/WP10), it is possible to use speech, or type in a sentence in natural language, or fill a form to get information about flights time schedule:

- speech input is characterised by (device = microphone, language = the pseudo natural language NL), where NL is defined by a grammar,

- written natural language input as (device = keyboard, language = the pseudo natural language NL),

- graphic input as (device = mouse, language = direct manipulation),

- graphic output as (device = screen, language = tables). (Results for a request are presented in a tabular form.)

3.2. Properties

A *property* is a characteristic of a system that can be defined as a predicate or measure expressed in terms of the observables of the system.

One can refer to the system glossary for the definition of a number of properties that are of interest to both HCI and system modellers [SM/WP26]. We will focus on two of them: *conformance* and *observability*.

From the notions of interaction languages and devices, we derive the CARE properties (Complementarity, Assignment, Redundancy, and Equivalence). These properties formally define how languages and devices relate to each other. We will show how they affect flexibility and robustness:

- Interaction flexibility covers the multiplicity of ways that the user and the system can exchange information.

- Interaction robustness denotes the features of the interaction that support the successful achievement and assessment of goals.

Conformance and observability

Conformance
Property that the presentation of a system mirrors the underlying behaviour of the system.

Observability
Property that the presentation of a system contains sufficient information to allow the user to determine the functional state of the system.

Properties like conformance and observability, indicate that system modelling is motivated by user-centred concerns. In particular, the YORK formal approach (Level 2 of the reference model) is able to, or aims to, provide a precise unambiguous definition for these notions, and demonstrate that a particular system verifies the property. These properties can be used to assess the usability of a system.

The CARE properties

As shown in figure 3, the CARE properties involve both devices and languages. They can be either permanent or transient, and either total or partial. A relation is permanent if it holds for any state of the system ; otherwise, it is said to be transient. A relation is total if it holds for any task supported by the system. It is partial when the relation is true for a subset of the task space. In this discussion, a task is supposed to be elementary, that is, it cannot be decomposed further into sub tasks: the body of an elementary task is expressed in terms of physical actions.

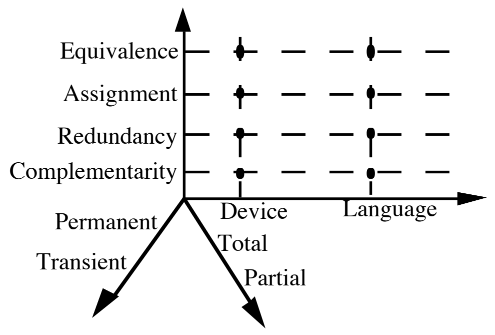


Figure 3: The dimensions of the CARE properties.

Having presented the general picture of our framework, we need now to define the relations more formally. To do so, we will consider the notions of system state and set of tasks to express the coverage of the relations over time and over the user's task space.

Language equivalence: L-Equivalence(L, s, T)
Interaction languages of a set L are equivalent over a state s and a non empty set T of tasks, if all of the tasks in T can be expressed using either one of the languages in L. Equivalence is permanent if the relation holds for any state. Equivalence is total if the relation holds for all of the tasks that can be performed with the system. In MATIS, the user can specify a request using direct manipulation or natural language. For example, the departure time of a flight may be specified by clicking "morning" on the screen or by saying "arriving in the morning". Direct manipulation and natural language are permanently and totally equivalent for tasks that are related to the specification of requests.

Language assignment: L-Assignment(l, s, T)
An interaction language l is assigned in state s to a set of tasks T, if there is no any other interaction language equivalent to l over s and T. Assignment is permanent if the relation holds for any state. Assignment is total if the relation holds for all of tasks that can be performed with the system. For example, in MATIS, window manipulation such as cut and paste tasks can be performed using direct manipulation language only. In particular, speech is not available for any task other than request specification tasks.

Language redundancy: L-Redundancy(L, s, t)
Interaction languages of a set L can be used redundantly in some state s for a task t, if these languages are equivalent over s and t, and if they can be used simultaneously to express t. In MATIS, the user can articulate the sentence "flights from Boston" while selecting "Boston" with the mouse in the menu of known cities. Here, speech and direct manipulation are used redundantly. As another example drawn from the literature [Neal 88], a wall is represented redundantly by the system via a red line (graphics interaction language) and the vocal message "mind the red wall!" (natural interaction language).

Language complementarity: L-Complementary(L, s, T)
Interaction languages of a set L are complementary over a state s and a non empty set T of tasks, if T can be partitioned such that for each partition Tp of T, there exists a language l of L assigned to Tp over s. Language complementarity is best illustrated by co-referential expressions. For example, in MATIS, natural language and direct manipulation are complementary over the request specification tasks using deictics. In particular, the user can articulate the sentence "Flights from this city to that city" while selecting "Boston" and

"Pittsburgh" in the menu of known cities. In this example, speech is assigned to denote the slots concerned in the request (i.e., the source and destination), and direct manipulation is assigned to the specification of the slot values (i.e., Boston and Pittsburgh).

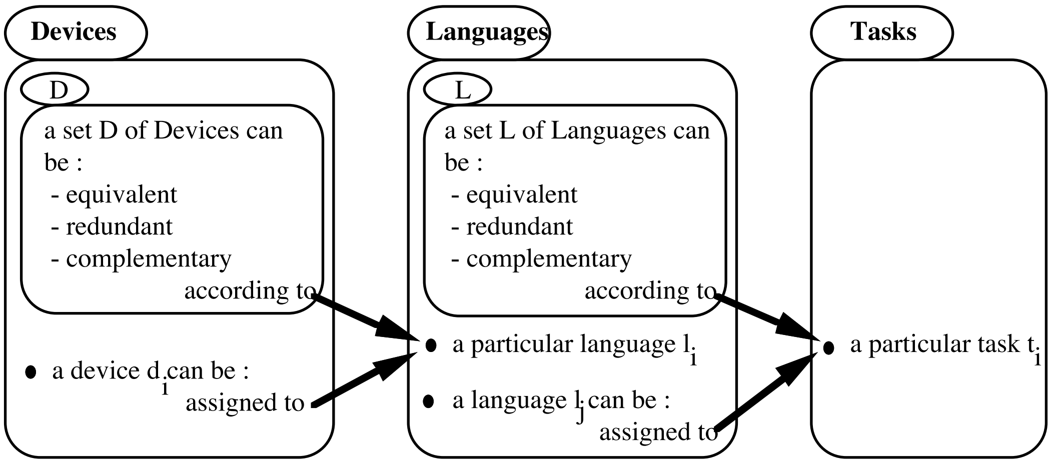


Figure 4: Overall representation of equivalence, assignment, redundancy, and complementarity between devices, languages and tasks.

Figure 4 shows the global picture of the relations between physical devices and interaction languages. Similar relations hold between devices and languages. A complete description of their formal definitions can be found in [Nigay 94]. To illustrate the approach, we will consider the equivalence of a set of devices for a particular language l:

Device equivalence: D-Equivalence(D, s, E, l)
Devices in a set D are equivalent over a state s and a non empty set E of expressions of an interaction language l, if all of the expressions of E can be elaborated using either one of the devices in D. Equivalence is permanent if the relation holds for any state. Equivalence is total if the relation holds for E equals to the set of expressions that define l. For example, in MATIS, the user who chooses to specify a request in natural language, can either type-in or utter the sentence: keyboard and microphone are permanently and totally equivalent over natural language.

Similarly, device assignment expresses the binding of a device for formulating expressions of a particular language. In MATIS, the mouse is permanently assigned to window resizing for direct manipulation language. An example of device redundancy would be the situation where the user spells a character using the microphone and, in the mean time, type in the same character.

3.3. Properties as criteria for usability assessment

To illustrate the discussion, we will use the applicability of the CARE properties as criteria for usability assessment.

From the above definitions, we conclude that equivalence provides a way to enhance flexibility: for input, the user has multiple choices among languages and/or devices and, for output, the system can choose among multiple renderings. Equivalence is also a good ingredient for robustness. For example, in a noisy environment, a MATIS user may switch to direct manipulation or key-in natural language sentences. For critical systems, equivalence of languages and/or devices may be required to overcome device breakdowns or processing limitations. At the opposite, assignment can be viewed as a restrictive feature. Redundancy provides freedom and may also be used to increase robustness. As for complementarity however, usage of multiple devices and languages may generate cognitive overload and extra articulatory synchronisation problems.

"Permanency" and "totality" of the CARE relations are also interesting usability factors. A permanent relation, which holds for any state, and a total relation which holds for all of the tasks and/or languages available, express the absence of exceptions. At the opposite, "partiality" and "transience" fall into the "inconsistency trap". For input, the user will have to remember the context (e.g., the tasks) for which the relations hold. For output, the system may appear unstable with regard to its rendering strategy.

The CARE properties can be analysed from additional perspectives. In particular, time and processing resources may provide useful insights. For example, in MATIS the keyboard and the microphone are not equivalent for specifying sentences in NL if one considers the time needed to elaborate the sentence. From the system perspective, the acquisition of a NL sentence involves more processing resources when uttered than when it is typed in.

In summary, the CARE relations provide a formal foundation for the usability assessment of interactive systems. We are aware that our current definitions may be enriched in many different ways. Additional constraints such as time and processing resources used by the user and/or the system are two relevant features. For now, we stick to the definitions provided above and discuss their implications on software design. Our technical solution for supporting the CARE properties draws upon our software architecture model: PAC-Amodeus [Nigay 91b].

3.4. Properties and concepts as classification means

A detailed description of classification schemes for innnovative interactive systems can be found in [Nigay 94]. In this paragraph, we reflect the essence of this work and show how some properties and concepts from Level 1 are useful for classifying interactive systems. In particular, we will organise interactive systems depending on the choices and combination of choices that the system supports at the physical device and interaction language levels.

The very first method, M^{2}LD (<u>M</u>ono/<u>M</u>ulti <u>L</u>anguage and <u>D</u>evice), for classifying or measuring the "interaction richness" of a system is to enumerate the maximum number of interaction resources that the system supports :

- *di,* the number of distinct input devices that the user can employ,
- *do,* the number of distinct output devices that the system uses to render conceptual units,
- *li*, the number of distinct input interaction languages supported by the system,
- *lo,* the number of distinct output interaction languages.

M^{2}LD provides an overall straightforward rough static classification. It does not however address the following questions which are time-dependent:

- In state s, the user has a well-formed intention: what are the choices in terms of input devices and input interaction languages? For example, in MATIS, for expressing a request, the user has the choice between natural language or filling a form (language equivalence) If natural language is chosen, then the keyboard or the microphone may be used (device equivalence).

- In state s, the system needs to render a conceptual unit whose state has changed: does the system perform choices in term of output languages and output devices? For example, the temperature in the functional core has changed. Has the system the choice between using an expression in natural language or using a graphical representation (language equivalence)? If natural language is chosen, will the message be displayed on the screen or uttered through the loud speaker (device equivalence)?

Thus device/language-equivalence and the opposite, device/language-assignment, are another way of measuring the "interaction richness" of a particular system over time (i.e., states) based on the availability of choices among interaction languages and devices.

The third and last method for classifying interactive systems is to consider the types of combination of devices and languages that the system permits. It implies that at least two languages (or devices) are available at some point in time.

- *exclusive usage* covers the sequential and independent use of languages (or devices); languages (or devices) are not used simultaneously (sequential use) and the expressions (or events) they convey are not combined (independent use).

- *concurrent usage* denotes the parallel but independent use of languages (or devices); languages (or devices) are used simultaneously (parallel use) but the expressions (or events) they convey are not combined (independent use).

- *alternate usage* means sequential and combined use; languages (or devices) are not used simultaneously (sequential use) but the expressions (or events) they convey are combined. Typically co-references between expressions supported by different languages (e.g., 'put that there') requires combination.

- *synergistic usage* corresponds to the parallel and combined use; languages (or devices) are used simultaneously (parallel use) and the expressions (or events) they convey are combined.

in summary, different types of availability and combination of interaction languages and devices provide good ways of measuring the interaction richness of an interactive system, to compare several systems, to classify a particular system, or to assess its usability. Each configuration has an impact on software architectures, the topic of the next paragraph.

4. Level 2 of the Reference Model: Conceptual Architecture Modelling

The definition of an appropriate software architecture is an important issue in user interface design. Without an adequate architectural framework, the construction of interactive systems is hard to achieve, the resulting software is difficult to maintain and iterative refinement is made impossible. Software tools, such as interaction tool kits[1], application skeletons, and User Interface Management Systems (UIMS), designed to support the construction of user interfaces, do not alleviate the problem.

Interaction tool kits such as Motif [OSF 89], application skeletons such as MacApp [Schmucker 86], and presentation-driven UIMS such as Interface Builder [Webster 89], do not embed the architectural principles that a software designer needs. For example, the "call-back procedure" paradigm made popular by X Window [Scheifler 86], does not make any 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. Software tools for the construction of user interfaces will not eliminate architectural issues as long as the construction of these interfaces requires programming. Clearly, a reference model for identifying and organising and reasoning about the components of interactive software is a necessity. This is the primary motivation for Level 2 of the Amodeus system reference model.

The literature shows a wide variety of architecture models revealing distinct goals, usage, or scientific beliefs. In order to organise our current knowledge and to clarify our understanding about architectural issues, we propose a dimension space that may be useful for characterising architectural models of interactive systems. This minimal 4D space is one component of Level 2 of the Amodeus Reference model. It is presented in section 4.1.

The second component of Level 2 is the models we recommend for designing the software architecture of innovative systems. An overview, state of the art, of significant software architecture models is presented in Annex 3 according to the successive refinements of the “Functional Core-User Interface” duality: First, the Seeheim model which focuses on the components embedded in the User Interface; then the Arch model and its generic companion, the Slinky metamodel, which consider software requirements and constraints such as efficiency and the existence of interaction tool kits; finally the multi-agent paradigm.

In this solution space, we recommend the agent-based approach whose benefits are discussed in Annex 3 and motivated through the MSM dimensions of Level 1. Thus, as the second component of Level 2, we propose:

(1) the CNUCE, YORK agent modelling approaches for abstract reasoning and property proving,

(2) the LGI PAC-Amodeus modelling technique when concerns about implementational issues prevail.

4.1. Characterising software architecture models

An architectural model identifies entities and describes how these units relate to each other. The nature of the entities depends on the goal or the perspective adopted by the designers of the model. In particular, a software architecture model may be conceptual or aimed at implementation:

[1]“interaction toolkit” and “user interface toolkit” are equivalent expressions.

- within a conceptual model, an entity denotes a concept, for example the notion of dialogue control or that of an interactor [Duke 92]. The model then shows how the concepts relate to each other in the design space;

- within an implementation model, an entity corresponds to a software component, for example the dialogue controller. The model then specifies the relationships between the components. Such relationships are driven by practical software engineering considerations such as communication protocols and correspondence with reusable code such as libraries (e.g., Xlib and Motif).

Orthogonal to the perspective adopted, is the granularity of the entities: "How big is a chunk?" For example, what does the notion of dialogue control cover? What is an interactor? Can these notions be refined in terms of more elementary concepts?

Another interesting dimension is how well a model supports "quality criteria". Quality criteria are useful notions for evaluating a particular model against specific requirements. They can be subdivided into a number of ways to account for distinct classes of requirements. One category includes software engineering factors such as reusability and maintainability; another class would cover the properties as described in Level 1 ; yet another one would characterise links with a theory as the theory of interactors investigated within Amodeus [Duke 93].

Whether it be conceptual or implementational, whether it be fine grained or not, whatever the quality criteria it is able to support, a model:

- may either serve as a guide during the software design process of an interactive system, or

- be used a posteriori as a reference to evaluate a particular software design or to reason about a particular solution within the design space provided by the model.

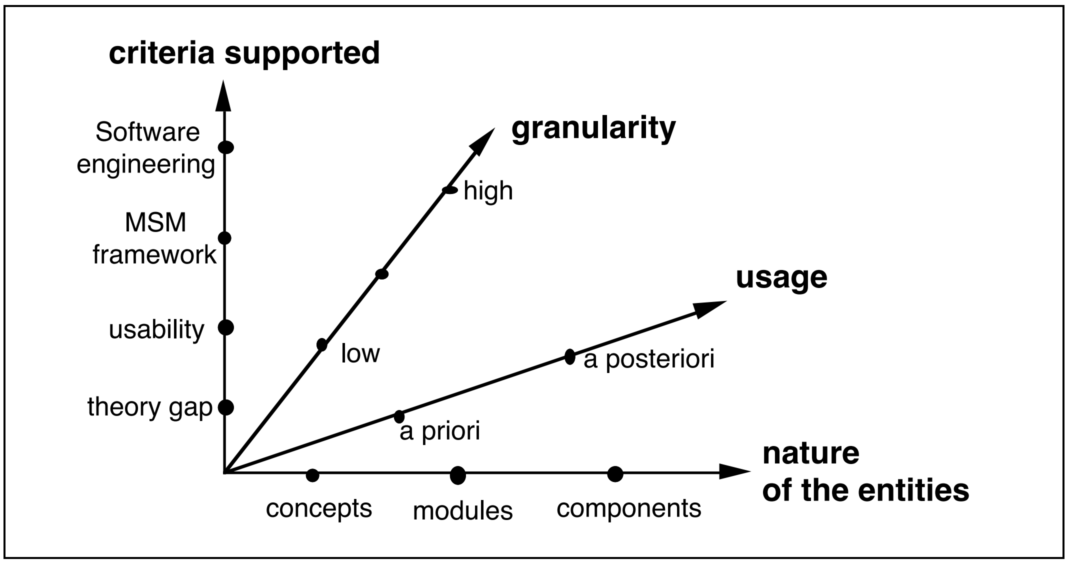


Figure 5: A minimal dimension space for characterising software architecture models (an early attempt).

Figure 5 summarises our minimal dimension space for characterising software architecture models for interactive systems. It includes, the following four dimensions:

- the nature of the entities involved: do the entities represent concepts, software components, levels of abstractions, a combination of these, etc.?
- the granularity of the entities: what is the level of refinement supported by the model? The level of refinement can be measured in terms of the number of components it advocates or in term of the potentiality for refinement (e.g., recursive decomposition);
- the usage of the model: can it be used as a driving guide during the design process of an architectural solution, or should it be used as a reference to assess the soundness of a particular architectural design?
- the quality criteria: how good is the architectural model to support quality requirements? As discussed above, quality requirements illustrate multiple perspectives from software engineering to HCI factors, and theoretical issues. In annex 3, one will find a comparative evaluation of salient architectural models according to the MSM dimensions of Level 1.

The minimal 4D space of Figure 5 is the first component of Level 2. The second deliverable of Level 2 is an architectural model applicable to the software design of innovative systems. We suggest three agent-based modelling techniques: the York, CNUCE and PAC-Amodeus approaches.

CNUCE and YORK's principles, power, limitations and interest are demonstrated in Part 1 of this report. Section 4.2 and 4.3 describe PAC-Amodeus in detail along with its prescriptive design rules. In 4.4, we close the discussion by showing how the three agent-based approaches developed in Amodeus can be used in a complementary way.

4.2. PAC-Amodeus: the principles

This section summarises the principles of PAC-Amodeus. A detailed description is available in [Nigay 94, Nigay 91a].

PAC-Amodeus blends together the Arch and PAC conceptual models. As discussed in Annex 3 (section A3.3), the Arch model has the nice feature of considering engineering reality such as trade-offs between software criteria and the constraints imposed by implementation tools (e.g., Motif) or the pre-existence of a *functional core*. The five component structure of Arch includes two component adapters which allow the software designer to insulate the key component of the user interface (i.e., the Dialogue Controller) from the variations of the functional core and implementation tools.

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 the dialogue controller, 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 6 shows the resulting structure.

As shown in Figure 6, PAC-Amodeus adopts the same overall structure than Arch: the Functional Core implements task domain concepts, the Interface with the Functional Core is used to insulate the dialogue controller from the functional core, the Dialogue Controller is in charge of task sequencing, the Presentation techniques Component insulates the Dialogue Controller from the implementation tools and the Low Level Interaction component denotes services at the toolbox level.

PAC-Amodeus goes one step further than Arch in two ways:

(1) PAC-Amodeus makes explicit the boundaries of the Presentation Techniques and the Low Level Interaction components using the notions of device and language as defined in Level 1: the Low Level Interaction Component is necessarily device dependent and language dependent; the Presentation Techniques component is device independent but language dependent; the other components of the interactive system, including the Dialogue Controller, are both device and language independent.

(2) PAC-Amodeus decomposes the Dialogue Controller into a set of co-operative PAC agents. This set is derived for a particular system using the heuristic rules presented in 4.3. In a nutshell, a cluster of agents is used to support a user's task. Interleaved and parallel tasks derived by a task analysis, are modelled as clusters of co-operating agents. Clusters of hierarchical agents are convenient ways of modelling the *abstraction/rendering functions* mentioned in Level 1 (see glossary or the MSM dimensions in Annex 2).

By doing so, we obtain an engineered agent-based model which supports Level 1 properties inherited from the agent paradigm.

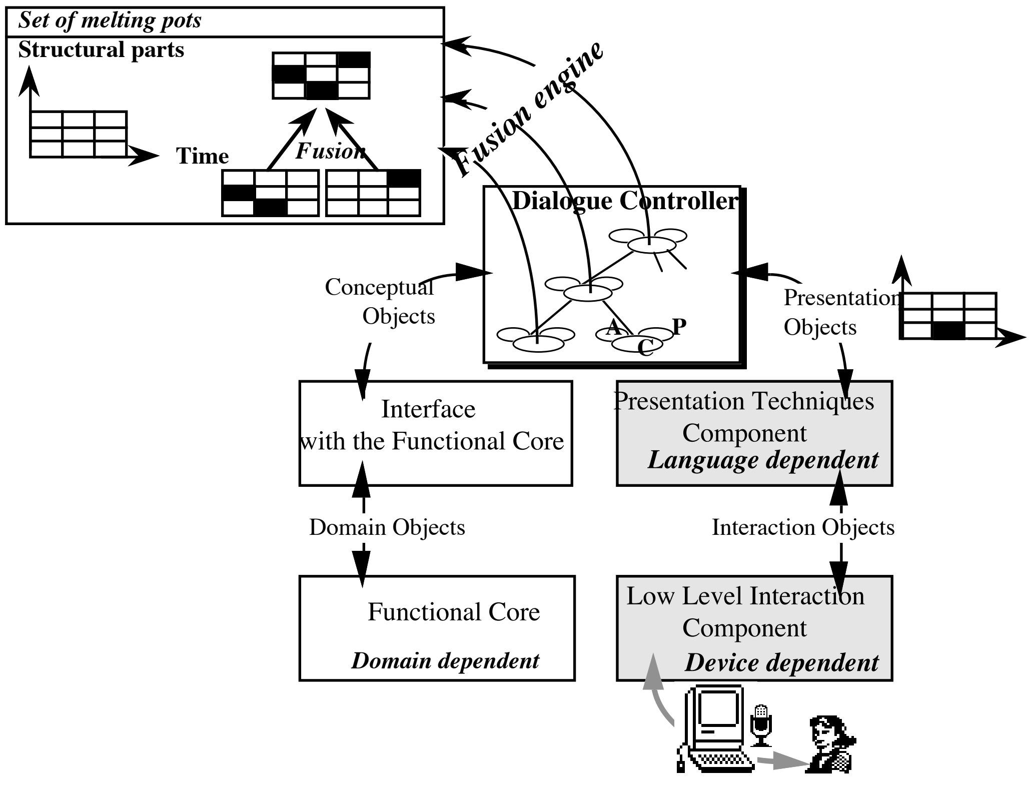


Figure 6: The PAC-Amodeus software components and their relationship with Level 1 concepts such as data fusion, interaction languages and devices.

Orthogonal to role of an agent as an abstraction/rendering process, an agent has perspectives: Presentation, Abstraction, Control. These facets are used to express different but complementary and strongly coupled computational perspectives.

- The Presentation facet of an agent implements the perceivable behaviour of the agent. As shown in Figure 7, it is related to some presentation object of the Presentation Techniques component.

- The Abstraction implements the competence of the agent (i.e., its expertise). It is the functional core of the agent. It maintains the abstract state of the agent. It may be related to some domain object(s) maintained in the Interface with the Functional Core. The abstraction facet of an agent provides a good mechanism for performing domain-knowledge delegation (i.e., deporting domain dependent information in the user interface portion of the system). This delegation is useful to improve performance (e.g., respect response time stability) as well as semantic feedback.

- The Control part of an agent is in charge of two functions: linkage of the Abstraction part of the agent to its Presentation portion and maintenance of the relationships of the agent with other agents. The linkage serves two purposes: 1) formalism transformations between the Abstraction and the Presentation portions of the agent, and 2) data mapping between the abstract facet and the presentation facet. Relationships between agents may be static or dynamic. Dynamic relationships are required when agents are dynamically created/deleted. Relationship maintenance by the control part of an agent covers the communication and the synchronisation mechanism between this agent and its co-operating partners.

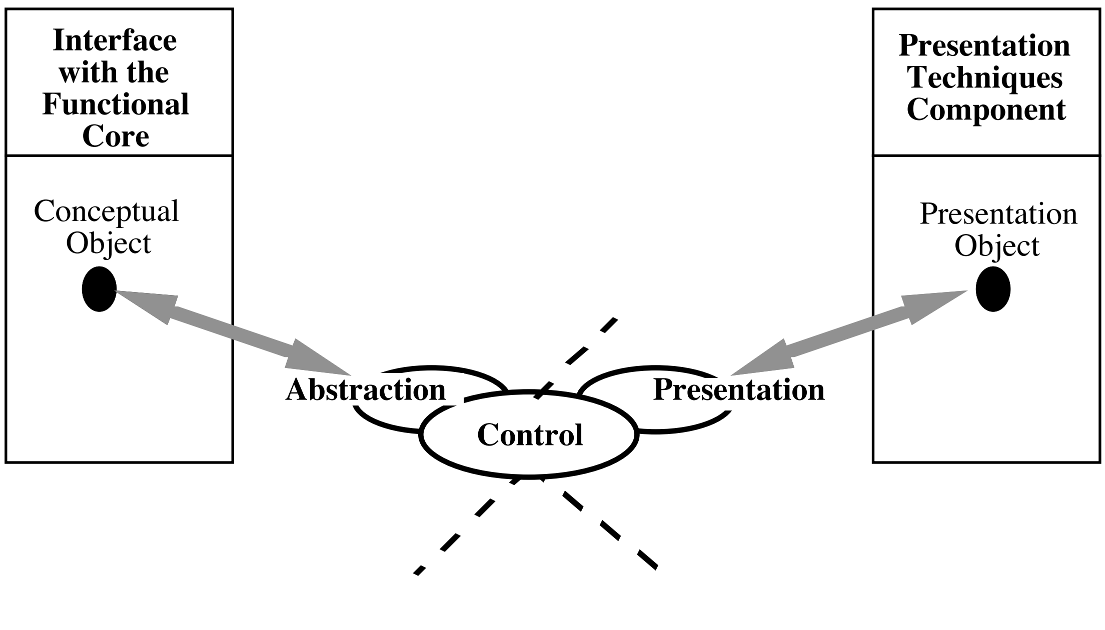

Interface with the Functional Core
Conceptual Object
Presentation Techniques Component
Presentation Object
Abstraction
Control
Presentation

Figure 7: 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 [Nigay 91b]

Next paragraph presents heuristic rules that a software designer can use when devising the general software architecture for a particular system. These rules are part of the Transfer and Assay effort of PAC-Amodeus within RP4. They explain how to use the model.

4.3. Heuristic Rules for multi-agent models

The following rules have been devised in the context of the PAC-Amodeus model and have been implemented in the form of an expert system, PAC-Expert [Nigay 91a]. PAC-Expert is able to automatically identify the agents as well as their role from the description of the user interface of the system. The rules are organised along three dimensions: window existence, window content, window links.

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.

<u>Rule 1</u>: 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 tool kit 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.

<u>Rule 2</u>: 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.

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.

<u>Rule 3</u>: 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 Interaction Tool kit 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").

<u>Rule 4</u>: 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.

<u>Rule 6</u>: 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.

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

Rule 7: 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 8: 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. 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 subagents, is called a "cement agent".

Rule 9: 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.

4.4. Level 2 Summary

In summary, Level 2 of the Amodeus reference model provides software designers with conceptual architectural tools for reasoning about and designing software architectures. The YORK and CNUCE approaches are aimed at formal reasoning while the PAC-Amodeus approach is more concerned with engineering issues. Figure 8 shows how the notion of agent is derived within Amodeus, and Figure 8-bis makes explicit the representations of an interactive system using the three modelling techniques provided by the system reference model.

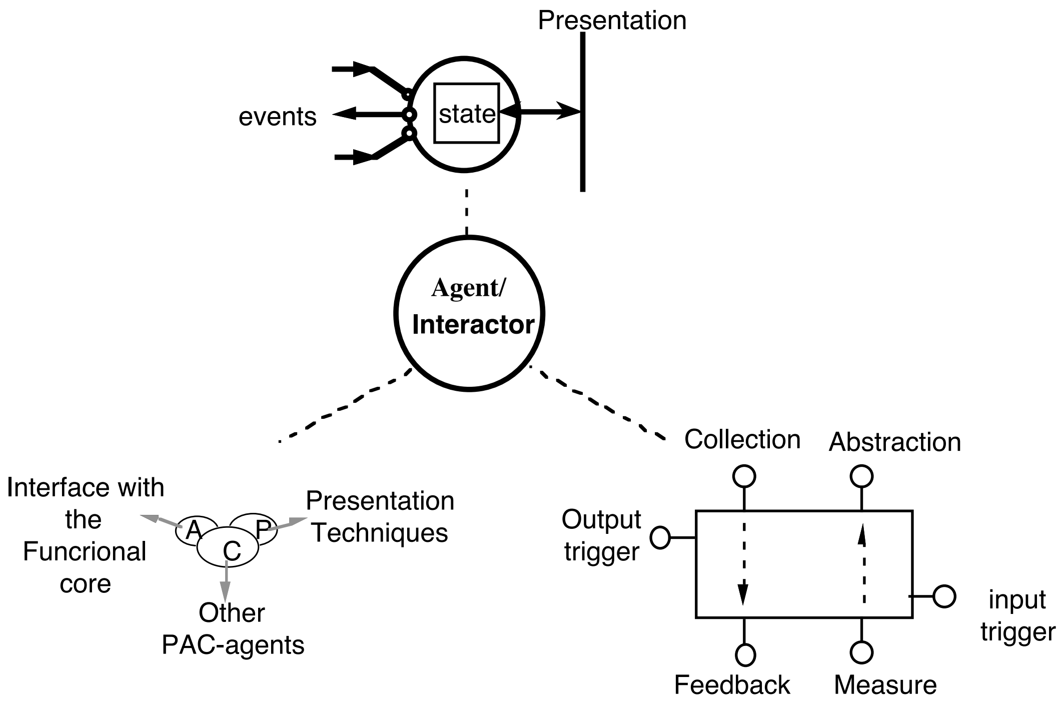
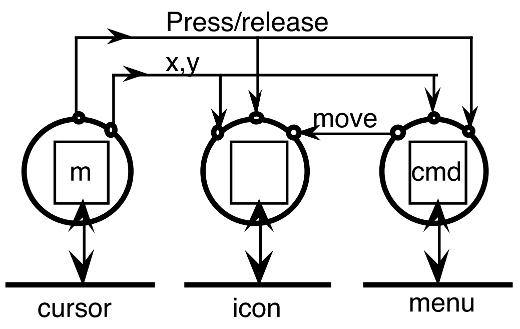


Figure 8: The Amodeus agent models.

In all three cases, an agent is an abstraction of an entity within an interactive system able to perform input and output operations, mediating between a user and a functional core: it receives information from either sides, processes that information and delivers it to another agent of the system for further processing.

A CNUCE interactor uses gates to communicate events to other agents of the system but has no explicit representation of state: the state of the system is "carried" in the events. A York interactor exchange information through events and has an internal state which is rendered by a Presentation. The state of the system is the state of the collection of agents that models the system. A PAC agent makes explicit three conceptual perspectives on an interactor that are useful to consider at multiple levels of abstraction but leaves open the specifications of state and events. This gap can be bridged using a York or a CNUCE approach. In SM/WP27 we describe heuristics rules that express the mapping between PAC architecture and the CNUCE modelling technique. Figure 8-ter summarises these rules.



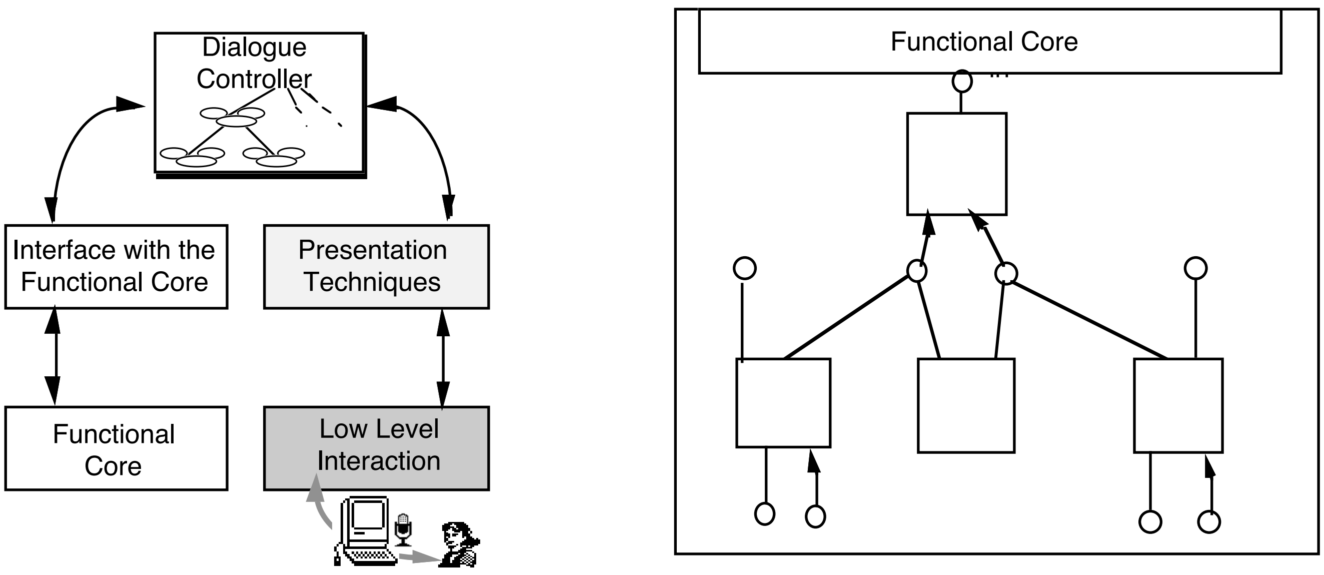
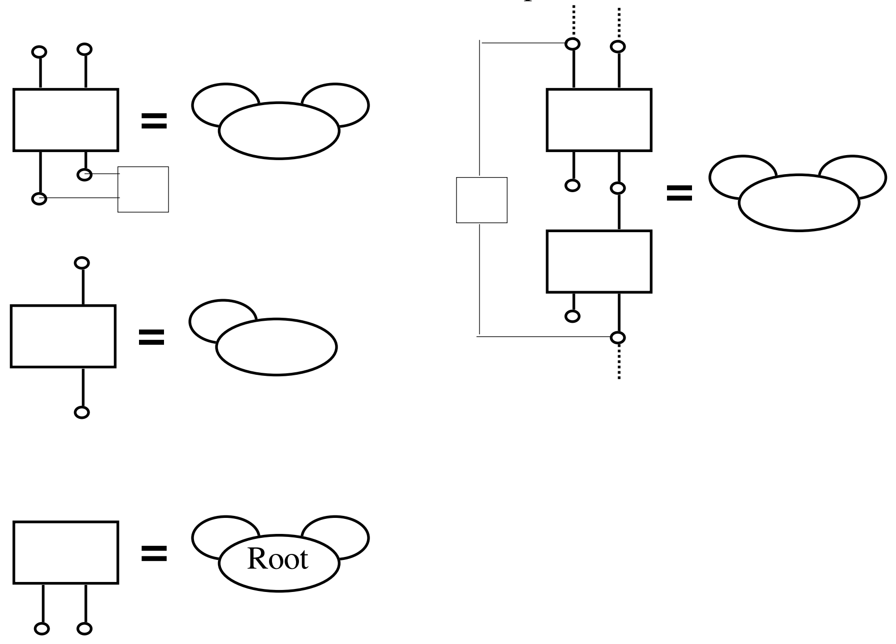


Figure 8-bis: The Amodeus agent-based modelling techniques.

Rule 1:

One interactor or a linear set of interactors with optional controller = one PAC agent



Rule 2:
Two interactors linked by a controller = Two PAC agents, sons of one PAC agent. A CNUCE controller linking two otherwise independent interactors is modelled in PAC as an agent whose competence is to control the two agents.

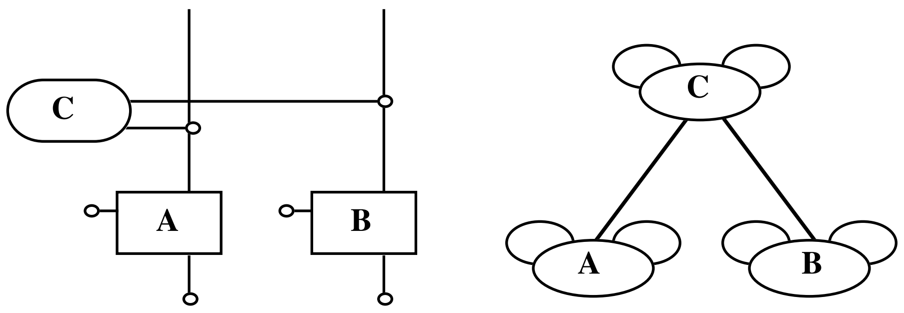


Figure 8-ter: Mapping rules between PAC and CNUCE interactors.

5. Level 3 of the Reference Model: Mechanisms

Level 3 of the Amodeus system reference model aims at providing software implementers with reusable software mechanisms that follow from Level 1 properties. At present, Level 3 offers only one general such mechanism: that for fusion.

Figure 6 in Section 4 shows how a general service such as the fusion mechanism can be integrated within a PAC-Amodeus architecture. Every PAC agent of the Dialogue Controller has access to the mechanism through its Control facet. This shared service can be viewed either as a reusable technical solution (i.e., a skeleton) or as a third dimension of the architectural model.

5.1. Principles of the Fusion Mechanism

The fusion engine uses a uniform representation model to perform fusion: the melting pot. As shown in Figures 9, a melting pot is a 2D structure. On the vertical axis, the "structural parts" model the structure of the task objects that the Dialogue Controller is able to handle. Events generated by user's actions are abstracted and mapped onto these structural parts. The Low Level Interaction and Presentation Techniques components are in charge of this process. Events are time-stamped: a mapped event defines a new column on the horizontal temporal axis. The structural decomposition of a melting pot is described in a declarative way outside the fusion engine. As a result, the fusion mechanism is domain independent: structures that rely on the domain are not “code-wired”. They are used as parameters for the fusion engine.

Using MATIS as an illustration, Figure 9 shows how redundancy and complementarity are handled by the fusion mechanism. In the redundancy example, the user has uttered the sentence “Flights from Boston” while selecting “Boston” with the mouse in the menu of known cities. The speech act is translated into the bottom left melting pot: at time t_i, the slot

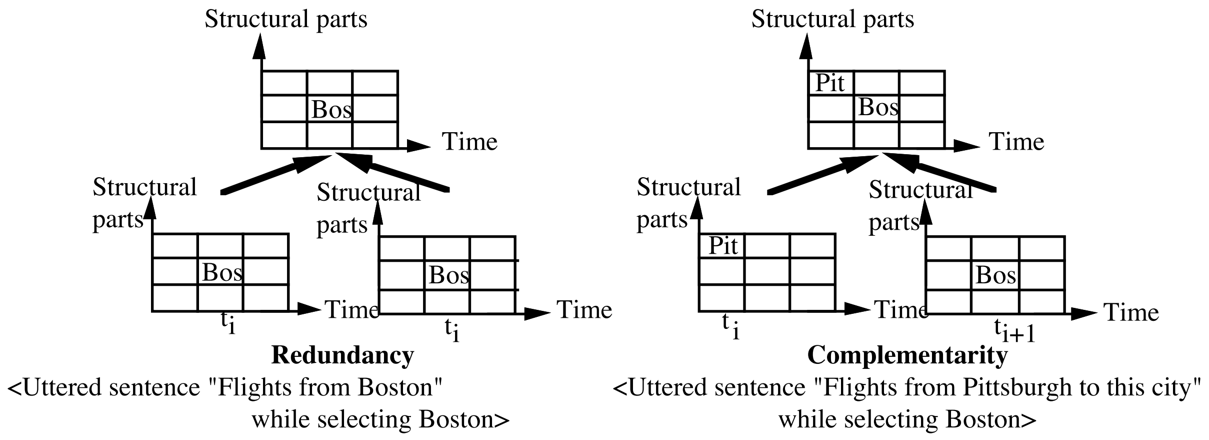

Structural parts
Pit
Bos
Time
Structural parts
Structural parts
Bos
Time
t_i
Pit
t_i
Bos
t_{i+1}
Redundancy
<Uttered sentence "Flights from Boston" while selecting Boston>
Complementarity
<Uttered sentence "Flights from Pittsburgh to this city" while selecting Boston>

Figure 9: The effect of redundancy and complementarity on the fusion mechanism.

"from" is filled in with the value "Boston". The melting pot next on the right results from the mouse selection. The fusion engine combines these two melting pots into a new one (top left). A similar reasoning applies to complementarity.

The fusion algorithm adopts an "eager" strategy: it keeps waiting for further information and always makes attempts to combine input data. This approach has the advantage of providing the user with immediate feedback before the functional core is accessed. The drawback is the possible occurrence of incorrect fusions which must be undone. Incorrect fusion occurs due to the different time scales required to process data specified through different languages and devices. As a result, the sequence of melting pots is not necessarily identical to the user's actions sequence. For example, in MATIS melting pots that correspond to direct manipulation expressions are built faster than those from voiced utterances. We will show how our engine deals with undesirable fusions.

The criteria for triggering fusion are threefold: the logical structure of commands, time, and context. When triggered, the engine applies three types of fusions in the following order: micro fusion, macro fusion, and contextual fusion.

- *Micro fusion* is performed when input data is structurally complementary and very close over time: i.e., micro fusion combines inputs if they have been produced in parallel or in pseudo-parallelism. (Intersection of time intervals.)

- *Macro fusion* is performed according to the same criteria as micro fusion but combines data that belong to a given temporal window. (Temporal proximity.) Macro fusion implies proximity of time intervals as opposed to micro fusion which implies the intersection of time intervals.

- *Contextual fusion* is performed according to the structure of the data to be combined and the current context. For example in MATIS, the context corresponds to the current request. Contextual fusion combines new input data with the current request if their respective structures are compatible.

Having presented the driving principles of the fusion mechanism, we now focus on the technical details. Our fusion algorithm has been implemented in C and embedded in a PAC-Amodeus architecture for MATIS. We first introduce the metrics associated with each melting pot. We then explain how micro fusion is implemented and show how the fusion mechanism supports redundancy and complementarity. Finally, we present the management of the set of melting pots and the exchanges of melting pots within the hierarchy of PAC agents.

5.2. Metrics for a melting pot

Figure 10 introduces the metrics used to describe a melting pot m_i:
$m_i = (p_1, p_2, \dots, p_j, \dots, p_n)$: m_i is comprised of n structures p1, p2, ...pn.
$info_{ij}$: piece of information stored in the structural part p_j of m_i.
$Tinfo_{ij}$: time-stamp of $info_{ij}$.
$Tmax_i$: time-stamp of the most recent piece of information stored in m_i.
$Tmin_i$: time-stamp of the oldest piece of information stored in m_i.
$Temp\_win_i$: duration of the temporal window for m_i.
Δt: Remaining life span for m_i.

A melting pot of conceptual information p_1, The component of as the destination request is an conceptual unit. information is system computes (Tmax and Tmin) from the time conceptual units :
So, for $m_i = (p_1,$
$Tmax_i$ =
$Tmin_i$ =

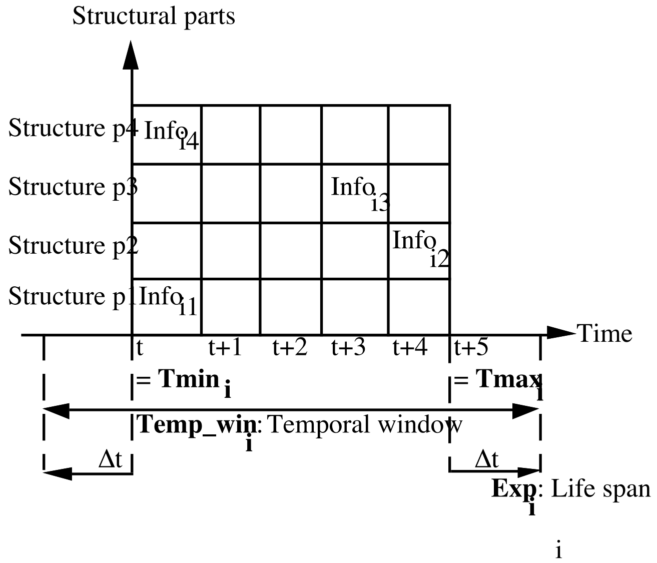


Figure 10: Metrics used to define a melting pot mi.

encapsulates a set units or pieces of $p_2, \dots, p_j, \dots, p_n$. a command such slot in a MATIS example of Each piece of time-stamped. The the boundaries of a melting pot stamps of its

$p_2, \dots, p_j, \dots, p_n)$,
$Max(Tinfo_{ij})$ and
$Min(Tinfo_{ij})$

The temporal window of a melting pot defines the temporal proximity (+/- Δt) of two adjacent melting pots: When Δt reaches a threshold (i.e., a pre-specified critical value), the engine undertakes a macro fusion:
So, for $m_i = (p_1, p_2, \dots, p_j, \dots, p_n)$, $Temp\_win_i = [Tmin_i - \Delta t, Tmax_i + \Delta t]$

The last metrics used to manage a melting pot is the notion of life span, Exp_i:
$Exp_i = Tmax_i + \Delta t = Max(Tinfo_{ij}) + \Delta t$.
This notion is useful to remove a melting pot from the set of candidates for fusion.

5.3. The Driving Rules

The fusion mechanism is driven by a set of rules. Rule 1 deals with micro fusion. Because the strategy is "eager", micro fusion is first attempted and triggered on the arrival of a new melting pot from the Presentation Technique Component. Since it models a user's action at instant t_j, this melting pot is composed of one column only. Rule 1 makes it explicit the occurrence of micro fusion: if the content of the new melting pot is complementary with a column of an existing melting pot and if the time-stamps of this column is close enough to t_j (i.e., within Δmicrot), then micro fusion is performed.

Rule 1
Micro fusion

Given:

• $col_{it} = (p_1, p_2, \dots, p_j, \dots, p_n)$:
one column at time t of an existing melting pot m_i.
• $col_{i't'} = (p'1, p'2, \dots, p'j, \dots, p'n)$
a one column melting pot $m_{i'}$ produced at time t'
• $i \neq i'$

col_{it} and $col_{i't'}$ are combined if:

• they are complementary:
Complementary (col_{it} , $col_{i't'}$) is satisfied if:
$\forall k \in [1..n] : \exists\, info_{ik} \wedge \neg (\exists\, info_{i'k})$

• their time stamps are temporaly close in the following way:
Close (col_{it}, $col_{i't'}$) is satisfied if:
$t' \in [t-\Delta microt, t+\Delta microt]$

Micro fusion may involve undoing a previous fusion. We illustrate this situation in Figure 11.

In this example, a MATIS user utters the sentence "Flights from Pittsburgh to this city" while selecting "Boston" with the mouse at time t+8. Because speech processing takes longer, the fusion mechanism will first receive the selection melting pot. If the conditions for contextual fusion are satisfied (described further), the content of this melting pot is combined with an existing melting pot m_i. The result of this process is represented in Figure 11 as Info-i1 in column t+8. The uttered sentence is then received at time t+9 within a new melting pot $m_{i'}$ as Info'-i'2 . Because Info-i1 and Info'-i'2 are very close in time, they are combined and the engine undoes the earlier fusion performed on Info-i2 and Info-i1. This gives birth to two melting pots.

Before applying the rule on Micro fusion, the mechanism checks for redundancy using Rule 2. Redundancy is defined as two close columns that contains the same information. Figure 12 gives an example of redundancy.

Rule 2
Redundancy

Given:

• $col_{it} = (p_1, p_2, \dots, p_j, \dots, p_n)$:
one column at time t of an existing melting m_i
• $col_{i't'} = (p'1, p'2, \dots, p'j, \dots, p'n)$

Figure 11: A case of micro fusion where an already performed fusion must be undone.

one column at time t' of a new melting pot $m_{i'}$

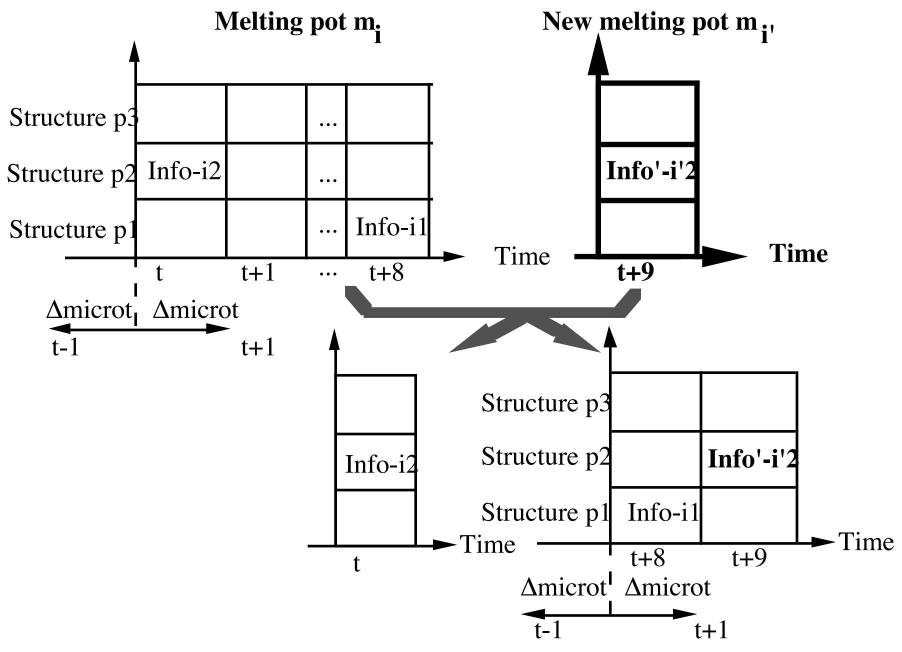
• $i \neq i'$

col_{it} and $col_{i't'}$ are redundant if:

• they contain the same information in the same slots:
Redundant (col_{it}, $col_{i't'}$) is satisfied if:
$\forall k \in [1..n]$:
$\exists\ info_{ik}\ \Lambda\ \exists\ info_{i'k}\ \Lambda\ info_{ik} = info_{i'k}$
Λ
$\forall k' \in [1..n]$: $\neg (\exists\ info_{ik'})\ \Lambda\ \neg (\exists\ info_{i'k'})$

• their time stamps are temporaly close:
Close (col_{it}, $col_{i't'}$) is satisfied if:
$t' \in [t-\Delta microt, t+\Delta microt]$

Macro fusion is driven by rules similar to those used for micro fusion where Δmicrot is replaced by temporal windows. Whereas time has a primary role in micro- and macro-fusions, it is not involved in contextual fusion. As described above, contextual fusion is the last step in the fusion process. The driving element for contextual fusion is the notion of context. In MATIS, contexts are in a one-to-one correspondence with requests. There is one context per request under specification and the current request denotes the current context. (The user may elaborate multiple requests in an interleaved way.)



When a melting pot is complete (all of its conceptual units have been received), and its life span expectancy Exp_i expires, it is removed from the set of candidates for fusion. Rule 3 expresses these conditions formally. Exp_i is used for making sure that incorrect fusions have not been performed: when a melting pot is complete, the engine keeps it for a while in the pool of candidates in case the next new melting pots trigger "undo" fusions.

Rule 3

Conditions to remove a melting pot from the list of candidates for fusion:

Melting pot $m_i = (p_1, p_2, ..., p_j, ..., p_n)$ is removed if:

- m_i is complete: $\forall p_j \in m_i, \ \exists \ info_{ij}$
- and its span life is over: current date = Exp_i

A melting pot removed from the fusion pool is sent to a PAC agent for further processing. Next paragraph describes how melting pots are linked to PAC agents.

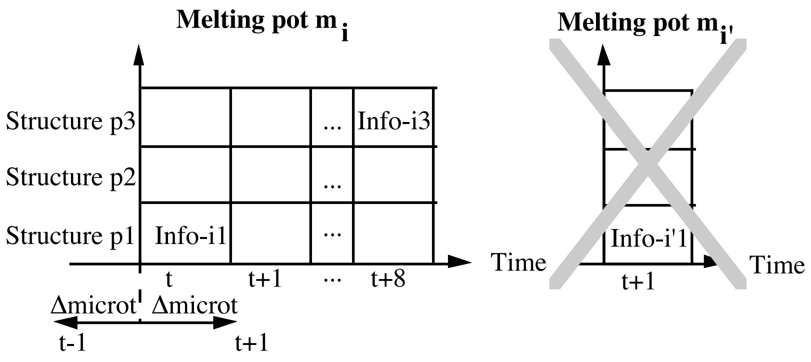


Figure 12: An example of redundancy.

5.4. The fusion mechanism and the hierarchy of agents

The PAC agents of the Dialogue Controller are in charge of task sequencing as well as processing the content of the melting pots. This activity is part of the abstraction and rendering processes as described above in Section 4. Abstracting involves multiple processing activities including the invocation of the fusion engine. When calling the engine, a PAC agent provides a melting pot as an input and output parameter: moreover the melting pot is returned to the calling agent when its life expectancy has expired. Depending on the current candidates in the fusion pool, the content of the melting pot may or may not have been modified by the fusion engine.

Data fusion is one aspect of abstraction. The enrichment of information is also performed by exchanging melting pots across the hierarchy of agents. There is one such hierarchy per task. The set of melting pots are partitioned according to the set of tasks. As a result, an agent hierarchy handles the melting pots that are related to the task it models. The benefit of this partitioning is that the fusion engine will not try to combine melting pots that belong to different task partitions. For example in MATIS, if the user utters "Flights from Pittsburgh to this city" while selecting a result window, the two melting pots that model the users physical actions will not belong to the same set. As a result, the fusion mechanism will not try to combine them by micro fusion.

6. Conclusion

In its current stage, the contribution of the system Amodeus reference model is threefold :

1) Level 1 of the reference model defines a set of key concepts and vocabulary to reason about innovative interactive systems. These concepts take the form of properties and classification schemas that are formally expressed in a non ambiguous way. They are intended as a common set of knowledge that can be shared between HCI and software modellers and designers.

2) Level 2 of the reference model provides software designers with modelling techniques that instantiate the key concepts of Level 1 in three complementary ways: the York, the LGI and the CNUCE approaches. All three use the notions of agent and interactor as a common ground but each one derives the notion of agent to serve a specific purpose. At York, the purpose of the theory of interactor is to formally express behavioural properties of a system and prove that these properties are satisfied by a particular configuration of interactors. At LGI, the purpose of the PAC-Amodeus architecture model is to provide software designers with a framework and heuristic rules to identify the software agents that are necessary to satisfy a particular system specification. At CNUCE, the purpose is to prove behavioural properties of a particular configuration of agents such as the one identified through the PAC exercise. Although they may overlap on some aspects, the York, LGI and CNUCE modelling techniques may be used sequentially in a top-down way by system modellers and designers.

3) Level 3 of the Amodeus reference model is concerned with mechanisms. The goal is to provide implementers with reusable algorithms and techniques that are relevant to the implementation of innovative interactive systems.

This is the Amodeus reference model in its first stage of definition. It currently reflects the work conducted collaboratively within the Amodeus consortium. As planned in the technical annex, it will be refined in year 3 of the project. In particular, the theory of interactor needs to be consolidated by embedding it into a formal theory which admits reasoning about properties of interactive systems, such as those that appear in the System Modelling Glossary. Some further work is also required to refine the theory of presentations. The PAC-Amodeus architecture and its reusable mechanisms need to be extended to multiple output modalities and consider its applicability within CSCW.

7. References

[Arch 92]
Arch, “A Metamodel for the Runtime Architecture of An Interactive System”, The UIMS Developers Workshop, SIGCHI Bulletin, 24(1), ACM, January, 1992.

[Bass 91]
L. Bass and J. Coutaz, Developing Software for the User Interface, Addison Wesley Publ., 1991.

[Bernsen 93]
N.O. Bersen, "Taxonomy of HCI Systems: State of the Art", ESPRIT BR GRACE, Deliverable 2.1, 1993.

[Bier 92]
E. Bier, S. Freeman, K. Pier, "MMM: The Multi-Device Multi-User Multi-Editor. In proceedings CHI'92, ACM:New York, 1992, pp. 645-646.

[Boehm 81]
B.W. Boehm, “Sotfware Engineering Economics”, Prentice-Hall, 1981

[Boehm 88]
B.W. Boehm, “A spiral Model of Software Development and Enhancement”, IEEE Computer, May 1988

[Bourguet 92]
M.L. Bourguet: Conception et réalisation d’une interface de dialogue personne-machine multimodales; Thèse de docteur de l’INPG, mars 1992.

[Coutaz 87]
J. Coutaz, “PAC, an Implementation Model for Dialogue Design”, Proceedings of Interact'87, Stuttgart, September, 1987, pp. 431-436.

[Coutaz 91]
J. Coutaz and S. Balbo, “Applications: A Dimension Space for UIMS’s”, Proceedings of the Computer Human Interaction Conference, ACM, May 1991, pp. 27-32.

[Coutaz 93a]
J. Coutaz, L. Nigay, D. Salber, The MSM framework: A Design Space for Multi-Sensori-Motor Systems. In Human Computer Interaction, 3rd International Conference EWHCI’93, East/West Human Computer Interaction, Moscow. L. Bass, J. Gornostaaev, C. Unger Eds. Springer Verlag Publ., Lecture notes in Computer Science, Vol. 753, 1993, pp.231-241.

[Coutaz 93b]
J. Coutaz, Architectural Design for User Interfaces; The Encyclopedia of Software Engineering; Wiley & sons Ed., 1993, pp. 38-49.

[Coutaz 94]
J. Coutaz : Evaluation Techniques : exploring the Intersection between HCI and Software Engineering, Workshop on “Software Engineering and Human Computer-Interaction”, International Conference on Software Engineering, ICSE’94, Sorrento, May, 1994.

[Demazeau 91]
Y. Demazeau and J-P Müller, “From Reactive to Intentional Agents”, Decentralised Artificial Intelligence, vol. 2, Demazeau & Muller eds., North Holland, Amsterdam, 1991.

[Dix 91]
A. Dix, "Formal Methods for Interactive Systems", Academic Press Publ., 1991.

[Duce 91]
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 92]
D. Duke, M. Harrison, “Abstract Models for Interaction Objects”, The Amodeus Project, Esprit Basic Research 7040, Amodeus Project Document, System Modelling/WP1, 1992.

[Duke 93]
D. Duke, M. Harrison, “Towards a Theory of Interactors”, The Amodeus Project, Esprit Basic Research 7040, Amodeus Project Document, System Modelling/WP6, 1993.

[Faconti 93]
G. Faconti, “Towards the Concept of Interactor”, The Amodeus Project, Esprit Basic Research 7040, Amodeus Project Document, System Modelling/WP, 1993.

[Foley 84]
J.D. Foley, A. Van Dam, The Fundamentals of Computer Graphics, Addison Wesley, 1984.

[Goldberg 84]
A. Goldberg, Smalltalk-80: The Interactive Programming Environment, Addison-Wesley Publ., 1984.

[Green 85]
M.W. Green, The Design of Graphical Interfaces, PhD Thesis, Tech. Report CSRI-170, Computer Systems Research Institute, University of Toronto, Canada, M5S 1A1, April, 1985.

[Harrison 90]
M. Harrison and Thimbleby, Formal Methods in Human Computer Interaction, Cambridge University Press, 1990.

[Hayes-Roth 79]
B. Hayes-Roth and F. Hayes-Roth, “A Cognitive Model for Planning”, Cognitive Science, 3, 1979, pp. 275-310.

[Hemjslev 46]
L. Hemjslev, "Structural Analysis of Language", Studia phonetica, Vol. 1, 1946, pp. 69-78.

[Hill 86]
R. Hill, "Supporting Concurrency, Communication, and Synchronization in Human Computer Interaction- the sassafras UIMS, ACM trenasactions on Graphics, 5 (3), 1986, pp. 179-210
[Hill 92]
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 89]
Ilog, “Aïda, Environnement de développement d'applications”, Manuel de référence Aïda, Version 1.32, Copyright Ilog, mars, 1989.
[Linton 86]
M. Linton and C. Dunwoody, “Partitioning User Interfaces with Interactive Views”, Computer Systems Laboratory, Stanford University, Stanford, CA 94305-2192, communication privée, April, 1986.
[McCall 77]
J. McCall, Factors in Software Quality, General Electric Eds, 1977.
[Moran 81]
T. Moran, “The Command Language Grammar: a representation for the user interface of interactive computer systems”, International Journal of Man-Machine Studies, 15, 1981, pp. 3-50.
[Neal 88]
J. Neal, C. Thielman, K. Bettinger, J. Byoun, "Multi-modal References in Human-Computer Dialogue", Proceedings of AAAI-88, 1988, pp. 819-823.
[Nigay 91a]
L. Nigay and Coutaz, Software design rules for multi-agent architectures, Amodeus BRA 3066 RP2/WP17, 1991.
[Nigay 91b]
L. Nigay and J. Coutaz, “Building user interfaces : Organizing software agents”, proceedings of ESPRIT’91 conference, Bruxelles, November, 1991.
[Nigay 93]
L. Nigay, J. Coutaz, D. Salber, “Matis, a Multimodal Air Travel Information System”, The Amodeus Project, Esprit Basic Research 7040, Amodeus Project Document, System Modelling/WP10, 1993.
[Nigay 94]
L. Nigay, " Conception et modélisation logicielles des systèmes interactifs : application aux interfacs multimodales". Thèse de doctorat de l'Université Joseph Fourier, 1994.
[Norman 86]
D.A. Norman, “Cognitive Engineering”, in User Centered System Design, D.A. Norman & S.W. Draper, Lawrence Erlbaum Associates, pp. 31-61, 1986
[OSF 89]
OSF, “OSF/Motif, Programmer's Reference Manual”, Revision 1.0; Open Software Foundation, Eleven Cambridge Center, Cambridge, MA 02142, 1989.
[Paterno 93]
F. Paterno, “A Methodology to Design Interactive Systems based on Interactors”, The Amodeus Project, Esprit Basic Research 7040, Amodeus Project Document, System Modelling/WP6, 1993.
[Pfaff 85]
G.E. Pfaff and co-workers, User Interface Management Systems, G.E. Pfaff ed., Eurographics Seminars, Springer Verlag, 1985.
[Scheifler 86
R.W. Scheifler and J. Gettys, “The X Window System”, ACM Transactions on Graphics, 5(2), April, 1986, pp. 79-109.
[Schmucker 86]
K. Schmucker, “MacApp: An Application Framework”, Byte, 11(8), 1986, pp. 189-193.
[Thimbleby 90]
H.W. Thimbleby, User Interface Design, ACM Press, Addison Wesley, 1990.
[Valdez 89]
J. Valdez, “XVT, a Virtual Toolkit,” Byte ,14(3), 1989.
[Webster 89]
B. F. Webster, The NeXT Book, Addison Wesley, 1989.
[Whitefield 91]
A. Whitefield, F. Wilson & J. Dowell, “A framework for human factors evaluation”, Behaviour and information technology, vol. 10, no. 1, pp. 65-79, 1991

Annex 1
Software Development Processes

This section is based on an unpublished document written by D. Duke, York, and on a paper by J. Coutaz, LGI, presented at the ICSE'94 workshop on HCI&Software Engineering [Coutaz 94].

A1.1. Historical background

At the start of the 70's it was realised that a key problem in software development was that the development process itself was poorly defined and often implicit. Drawing on concepts from other engineering disciplines, Royce proposed a model of the development process which has since become known as the `Waterfall Model' because of its characteristic structure. In light of experience, it turned out that revisions to this model were required. One was simply to distinguish more phases in the development process. The second was to make explicit backtracking into earlier phases to support iterative development.

An important mechanism that leads to backtracking is the various kinds of testing that need to be carried out on the artefacts generated during the development process. This includes formal tests such as the correctness of a unit of a code, and less formal (but no less important) checks such as validating the product against user requirements. In fact, closely related to the development waterfall there is a second tier of processes concerned with verification (are we building the product right?) and validation (are we building the right product?) [Boem 81].

Verification and validation activities are explicitly represented in the "V" life cycle model.

A1.2. The "V" model

Figure A1.1 shows the steps of the V life cycle development model. We now consider each step in turn, and briefly describe its role in the overall development process.

Feasibility study: This is concerned with identifying needs for change within some organisational context and in determining whether a software system is an appropriate solution. This may be outside the scope of software engineering since many issues here are managerial or sociological, but knowledge of software engineering will be important in accessing feasibility of options.

Requirements analysis: Having determined that there is a need for a software system in some context, the next step is to determine the goals and constraints of the system in consultation with its intended users. The output at this level should be understandable by both developers and users. Recently there has been interest in using formal languages to capture requirements, and thus the distinction between this phase and the next (specification) is becoming increasingly fuzzy.

Software specification: the role of specification is to translate the requirements, which are often vague and ambiguous, into a document that precisely states the required behaviour of the software product. The specification informs the developers about what they need to build and is the benchmark for verifying the correctness of the software and validating its adequate. Deliverables of this stage include the external specifications of the software, i.e., the portion of the system that is perceivable to the user.

Architectural design: The concern here is in developing a suitable software architecture to meet the functional requirements imposed by the specification and other general constraints such as performance, maintainability, and usability. This phase will typically identify the components which will make up the system and produce an interface specification for each such structure.

Detailed design: Each software component identified in the previous phase is given a detailed design specification which includes: name, summary of purpose, assumptions, interface, error handling, algorithms, and details of all information passed into and out of the structure. Also considered here are the test cases for verifying the module.

Coding: The program code for each module is developed and documented, test data is generated, and various *test harnesses* and *stubs* (programs used for testing the program) are produced.

Unit testing: Each module or software unit is tested in isolation, and minor errors are corrected. Significant problems may require a re-evaluation of earlier products.

Integration: Although each component may function correctly in isolation, problems may occur when the modules are linked together to produce the complete system. Integration proceeds incrementally, following a plan developed in the architectural design phase. Modules are connected one by one into a partially complete system, and each addition is followed by tests to ensure that the resulting system functions correctly.

Acceptance testing: Previous testing phases have concerned verification; once the final product has been generated it must be validated. This will involve evaluation of the product by the customer with respect to the requirement document.

Operational testing: Even products that pass acceptance testing may still exhibit problems when used operationally in the work context for which they were designed. In the worst case, failure in feasibility or requirements analysis may mean that the system is fundamentally unusable. More typically, extended use will reveal development mistakes (euphemistically called `bugs') or other problems such as performance or poor reliability.

Operation and maintenance: The context in which a product is used is rarely static and changing requirements mean that the software may need to be updated or enhanced with new functionality. Such upgrades also provide an opportunity to correct latent problems. However, changes to software systems increasingly lead to structural degradation and eventually it is no longer cost effective to upgrade an existing system. The life cycle must then start again with a re-evaluation of needs and requirements and the development of a replacement.

The V model is acceptable in terms of backtracking but explicit usability testing appears too late in the development process: although tests are devised for each step early in the development process, actual testing occurs once implementation has taken place. A single late evaluation of usability is too risky in terms of resources and quality objectives. The literature witnesses too many cases where late usability testing results in substantial changes that are unfavourably received by the implementers or too costly to implement. Clearly, early and continuous testing is a useful practice provided that it is performed in a cost effective way.

The spiral model offers an appropriate framework to our problem.

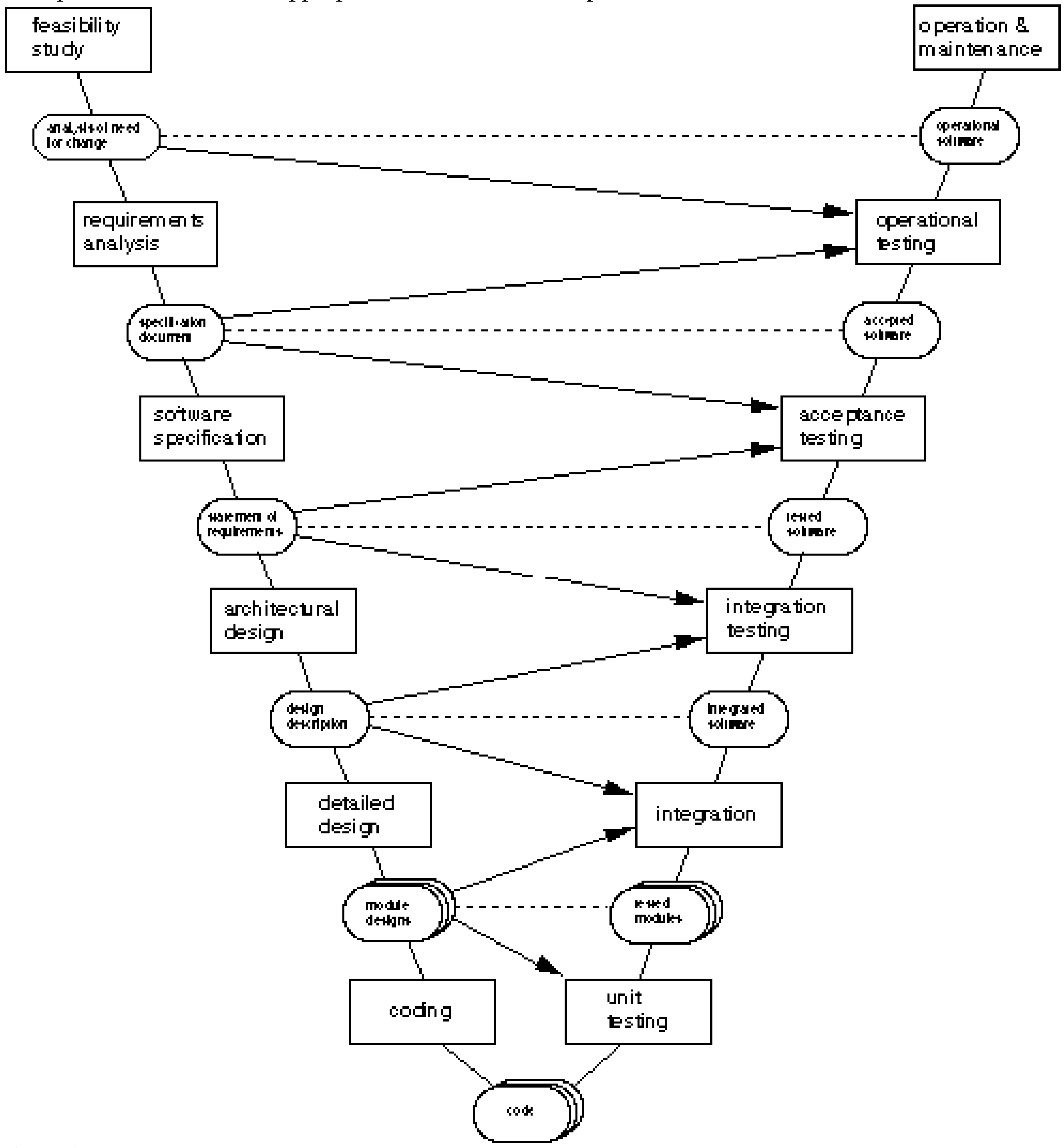


Figure A1.1: The "V" life cycle model (from McD 91]. For each artefact developed on the downward slope of the `V', there is a corresponding version of the software on the upward slope. The dashed lines across the `V' indicate testing of the product against various development steps. Arrows across the gap indicate where an artefact from the development side is used in testing the software on the Validation&Verification side.

A1.3. The Spiral model

Figure A1.2 shows the spiral model of software development [Boem 88]. The radial dimension represents the cumulative cost of development while the angular dimension represents progress through the development process.

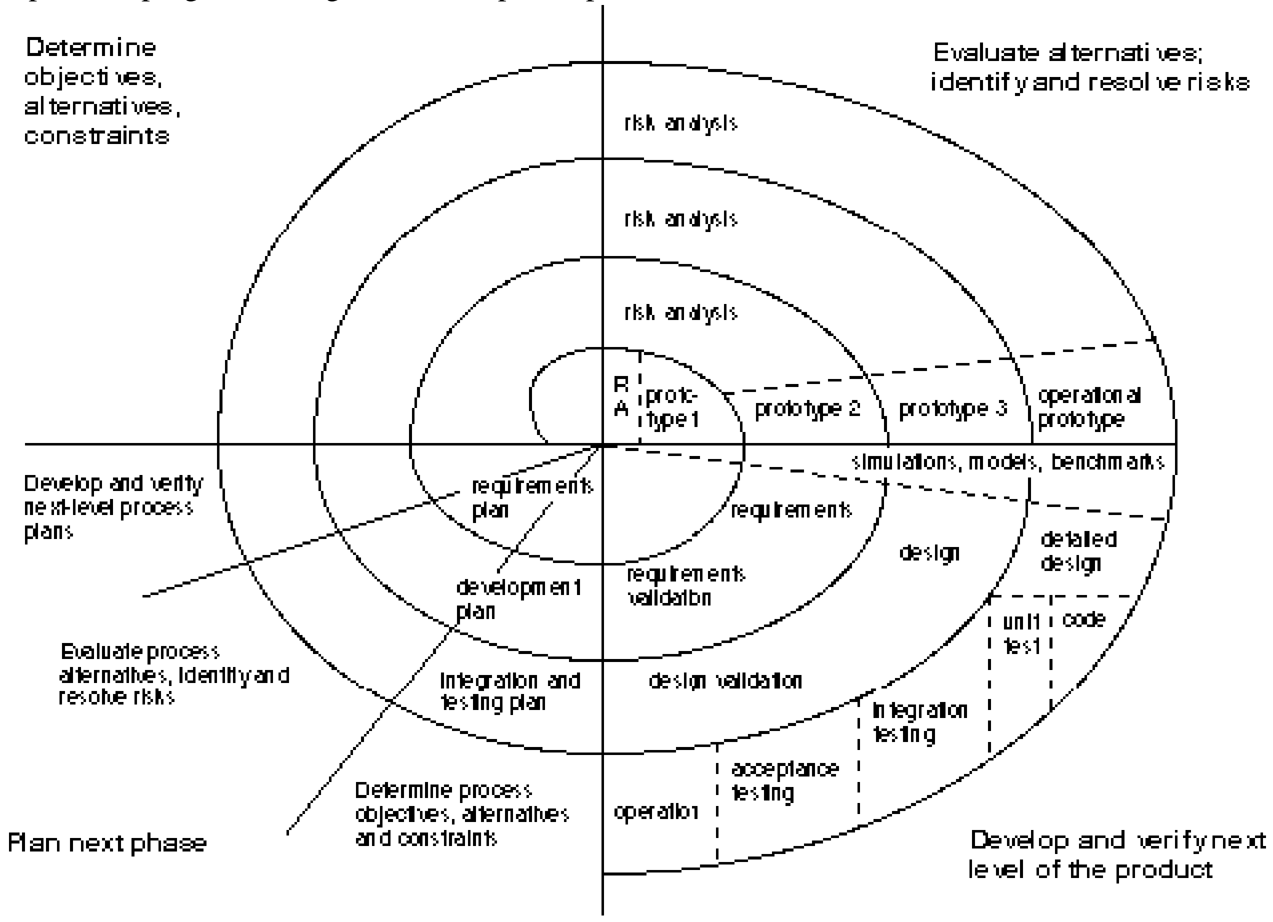

Determine objectives, alternatives, constraints
Evaluate alternatives; identify and resolve risks
risk analysis
risk analysis
risk analysis
R A
prototype 1
prototype 2
prototype 3
operational prototype
simulations, models, benchmarks
Develop and verify next-level process plans
requirements plan
requirements
design
detailed design
development plan
requirements validation
unit test
code
Evaluate process alternatives, identify and resolve risks
integration and testing plan
design validation
integration testing
acceptance testing
Determine process objectives, alternatives and constraints
operation
Plan next phase
Develop and verify next level of the product

Figure A1.2: The Spiral model of software development.

Each cycle around the spiral begins in the upper left quadrant with the determination of objectives related to the next phase of development (performance, functionality, etc.), the means of achieving those objects (design alternatives, software reuse. etc.) and constraints on those alternatives, such as cost, personnel. etc. The top right hand quadrant involves an evaluation of the alternatives, and this may involve building prototypes, developing analytic models, consulting appropriate experts (e.g., human factors).

Given an analysis of risks, the utility of the spiral now comes to bear in that the development step now chosen depends on the nature of the risks; a technically risky phase may be best addressed by an evolutionary development step, while if technical and prototyping risks are minimal, the next phase may be to follow a traditional life cycle model like the waterfall. Finally, in the lower left quadrant, the next phase is planned, and the cycle completes with a review of the previous cycle to ensure that all parties (technical, users, managerial) are satisfied that development can continue into the next phase and cycle.

Clearly, the spiral model supports user interface design through successive refinements and complementary testing, minimising the replication of expensive experiments. The difficulty however is to instantiate the appropriate phases for the particular case at hand and for each step, to identify the appropriate usability tests. The identification of the phases relies on a risk and cost analysis that requires a lot of expertise in project management. In addition, it is too tempting for a software project manager to define the phases in relation with technical constraints, and to ignore or underestimate the importance of the user-centred principle.

The spiral model is certainly suitable for the development of prospective and innovative systems whose ultimate functions are unclear. It is less practical for well structured projects whose development is limited in time. In this case, software engineers would favour the V model easier to apply and more tractable. One way then to accommodate "design for usability" within the V model is to perform targeted and complementary usability tests at every step of the development process. Modelling techniques developed by the Amodeus consortium can be incorporated in a complementary way within the various steps of the development process.

Annex 2
The MSM framework

This annex is based on an article by Coutaz, Nigay & Salber presented at EWHCI'93, Moscow [Coutaz 93].

A2.1. Introduction

Parallel to the development of the Graphical User Interface technology, natural language processing, computer vision, 3-D sound, and gesture recognition have made significant progress. Artificial and virtual realities are good examples of systems that aim to integrate these diverse interaction techniques. Their goal is to extend the sensory-motor capabilities of computer systems to better match the natural communication means of human beings.

The sensory-motor abilities of systems may be augmented with various degrees of sophistication. This extension may range from the construction of new input/output devices to the definition and management of symbolic representations for the information communicated through such devices. The span of possibilities and the novelty of the endeavour explain the variety of the terms, such as multimedia and multimodal, used to qualify these systems. As demonstrated by our framework, **multimedia and multimodal systems are both "multi-sensory-motor" (MSM) systems but with distinct degrees of built-in cognitive sophistication**.

As shown in Figure A2.1, MSM is comprised of 6 dimensions:

- The first two dimensions deal with the notion of physical devices : the number and nature of these devices (input/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 granularity of concurrency. These issues are discussed in detail before we comment on the distinction between multimedia and multimodal interactive systems.

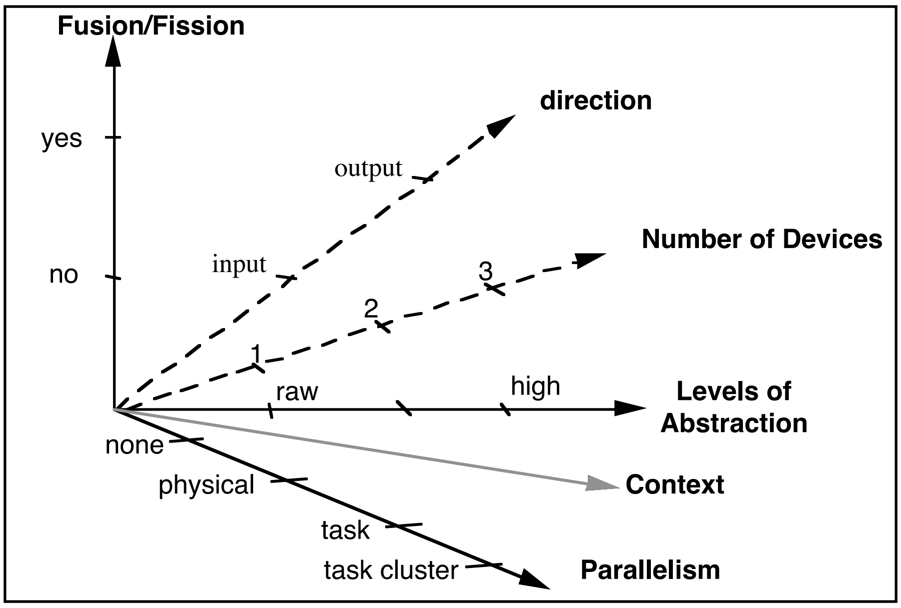


Figure A2.1. MSM: A 6-D space to characterise multi-sensory-motor interactive systems.

Information acquired by input physical devices is transformed through multiple process activities. This sequence of input transformations forms the acquisition or *abstraction function*. In the other direction, internal information (e.g., system state) is transformed to be made perceivable to the user. This sequence of output transformations defines the *concretization or rendering* function. The abstraction and the rendering functions can be both characterised with four intertwined ingredients: level of abstraction, context, fusion/fission, and parallelism. These dimensions are presented in the following paragraphs.

A2.2. Level of Abstraction

The notion of level of abstraction expresses the degree of transformation that the interpretation and rendering functions perform on information. It also covers the variety of representations that the system supports, ranging from raw data to symbolic forms. The span of representations should be considered on a per-digital channel basis. Thus, for a given digital input channel, the abstraction function can be characterised by its power of "abstracting" raw data into higher representational expressions. The rendering function is characterised by the level of abstraction it starts from to produce perceivable raw information through output digital channels.

Computer vision, speech recognition as well as speech synthesis systems operate along these principles. For example, speech input may be recorded as a signal, or described as a sequence of phonemes, or interpreted as a meaningful parsed sentence. Each representation corresponds to a particular level of abstraction resulting from an abstraction function. For output, the process is similar: data may be produced from symbolic abstract representations or from a lower level of abstraction without any computational knowledge about meaning. For example, a vocal message may be synthesised from an abstract representation of meaning, from a pre-stored text (i.e., text-to-speech) or may simply be replayed from a previous recording.

A2.3. Context

The capacity of a system to abstract along a channel may vary dynamically with respect to "contextual variables". More formally, a *context* is a set of state vectors used by the system to control the interpretation/rendering functions. For example, in vi, when in command mode, typed text is transformed into a high level abstraction whereas the same text entered in input mode is recorded as is without any transformation. Context constrains the configuration of digital processes used at some point in time to process information. We observe an analogy with the cognitive resources configuration claimed in ICS.

Figure A2.2 shows one possible set of state vectors of a Context. These are defined along four dimensions : topic, variability, persistence, and the owner.

- Topic includes:

  . data about the environment (e.g., noise and light which are respectively pertinent for speech recognition and computer vision), information about the user (e.g., login name as well as "the user wants to know", "the user knows", etc.),

  . data about the state of the user interface (e.g., the current mode as defined by Dix in [Dix 91]),

  . data about the dialogue strategy (e.g., reactive, directive, co-operative, negotiated, and intentional [Bourguet 92]),

  . data about the functional core (e.g., state variables that are of interest for the task).

- Variability covers the capacity of topical data to vary over time. For example, in the user state vector, the login name is constant whereas the beliefs may vary through interaction.

- Persistence denotes the property of topical data to be memorised across *sessions*. In particular, for a given user, the login name is persistent whereas in a non monotonic reasoning scheme, the facts that the system knows about the user are transient.

- the owner denotes the agent (i.e., a software component) that is in charge of maintaining a particular state vector.

Detailed description of the notion of context can be found in [Nigay 94].

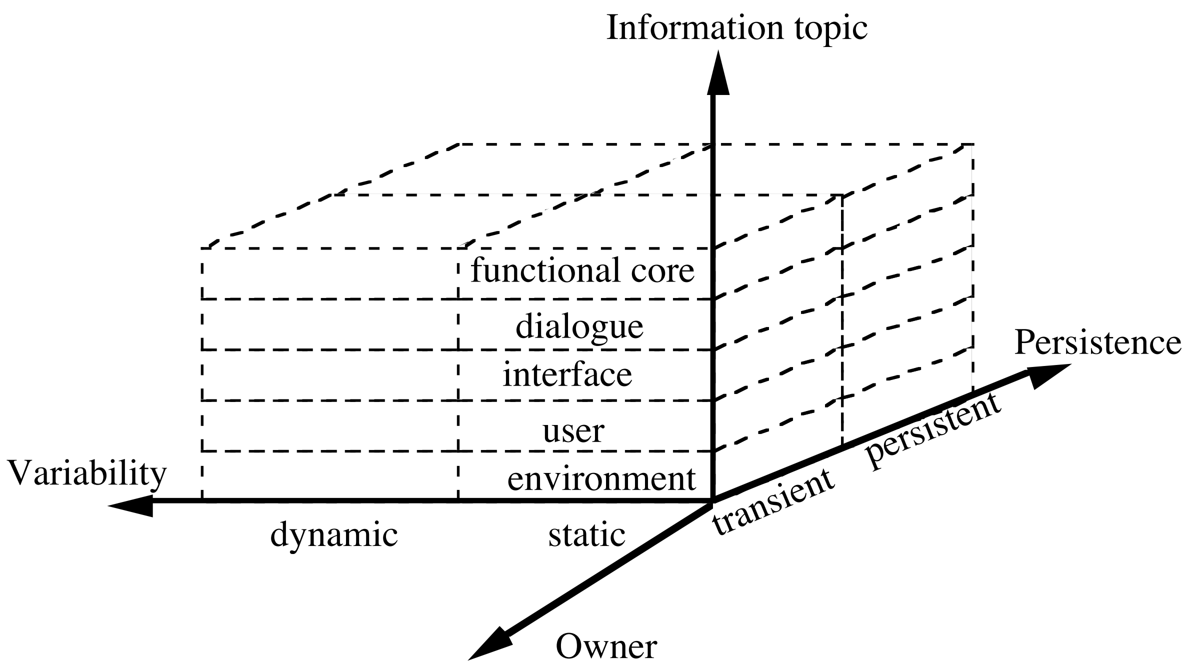


Figure A2.2: the dimension space of the notion of Context.

A2.4. 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 concretization phenomena.

The Abstraction function and Fusion. Considering fusion for the abstraction function:

- at the lowest level, information chunks may (or may not) originate from distinct input devices;
- at higher levels, information chunks may (or may not) come from distinct context owners.

For example, the sequence of events "mouse-down, mouse-up" that occurs in the palette of a graphics editor are two information chunks that originate from the same input device and from the same context owner (i.e., the palette). They are combined within the context of the palette to form a higher information chunk (i.e., the selection of a geometric class). The drawing area constitutes another context owner. Events that occur in the drawing area are interpreted as the effective parameters of the geometric function. They are combined with the selected geometric class to complete the function call in the task-domain. Thus, in this example, fusion occurs between information chunks originating from the same input devices but, as the interpretation proceeds at higher levels of abstraction, it also involves different context owners.

The "put that there" paradigm as in Cubricon [Neal 88] and ICP-Plan [Bourguet 92] offers an example of fusion between chunks originating from distinct input digital channels. In this example, fusion is required to solve the co-references expressed through distinct input devices.

The Abstraction function and Fission. It may be the case that information coming from a single input device or from a single context owner need to be decomposed in order to be understood at a higher level of abstraction.

For example, consider the utterance “show me the red circle in a new window”. This sentence, received through a single input device, references two domains of discourse: that of the graphics task (i.e., “the red circle”) and that of the user interface (i.e., “a new window”). In order to satisfy the request, the system has to decompose the sentence into two high level functions: “create a window” and “draw a red circle” in the newly created window.

The Rendering Function and Fusion. The rendering function can perform fusion at multiple levels of abstraction. One of them, which takes place at the highest level of abstraction (i.e., the domain adapter, see the Arch Model in Annex 3) has been discussed in [Coutaz 91]. At the lowest level, it appears as multiple information chunks rendered through a single output channel.

For example, the picture of a town may be combined to a graphical representation of the population growth. The notions of town and population which are handled by two different context owners within the internal processes of the system, are combined at the lowest level and presented through a single output device.

The Rendering Function and Fission. Rendering may also incorporate fission at multiple levels of abstraction. The highest level has been discussed in [Coutaz 91] (see also discussion about the Arch Model in Annex 3). At the lowest level, fission occurs when an information chunk gives birth to multiple representations whether it be through a single or multiple output devices.

For example, the notion of wall in our mobile robot system [Bass 91] may be represented as a line or as a form on the screen. These distinct representations of the same concept use only one output devices. Alternatively, the spoken message “watch this wall!” along with a blinking red line on the screen uses two distinct output devices to denote the same wall.

A2.5. Parallelism

Representation and usage of time is a complex issue. In our discussion, we are concerned with the role of time within the abstraction and rendering functions. How does time relate to levels of abstraction and context? How does it interfere with fusion and fission? Parallelism at the user interface may appear at multiple grains: at the physical level, at the task and task cluster levels.

Parallelism at the Physical Level. For input, the physical level corresponds to the user actions that can be sensed by input devices as an information chunk (e.g., an X window event). For example, a mouse click, a spoken utterance are information chunks. For output, the physical level denotes output primitives, that is the information chunks that can be produced by output devices in one burst. For example, a spoken message or the reverse video of an icon.

For input, parallelism at the physical level allows the user to trigger multiple input devices simultaneously. If these devices are organised along distinct channels, then the user solicits multiple input devices in parallel. Similarly, physical parallelism for output may take the form of simultaneous outputs through distinct digital channels or may occur through a single channel. The fission example “watch this wall” associated with “the blinking red line”, requires parallelism at the physical level using multiple output devices.

Parallelism at the Task Level. From the system’s perspective, a task (i.e., an elementary task) cannot be decomposed further but in terms of physical actions. For input, an elementary task is usually called a command, that is, the smallest fusion/fission of physical user’s actions that changes the system state. For output, an elementary task is the set of output physical primitives used to express a system state change.

True parallelism at the command level allows the user to issue multiple commands simultaneously. It necessarily relies on the availability of parallelism at the physical level. Pseudo-parallelism at the command level as in Matis [Nigay 93], allows the user to build several commands in an interleaved way as in multi-thread dialogues. Then, parallelism at the physical level is not required.

Figure A2.3 illustrates all possible relationships between parallelism at the physical level, and fusion and fission, to form commands within the abstraction function. In A2.3-a, multiple simultaneous inputs from device 1 must be dispatched into two higher context owners (e.g., agents) to build two distinct commands in parallel. For example, in MMM, two users may manipulate two physical mouses simultaneously to respectively modify the size and colour of a shared rectangle [Bier 92]. In configuration A2.3-b, simultaneous actions on distinct input devices must be combined to build a single command (as in the “put that there” paradigm). In A2.3-c, physical actions follow two independent paths. For example, the user may say “close top window” while moving a file icon in the trash. In this case, two independent commands must be built in parallel.

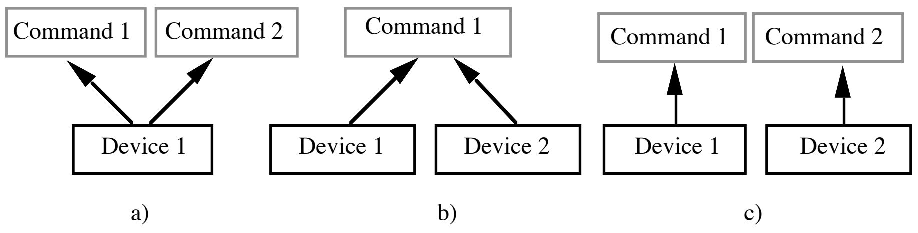


Figure A2.3. Relationships between parallelism at the physical level, fusion and fission, and commands.

The diversity of the relationships shown in Figure A2.3 is a good indicator of the difficulty to implement the abstraction function. In particular, which criteria should be considered to trigger fusion and which strategy should be adopted? The fusion mechanism presented in Level 3 of the Amodeus reference model is one first proposal.

A similar analysis should be done for output elementary tasks. So far, we have not experienced enough exemplars to generate a sound discussion on this issue. However, we can relate two interesting examples. The first one is usage of time to synchronise information chunks over output devices. QuickTime is a good illustration of this capacity. The second example is interleaving between inputs and outputs at the task level. For example, as the system moves an object, the user may dynamically change the speed or the colour of the object. We observe a temporal overlap between input and output at the task level. In this example, the duration of the system’s outputs covers the duration of a sequence of user’s commands and can be dynamically affected by these commands. In the other way round, rubber banding or reverse video of candidate recipients in the Macintosh finder are examples of duration of user's inputs covered by system's outputs.

Parallelism at the Task Cluster Level. From the system's perspective, a task is a cluster of tasks that structures the work space. For example, in our mobile robot system, the command space is organised into three subspaces: one for providing the robot with cartographic details, the second to specify missions to be accomplished, the third one to observe and control the robot during mission execution. For input, parallelism at the task level expresses how much parallelism (actually pseudo-parallelism) is supported by the system between clusters of commands. Note that parallelism at the cluster level does not necessarily imply parallelism at the command level.

For output, a similar organisation in terms of clusters of parallelism may be observed. We have not studied this perspective yet.

A2.6. Summary

The analysis of the behaviour of MSM interactive systems should be considered along the following dimensions:

- the nature of input and output physical devices,
- the granularity of parallelism supported by the system along the input and output devices (i.e., physical actions, task level, task cluster),
- for each device and context, and for combinations of input or output devices, the capacity of the system to support abstraction/rendering through the fusion and fission mechanisms.

Annex 3
Conceptual Software Architecture Models

This Appendix is based on a book chapter by Coutaz in Encyclopaedia of Software Engineering, O'Wiley, 1993 [Coutaz 93b].

A3.1. Introduction

The consensus in user interface software design relies on the distinction between the Functional Core and the User Interface. The *functional core* of an interactive system is the software portion that implements task domain concepts (i.e., concepts identified by task analysis as relevant to the user to accomplish the tasks in that domain). The *user interface* covers the components which define the perceivable behaviour of the system. It implements the abstraction and rendering functions (see Annex 2) which process information between the functional core and the physical input and output devices of the system. This distinction has been admitted as a sound basis for iterative refinement and has helped in the emergence of UIMS technology. Figure A3.1 illustrates the foundations of user interface software architecture modelling.

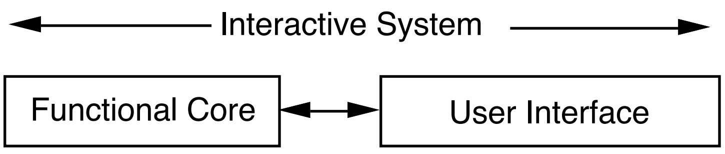


Figure A3.1 : The foundation of user interface software architecture modelling.

In the following sections, we present the successive refinements of the "Functional Core-User Interface" duality: First, the Seeheim model which focuses on the components embedded in the User Interface; then the Arch model and its generic companion, the Slinky metamodel, which consider software requirements and constraints such as efficiency and the existence of interaction tool kits; finally the multi-agent paradigm which stresses parallelism and modularity at multiple levels of abstraction. All of these models are basically usable as reference frameworks to assess the soundness of a particular architectural design depending on the system characteristics and on the project type (e.g., prototype, product, size, software life cycle).

A3.2. The basis : the Seeheim model

The term "Seeheim model" originated at a workshop in Seeheim, Germany [Pfaff 85]. As shown in Figure A3.2, this model refines the user interface portion of an interactive system into three components: the Application Interface[2], the Dialogue Controller and the Presentation component. The role of each component is roughly described as the semantic, syntactic and lexical functionalities of the user interface. The Application has the same definition as the Functional Core: it models the domain-specific concepts (i.e., the

[2] "Application Interface" and "Functional Core Interface" are equivalent expressions.

semantics), whereas the Dialogue Controller and the Presentation deal with syntactic and lexical issues respectively. The Application Interface defines the view that the Dialogue Controller has about the Application.

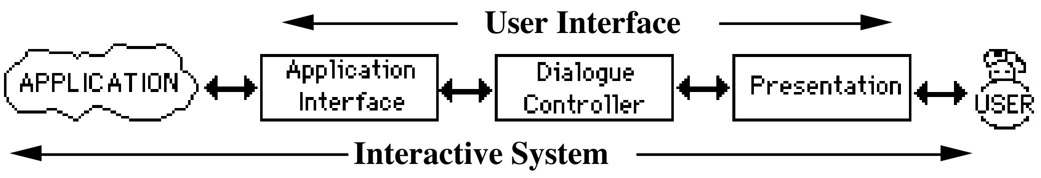


Figure A3.2: The Seeheim Model.

The Seeheim model is based on notions (semantics, syntax, and lexicon) which are well understood by computer scientists. Therefore, it is a useful framework for teaching how to devise the design process of an interactive system as well as how to organise its software architecture. Design methods such as CLG [Moran 81] and that of Mark Green [Green 85] and Foley and Van Dam [Foley &Van Dam 84], which advocate a top-down design approach (from the definition of the domain concepts to the identification of the interaction techniques used in the presentation) are consistent with the linguistic interpretation of the Seeheim model. With regard to software tools, the Seeheim model has opened the way to the transfer of knowledge from automatic compiler generation to automatic user interface generation. The result has been an emergence of a wide variety of rapid prototyping tools.

Unfortunately, experience has shown that user interface designs based on the linguistic approach present major limitations. By nature, a language view puts the emphasis on the form and is totally oblivious to the dynamics. This prejudice is acceptable for a compiler which processes fully-specified, static input expressions. In computer-human interaction and in particular in direct manipulation user interfaces, however, the form of an input expression may evolve in parallel with the production of an output expression; multiple input expressions may even be specified simultaneously (see MSM, annex 2). Unlike a compiler which performs a well-defined sequence of processing, the user interface must, in general, support parallelism. In addition, in order to augment the quality of domain-specific feedback, it should allow for partial input expressions to be interleaved with the production of output expressions; it should support a flexible granularity for input and output expressions; it should support multi-thread dialogues in order to allow the user to simultaneously handle several threads of reasoning as well as to behave in an opportunistic manner [Hayes-Roth 79].

Thus, applying a centralised sequential linguistic view on top of the Seeheim model, makes impossible the satisfaction of some of the most fundamental user-centred requirements. In addition, the Seeheim model does not provide software designers with any help to perform unavoidable and difficult engineering trade-offs. Such trade-offs affect the optimisation of the development process as well as the quality of the end product. The Arch model and its companion, the Slinky metamodel, aim at filling the gap.

A3.3. The Arch Model

The Arch model resulted from the work of a user interface tool developers group [Arch 92]. Its purpose was to provide developers with a framework for understanding engineering trade-offs. It is not intended to be prescriptive but rather usable as a reference for evaluating a particular candidate run-time architecture. The authors observe that "user interface developers sometimes find both the application domain functionality[3] and the User Interface Tool kit(s) to be existing constraints upon the development of a user interface... For this reason, the domain software and the User Interface Tool kits form the two bases of the Arch model." [Arch 92, p. 34]. As shown in Figure A3.3, the Arch model is a refinement of the Seeheim model where engineering reality has been injected. In addition, the Arch model makes explicit the nature of the information that crosses the boundaries between the components.

The Components of the Arch Model
The Domain-Specific Component and the Domain Adapter Component are different terms for denoting the notions of Application and Application Interface introduced by Seeheim. Architecture models however, have a better understanding of the role of the Domain Adapter[4] as well as on the nature of the data exchanged between the Domain-Specific Component and the Domain Adapter. These issues will be further discussed in a later paragraph about Adapter Components.

The Dialogue Component corresponds to the Seeheim Dialogue Controller although in Seeheim, the role of this component was limited to some obscure syntactic sequential processing such as the combination of lexical inputs into command level abstractions. In the Arch model, the Dialogue Component "has responsibility for task-level sequencing, ..., for providing multiple view consistency and for mapping back and forth between domain-specific formalisms and user-interface-specific formalisms" [Arch 92, p. 34].

The domain-specific formalism describes the entities that are exchanged with the Domain Adapter whereas the user-interface-specific formalism expresses semantically identical information driven by presentation considerations. For example, the notion of temperature in the Domain Adapter would be represented as a real number whereas, at the presentation level, it would be represented as the mercury height of a thermometer. The first formalism, i.e., the real number, is suggested by computational purpose whereas the second formalism, a mercury height, depends on the formalism provided by the Presentation component (e.g., graphics, voice).

The Arch model segments the Seeheim Presentation into two levels of abstraction: The Presentation Component and the Interaction Tool kit Component. The Interaction Tool kit Component implements the physical interaction with the end-user. The Presentation Component "provides a set of tool kit-independent objects for use by the Dialogue Component ..." [Arch 92, p. 34]. A tool kit-independent object is called a "presentation object" whereas "interaction objects" refer to interaction techniques supplied by interaction tool kits. For example, a "selector" presentation object can be implemented in the tool kit using either a menu or radio buttons interaction objects. Therefore, the Presentation Component acts as the second software adapter of the Arch model.

[3] "Application domain functionality", "Application", and "Functional Core" are equivalent expressions.

[4]The "Domain Adaptor" is also called the "Functional Core Adaptor" by Nigay and Coutaz (Nigay & Coutaz, 1991b).

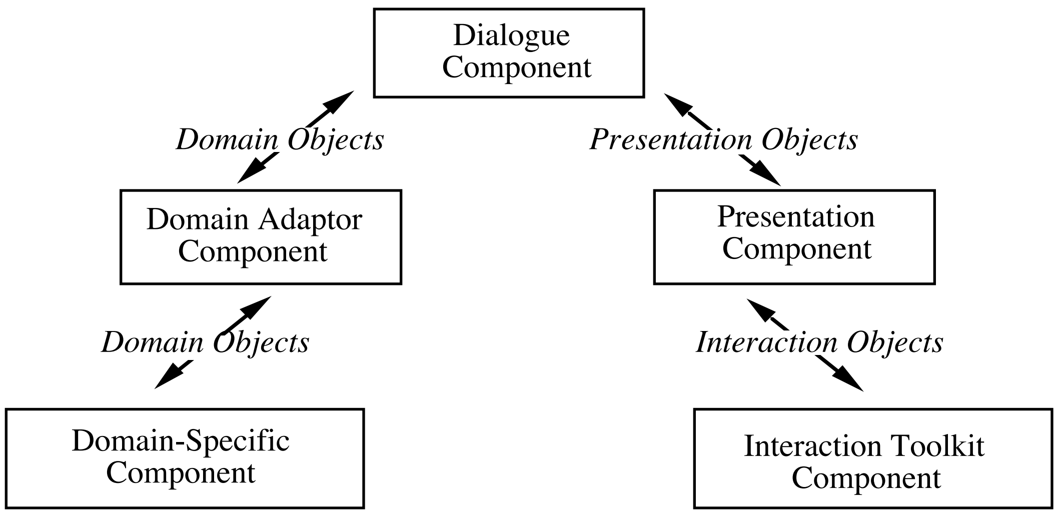


Figure A3.3: The components of the Arch model and their interfaces. [Arch 92]

The Adapters of the Arch Model

Adapter components, which define abstract interface machines, are useful concepts to improve code re-usability, portability, and modifiability. This subsection discusses two instances of such adapters in user interface software engineering: the Domain Adapter and the Interaction Tool kit Adapter, which insulate the keystone component (i.e., the Dialogue Component) from modifications in its mandatory neighbours: the Functional Core and the Interaction Tool kit.

The Domain Adapter Component

The Domain Adapter Component serves as a mediator between the Dialogue Component and the domain-specific concepts implemented in the Functional Core [Nigay 91b]. It is designed to absorb the effects of change in its direct neighbours. As any boundary, it implements a protocol. A protocol is characterised by temporal strategies and by the nature of data exchanged [Coutaz 91].

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

When considering interaction styles, synchronous co-ordination results in single threads of dialogue or, at best, in interleaved threads. Asynchronous co-ordination supports multiple concurrent threads of dialogue: the user may issue multiple commands simultaneously (using for example, a combination of voice and hands) while the Functional Core may have its own processing going on.

Exchange of data between the Functional Core and the user interface is performed through the Domain Adapter in terms of domain objects[5]. A domain object is an entity that the designer of the Functional Core wishes to make perceivable to, and manipulable by the user. 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 91] may be performed in the Domain Adapter by defining domain objects that reorganise the information modelled by the Functional Core. Reorganising may take the form of aggregating data structures of the Functional Core into a single domain object (i.e., fusion in the MSM sense, at a high level of abstraction, see Annex 2) or, conversely, segmenting a concept into multiple domain objects (i.e., fission, in the MSM sense, at a high level of abstraction, see Annex 2). 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.

The Presentation Component
The Arch Presentation Component acts as a mediator between the Dialogue Component and the interaction tool kit. As shown in Figure A3.4, Coutaz proposes to refine the Presentation Component into two layers of abstraction: the Extension Layer and the Interaction Tool kit Adapter [Coutaz 91]. The Interaction Tool kit Adapter defines a virtual tool kit used for the expression of presentation objects. This expression is then mapped into the formalism of the actual interaction tool kit used for a particular implementation. Switching to a different tool kit requires rewriting the mapping rules, but the expression of the presentation objects remains unchanged.

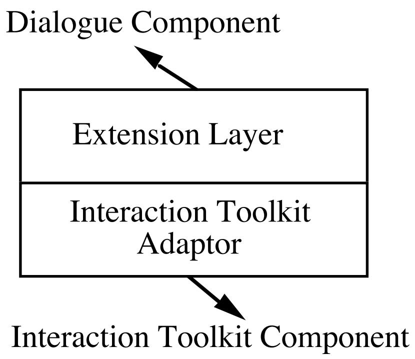


Figure A3.4: Refinement of the Presentation Component. [Coutaz 91]

Interaction objects are generally constructed from entities made available in interaction tool kits. In general, interaction tool kits such as X Intrinsics [OSF 89], 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 tool kit. For example, in an earlier version of X intrinsics, widgets (i.e., interaction objects or interactor) would occupy rectangular areas only. Under such conditions, the notion of a wall in a floor plan

[5]The term "object" should be considered here in the general sense as an entity. It does not suppose any particular implementation technique such as a class, a function, or a shared data structure but covers all of them.

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 tool kit.

The wall example shows that the Presentation 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 89], defines the boundary between these two layers.

Sometimes, the location for implementing a presentation object is not straighforward. For example, the thermometer used to present the notion of temperature could be implemented either at the toolkit layer or as part of the extension layer. If located in the tool kit, then the thermometer becomes a general purpose interaction object and thus, should be implemented according to the programming rules imposed by the tool kit to guarantee reusability. If located in the extension layer, the private status of the presentation object relaxes the reusability constraints of the underlying platform.

The thermometer example is merely a simple illustration of a more general problem: that of identifying the appropriate location for software functionalities. The Slinky metamodel is an attempt to provide an answer to this difficulty.

A3.4. The Slinky Metamodel

Experience shows that no single architectural model can satisfy all of the software design factors and criteria such as those reported by McCall [McCall 77]. Factors and criteria may be conflicting. For example, portability and modifiability may impede efficiency. In the case of the Arch model, two adapter components have been introduced to minimise dependence on Functional Core modifications as well as on the effective user interface tool kit. 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 modular distribution of functionalities and response time. The software foundations illustrated in Figure A3.1 stress that domain-specific objects should be confined in the Functional Core and the Functional Core Adapter. 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 may be costly with respect to the system response time. Therefore, it may be inconsistent with the expectation of the user.

Domain-knowledge delegation[6], which consists of down-loading functional core knowledge into the user interface is a way to reduce transmission load at critical points [Coutaz 91] and thus 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

[6]The term “semantic delegation” was previously used by Coutaz to denote “domain-specific delegation”.

knowledge is readily available in the user interface 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. "The term "Slinky" was selected to emphasise 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 92, pp. 35].

Thus, the Slinky metamodel is a generic framework from which particular instances of Arch models can be derived. For example, if efficiency prevails against tool kit portability, then the Interaction Tool kit Adapter may not be needed. If high quality semantic feedback and efficiency are important requirements as in semantically rich rubber-banding tasks, then domain-knowledge delegation may be performed. Such decision may result in reducing the relative importance, thus the code size, of the Functional Core. Multi-agent architectures presented in next Section provide a way to perform domain-knowledge delegation without jeopardising the basic "separation of concerns" principle illustrated in Figure A3.1.

A3.5. Multi-agent Models

Multi-agent models currently used for the software design of interactive systems are based on the functioning of stimuli-response systems. They structure an interactive system into a collection of specialised agents or interactor that have a state and produce and react to events (i.e., stimuli).

Agents
Agents of an interactive system that communicate directly with the user are called "interactors". An interactor provides users with a perceptual representation of its internal state [Duke 92, Duke 93]. It listens to them, it possesses an expertise which is convenient for them, and it provides them with an adequate feedback. Interactors are also coined as "interaction objects" (as in the previous section). An object is a generic term that covers the notion of computing entity 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.

An agent is a kind of object. Unlike some objects that may be passive (i.e., manipulated only), an agent is an active entity. For example, in the Serpent UIMS [Bass 91], the Functional Core Adapter is as a data base of passive domain objects. The data base itself is active (it is an abstract machine, an agent) but the domain objects that model the data base state are passive. Conversely, the interaction objects of the Interaction Tool kit Component are active. They are agents. In addition, they are in contact with the user: they are interactors.

The following expression clarifies the relationships between the sets of objects, agents and interactors:

an interactor *is-an* agent *is-an* active object

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 91]. 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, interactors are reactive agents. 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 consider reactive agents implicitly.

Benefits from Multi-agent Models
Multi-agent models that stress a highly parallel modular organisation 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:

•*Support for iterative design.* An agent defines the unit for modularity. It is thus possible to modify its behaviour without endangering the rest of the system.

•*Support for distributed applications.* 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 several workstations. This property is essential to the implementation of systems that support collaborative work.

•*Support for multi-thread dialogues.* An agent can be associated to each 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, then it is possible to use a collection of co-operating agents.

In addition to satisfying the requirements for better user interfaces, the multi-agent model 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, and where an event class denotes a method. An object and an agent are both highly specialised processing units, and both decide 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 multi-agent models and tools have been developed along the lines of the object-oriented and the event processing paradigms. But all of them deal with reactive agents only. MVC [Goldberg 84], PAC [Coutaz 87], ALV [Hill 92] are multi-agent models. Interviews [Linton 86] and Aïda [Ilog 89] are examples of toolkits based on such a model, whereas Serpent [Bass 91] and Sassafras [Hill 87] are run time kernels and user interface generators organised according to a multi-agent model. To summarise, none of these models and tools deny the principle of separation of the functional core from the user interface. They simply go further.

Multi-agent models 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.

A3.6. Comparative analysis of multi-agents models

We propose to compare multi-agent models according to their structure and according to the MSM dimensions (see Annex 2 for detailed description of MSM).

Comparative analysis based on agent structure

In MVC (model, View Controller), an agent is modelled along three perspectives: the Model, the View, and the Controller. A Model defines the functional competence of the agent (i.e., its task-domain). The View corresponds to the rendering function. It defines the perceivable behaviour of the agent. The Controller denotes the abstraction function of inputs. The View and the Controller define the user interface of the agent, i.e., its behaviour with regard to the user.

PAC (Presentation, Abstraction, Control) conveys similar ideas: a PAC agent has a Presentation perspective (its rendering and abstraction functions), an Abstraction (its task domain competence), and a Control. The Control is in charge of communicating with other agents as well as bridging the gap between its abstract and user interface facets.

ALV (Abstraction, Link, View) has a similar decomposition to PAC except that a Link has a more restricted role that 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.

In CNUCE, an agent possesses four perspectives : 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.

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

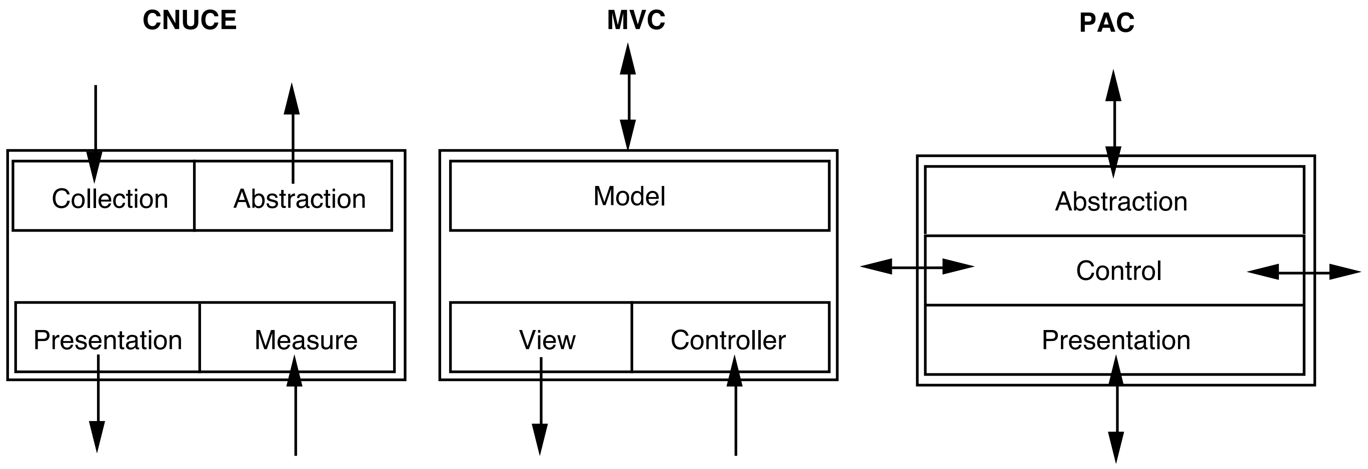


Figure A3.5: Equivalence between the perspectives of MVC, PAC and CNUCE agents. Arrows indicate the directions of inputs and outputs.

Figure A3.5 summarises the commonalities and the differences between MVC, PAC and CNCE:

- an MVC “View” <u>is equivalent to</u> a CNUCE “Presentation”.
- an MVC “Controller” <u>is equivalent to</u> a CNUCE “Measure”.
- an MVC “Model” <u>is equivalent to</u> a PAC or ALV “Abstraction” <u>is equivalent to</u> a CNUCE “Abstraction, Collection” couple.
- an MVC “View, Controller” couple <u>is equivalent to</u> a CNUCE “Presentation, Measure” couple <u>is equivalent to</u> a PAC “Presentation” <u>is equivalent to</u> an ALV View.
- a PAC “Control” has no explicit correspondence with the MVC facets. With regard to CNUCE, it covers the ALV Link and the internal co-ordination between the components of an agent as well as the notion of “Controller” introduced in [Faconti 93] to control the behaviour of an agent.

In summary, MVC decouples input techniques from outputs, whereas ALV, PAC, York and Interviews concentrate them in the notion of Presentation; contrary to PAC and ALV, MVC has no explicit notion of arbitrator for expressing the relationships and the co-ordination between the agents. In the Lisbon model, agents do not have perspectives but are organised as specialised categories to model the rendering and abstraction functions.

Comparative analysis along the MSM dimensions

In this paragraph, the discussion is organised along the dimensions of the MSM framework: input and output, levels of abstraction, context, fusion and fission, and parallelism.

input and output
All of the models considered in our discussion stress the fact that an agent may communicate with multiple agents. Agents are multi-channel capabilities. However, not all of the multi-agent 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 serves 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.

Levels of Abstraction
Information acquired by interactors is transformed by a population of agents before reaching the functional core. This transformation process, called the abstraction function in the MSM framework, defines the capacity of abstracting of the user interface. In the other direction, the rendering function corresponds to the capacity of the user interface to concretise information from to functional core into the formalism acceptable for interactors. The successive steps of such 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 abstraction 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 [Faconti 93]. 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 91]. 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 91]. 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.

Context
According to our definition for the notion of context (i.e., a set of state vectors with owners) and that context has an impact on the rendering and abstraction functions, agents (which are objects capable of initiating the performance of actions) are good candidate for being the owners of contexts.

Fusion and Fission
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 their computation).

Parallelism
An agent is a processing unit that can operate in parallel with other agents. It results from multi-agent architectures that multiple abstraction and rendering activities can take place simultaneously.

A3.7. Conclusion

Early experience as well as our analytic description based on the properties and concepts of Level 1 of the Amodeus reference model, indicate that multi-agents models are the way to go. They provide a good material for abstract reasoning (see the CNUCE and YORK formalisms) as well as for designing the global conceptual architecture of interactive systems. When the software designer needs to consider criteria related to technical constraints (e.g., portability of the user interface or pre-existence of the functional core), PAC-Amodeus, which is a blend of Arch and PAC modelling techniques, can be exploited fruitfully before specifying the detailed architecture of the system. PAC-Amodeus is described in detail in Section 4.2 of this report.