INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Université d'Ottawa • University of Ottawa

COORDINATION ORIENTED ARCHITECTURE FOR LARGE SOFTWARE SYSTEMS

by

(C) Gabriel M. Coifan

A thesis submitted in partial fulfillment of
the requirements for the degree of

Master of Applied Science,

Electrical Engineering

School for Information, Technology and
Engineering, University of Ottawa,
Ontario

May 1999

University of Ottawa, Ontario

Abstract

COORDINATION ORIENTED ARCHITECTURE FOR LARGE SOFTWARE SYSTEMS

After nearly fifty years of existence, software development evolved from craftsmanship to science, only lately breaking grounds towards an engineering discipline. The last ten years brought along a tremendous development of the software industry, fueled by a multitude of factors like the exponential increase in the computing power (associated with lower costs), combined though with increasing user needs, expectations, and increasing costs of software development. This rapid advance caught the software community ill prepared and suddenly uncovered what proved to be the two most important problems of the software industry today: how to derive proper requirements for a wide class of applications and how to handle the complexity of the solution.

A number of significant attempts have been made so far in order to cope with complexity; taken chronologically, modularization and information hiding, object orientation, design patterns and software architectures are the most popular of them. None of them is complete; for good design one needs to have, along with the appropriate structure, the matching design methodologies, processes and tools. Development of more reliable software at a reasonable cost is more and more of a problem; today's demands from software products outgrow development capabilities. By analogy with more established engineering fields, it is clear that one needs disciplined and well defined requirement management, design and implementation methodologies in order to develop quality software.

The purpose of the thesis is to introduce coordination-based design, as a step forward towards managing software complexity on the architectural level and improving non-functional characteristics of software systems: reliability, efficiency, maintainability, and usability. We also provide a critical view on basic ideas, trends and terminology in Software Architecture, develop a framework to define the need for Software Architecture and properly place it in the design context.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

I wish to express my appreciation and acknowledge the meaningful advice and guidance of Dr. Moshe Krieger, my supervisor and friend, whose contribution and words of wisdom were instrumental throughout my post-graduate degree. His previous work on Coordination Based Design provided much of the conceptual underpinning for the research described in this thesis. I feel rewarded to have been one of his 'schmucks'.

I would also like to acknowledge the support and understanding of my family and friends for never complaining about my rather asocial behavior, especially during the difficult period of writing the thesis. A special mention here deserves my current employer, Nortel Networks, which has been kind enough to let me work only the standard hours on the very final stretch of this work.

And last but not least, my heartfelt love goes to my wonderful parents. Even from far away, they always saw me through this effort with unfailing love and encouragement, only lately with that keen question over the phone: 'Haven't you finished yet?' To them, this work is dedicated.

LIST OF ACRONYMS

ADL	Architecture Description Language
CBD	Coordination Based Design
CODE	Coordination Oriented Development Environment
COTS	Commercial Off-the-Shelf
DSSA	Domain Specific Software Architecture
Go4	Gang of Four (E. Gamma, ea.)
Go5	Gang of Five (F. Buschmann, ea.)
HIS	Hospital Information System
HW	Hardware
MA	Multiactivity
OO	Object Oriented
PAD	Process Activity Diagram
PAL	Process Activity Language
SW	Software
SWA	Software Architecture

As outlined by C. Anderson [63], computer evolution was characterized by large jumps followed by periods of predictable evolution, very much like the evolution of species. This holds true though for most of the engineering disciplines, although the time scale is much more compressed. Over the passed half-century, advance in computer technology has known four important development stages; the first started around 1950, with data processing based on the mainframe technology (actually this is when the term 'data processing' was introduced). The next big step came in the early sixties when the minicomputer arrived, gaining a lot of popularity in process control applications (now computer programmers start to give up their ties...). Then, twenty years later, the personal computer was introduced, causing virtually everybody on the planet to go computer frenzy. Finally, we are now stepping into the era of distributed computing which is especially focused on the Web. Oddly enough, the size of software (SW) programs evolved in inverse proportion to the size of the computer to run on (starting with small simple programs on huge computers and ending up with huge programs running on small machines).

Software design by and large started to receive a great deal of attention in the late sixties and early seventies. For instance, E. Dijkstra mentioned as early as 1968 that one has to consider the way SW is structured, not only how to compute the correct result. Based on the premise that design is an activity which is separate from implementation, with its own techniques, tools and notations, the research in the seventies was mainly directed towards the development of CASE tools, in order to accommodate development of large SW systems. After 1980 the focus of research moved away from SW design specifically towards integrating designs and the design process into the broader context of the software development process and its management. With the advance of our ability to describe and analyze SW systems, the concept of programming-in-the-large gains visibility; progress in the area of formal descriptive techniques

(through architecture description languages - ADLs) should also be mentioned along. Finally, the 1990's seem to be the decade of SW Architecture; the term 'architecture' being used, in contrast with 'design', to evoke the perspective of SW on a larger scale.

One could say that, in spite of nearly fifty years of existence, computers and in particular software development, are new (that is, have a short history) and immature. This is evident especially when compared to other 'established' engineering disciplines like civil architecture, mechanics or chemistry, to just mention a few. There is though a far more important aspect of novelty that SW development brings with it, and that is the sheer joy of seeing things work (or apparently work...) right on spot; somebody went even further to consider SW artifacts as today's industry Golems. The flexibility of computer programming is fascinating and unprecedented by any other crafts known to mankind. While buildings or mechanical systems are relatively hard to put in place (therefore requiring a lot of forethought before the actual implementation), computer programs can be readily developed and put to work with immediate results. In this context, F. Brooks was mentioning [12] that SW development is the only discipline where the designer and the user are the same person. On top of that, we think that a very important dimension SW development brought along is its dynamic perspective. While in most of the classic engineering disciplines a 'static' view could be enough to describe a system, in SW we should always consider the 'dynamic' perspective (which usually is hardly the same with the static description of the same system). It is not enough to put components together as a SW system; the interaction between components becomes important, especially if the system is event driven. Requirements and design concepts, which used to be applied only to embedded systems[1], find their place nowadays into general-purpose systems. This is a sure indication of the increasing expectations every-day SW has to meet.

The enthusiasm generated with the broad acceptance of computers put aside many of the good engineering practices well proven in other fields, in favor of undisciplined and hasty SW product development. Many of the design principles accepted in other engineering disciplines are only now penetrating SW Engineering. The lack of an engineering perspective so far is the main reason for our inability to date to handle complexity, and ultimately for today's lack of well architected systems.

---

[1] Examples span both functional (interaction with the environment, time constraints) and non-functional requirements (availability, reliability) as well as exploit of parallelism or system organization around some core functionality.

However, with the expansion of complex computer-based systems, significant efforts have been made already in order to somehow formalize ad-hoc development procedures. Beyond time to market and cost considerations, the need for better-formalized design procedures is getting more and more critical due to reasons like:

- exponential growth of expectations concerning today's SW systems (more and more, these systems are geared towards naïve users, therefore expected to also incorporate a lot of user knowledge)
- increasing complexity of today's SW systems, partly due to technical progress, partly as a consequence of the expectations growth
- reliability requirements, mostly connected to computer penetration into mission critical applications (communications, military, avionics, nuclear, ...)

The goal of any design is to translate a set of requirements into a product that satisfies specific needs (in other terms, map the requirements space onto the solution space); this has to be done through a controlled design process in order to keep results under control. The translation from a need to a product involves a number of major steps going from problem analysis/specification through solution analysis/specification and from here to the final product (as outlined later in the thesis). What is important for now is that most often there is a gap when stepping from requirements to solution specification. We ended up not once with products that proved not to be the right one despite working correctly (the famous GAO Report in the early 80's mentions that more than 45% of the contracted SW could not be used!). The architectural view of a system is able to provide the means for an early evaluation and reasoning on the solution, therefore helping to bridge this major gap.

In this context, the purpose of this work is three-fold:

- to introduce coordination by and large and Coordination Based Design in particular, as an efficient way to handle system complexity on the architectural level (HW and SW together)
- to define the need for architecture in SW and properly place software architecture (SWA) in the SW development context
- to establish basic terminology, ideas and trends in SWA

In more detail, the thesis has the following layout:

Chapter 1 advocates the need for architecture, looking at SW as a product. We define SWA, emphasizing the fact that most often multiple viewpoints are

needed in order to get a complete picture. Next, SWA is placed in the context of SW design, and contrasted to other engineering disciplines. Finally, the importance of SWA for reuse will be considered, followed by a look at Domain Specific SW Architectures.

Chapter 2 discusses architectural styles, going from their definition to various taxonomies presented in the literature. After highlighting some of the main sources for architectural styles, we present the use of commercial off-the-shelf (COTS) as one of the important trends in SW development today. Finally, a significant set of architectural decisions and their influences are briefly discussed.

Chapter 3 focuses on coordination, first from an interdisciplinary perspective, then as applied to computer-based systems. In the same line, we identify some of the important coordination patterns. The second part of the chapter elaborates on how coordination could be applied to computer-based systems, in order to focus finally on coordination and SWA.

Chapter 4 introduces CODE (Coordination Oriented Design Environment) as an integrated SW development methodology, with matching system architecture and modeling tools. After a brief overview of the driving ideas behind CODE, as well as an outline of the development steps, we go into more detail with the analysis of the architectural and design phases. The important ideas of CODE are put to use in the end through a case study, which is meant to illustrate that these concepts could be applied in a systematic way.

Chapter 5 goes back to SWA styles, reviewing their taxonomies from a behavioral perspective. Along with a newly introduced classification, SWA styles are contrasted to patterns.

To conclude, we maintain in this thesis that for developing well-engineered[2] SW one has to address at least three levels of design. Each in its own provides the means for making engineering decisions at a particular stage of SW development:

- *SW architecture*, which focuses on the user needs along with the high level design alternatives
- *design in the large* (simply called 'design') considers the next level of design, for each of the SW components

[2] By and large, beyond functional requirements, meeting basic non-functional requirements as a product (usability, reliability, efficiency, manufacturability, etc.).

- *design in the small* (also called 'implementation') concerned with the actual coding of the components

DeReemer and Kron introduced the last two notions as early as 1975 [21], SWA coming as a late addition of the 90's. It is interesting to mention that, within the patterns community, there are authors who have chosen to structure their collection of patterns along the same lines (that is: Architectural Patterns, Design Patterns and Idioms) [14].

We hope to establish in this thesis that the proposed Coordination Oriented Architecture for computer-based systems provides the following advantages:

- an architecture that provides the user with a cognitive/mental model of the system, which is easy to understand, thus facilitating immediate user feedback.
- an architecture that is matched to the behavioral specification needed in most systems today and to modern development principles.
- an architecture that partitions the system in a top-down fashion, thus providing for overall system perspective, ease of refinement and traceability.
- an architecture that is not stand-alone but part of a complete development environment, capable of investigation and decision making.
- an architecture that considers complete computer-based systems – HW and SW – to which one can apply system constraints.

Our perspective on software development places modularization and object orientation at a sub-architectural level and the various architectural styles at the architectural level; at the same time we consider behavior and coordination as a basis for architectural abstractions and overall system development. Although often mentioned throughout the thesis, the SW development process and methodologies, as well as requirement engineering are not within our scope. The same holds true for object orientation and patterns. Each of these fields offer vast research areas on their own, therefore the thesis limits itself around SWA and coordination only.

Chapter 1

THE NEED FOR ARCHITECTURE IN SOFTWARE

'(...) every individual act of building is a process in which space gets differentiated. It is not a process of addition, in which preformed parts are combined to create a whole, but a process of unfolding (...) in which the whole precedes the parts, and actually gives birth to them, by splitting.' Christopher Alexander [6]

These words belong to one of the most influential architects of this century in civil architecture and probably the most influential civil architect ever in SW engineering. Even though it is mainly the pattern community that recognizes Alexander's influence in the SW field, we think that many of his thoughts are worth sharing by the SWA discipline as well.

To start with, the Webster's definition of Architect/Architecture, is

Architect: *(Gr. Architekton - chief worker)*

1. *one skilled in the art of building; one who designs buildings, draws up plans, generally supervises the construction.*
2. *any similar designer*
3. *any builder or creator*

Architecture: *(from Gr. Architekton)*

1. *the art, profession or science of designing or constructing buildings*
2. *construction; structure; workmanship*
3. *a system or style of building having certain characteristics of structure, decoration, etc.*
4. *architectural productions, collectively*
5. *any framework, system, etc. (military architecture – the art of fortification; naval architecture – the art of building warships; marine architecture – shipbuilding of all kinds)*

As suggested by the definition above, other areas of engineering have later borrowed the term architect/architecture, originally connected to the art of building. Applied to engineering by and large, we think there is a need for architectural focus whenever the entity to be developed meets one or more of the following characteristics:

- the entity is too large and/or too complex to be understood and implemented by one or a small group of individuals; one needs both a blueprint (for everybody to follow) and someone to coordinate and supervise the 'integrity' of the work. As systems become more complex, there might be a need for multiple levels of abstraction (system, module, component...), hence multiple levels of design
- the users and implementers are not the same people; in this case the architectural level is the best way to communicate the key requirements motivating a specific design.
- the lifetime of the system is long, therefore the architecture serves as a repository for the basic building principles to be maintained throughout the evolution of a system
- the entity becomes a product, that is, it has to meet, beyond functionality and performance, requirements in terms of non-functional attributes like reliability, efficiency, maintainability, etc.
- the entity is to be manufactured, which means well defined components, tools, cost effective design methodology and process. The cost of producing the entity should also be factored in: custom-design vs. integrating off-the-shelf components or, more often, both.
- user needs cannot be determined entirely ahead of time; substantial modifications might be required in the future; this is one area where a flexible architecture can make a tremendous difference.

The arguments above suggest that the architect is mostly acting as a mediator/facilitator between the user and the design community, which leads to the following re-definition of its role:

Architect: one who interfaces between the users and manufacturers/implementers of a system. It helps the user to define the functionality of the required system and does the design in the large to provide manufacturers with a blueprint for implementing the system

Architecture: both the internal partitioning of the system into modules and definition of the user interfaces (application interfaces, rather

than graphical). As such, architecture has to primarily fulfill user needs along with allowing developers to make the appropriate implementation decisions.

Considering computer based systems, the literature brings forward various uses of the term 'architecture', both from a user's and an implementor's perspective. The three ones we consider as being important are:

1. *HW architecture*:
   Users are programmers, language developers, database designers, network administrators; they see the computer as a machine language executor and as an interface to the hardware
   Manufacturers are unit (processor, memory, I/O) designers, computer HW designers, etc.

2. *System architecture*:
   Users are application developers/programmers who see the computer system as a high-level language executor and provider of services
   Manufacturers are computer system designers who integrate various HW/SW components into systems.

3. *SW architecture*:
   Users are the general computer users - scientific, business, military, or entertainment - seeing the computer as a provider of various services.
   Manufacturers are application programmers writing specific or general-purpose applications

It is obvious from the above that the system view is relative to what one wants to consider, the term 'architecture' having therefore various interpretations. Further on we shall restrict our discussion to SWA and its influence on SW development in general. It should be noted that many times one cannot separate SWA from the overall architecture of the system; most often, in the broader context of computer system architecture, SW acts as a 'glue' between the constituent subsystems.

SW as a product

Like many other disciplines, throughout the years SW development has focused its attention on various levels of abstraction, according to the size and extent of the applications to work on. It has been already mentioned in the introduction that we started from paying attention to programming languages, went through

data structures and ended by paying more and more attention to the structure of SW at the code, module and finally application level. More and more sophisticated requirements brought in a tremendous increase in complexity of today's SW applications. It now appears that it is very difficult (if not impossible) to implement such an application without going through an additional level of abstraction above the SW module, one that allows for high level reasoning and decision making at the system level. The conversion from intuition to implementation involves understanding, and in this context there is an acute need for tools and methodologies to handle the complexity of today's SW systems. SWA provides the framework for this, by abstracting unnecessary details and, still close enough to requirements, actually be the first development step.

We maintain for now that the proper place for SWA in the overall design process is between establishing the requirements for the future product and the more detailed design phase, where decisions are being made on which algorithms and procedures to be used (part of the modularization taking place eventually). More details on the place of SWA in the design process will follow in the next sections.

In contrast to the early years when scientists used to write simple programs from one end to the other (being their implementers and users at the same time), today's SW development becomes significantly complex, mainly for the following two reasons:

- it is implemented by a development community as opposed to a single person; this raises issues related to maintaining the same perspective on the architecture of the product across the community and throughout the development stages
- it often relies on 'off the shelf' components, which raises a lot of integration problems mostly related to interfacing of the components [27]

As already mentioned, most of the early work in programming (and computers by and large) was developed by extremely highly skilled people who had a thorough understanding of the problem at hand. On top of the issues outlined before, many of today's programmers do not have the background and time to thoroughly understand the scope and requirements of the overall application. As a consequence, programming migrates towards what is called 'hacking'; it becomes a lot more undisciplined. Due to 'time to market' pressures, the end quality of SW is knowingly being reduced. Even well known authors like Ed

Yourdon promote these days the concept of 'good enough SW'[3]. Large SW development companies embraced this approach, and, what is more important, customers are willing to accept it (maybe because of a lack of choice). In this context, the role of SWA becomes even more important by establishing the important guidelines to develop a product in short time, close to those user needs that are critical. As Brooks was pointing out in [12], there is no harder step in building a SW system than deciding precisely what to build. Other than SWA, no part of the work has a bigger impact on the SW system if done wrong; no other part is more difficult to rectify.

In order to handle complexity, SWA must act as both an *analysis* and *synthesis* methodology. As an *analysis* tool SWA is seen as a framework for understanding the system components and their interrelationships (especially those attributes that are consistent over time and implementations). Architecture makes large systems intellectually tractable and allows for system level reasoning by abstracting away the implementation details. Nevertheless, the architectural definition must go beyond the structure of the system and include the overall operational properties as well. As a *synthesis* tool, SWA provides the means for modules to act together according to the requirements; it also uses specific domain knowledge to construct and maintain modules, systems and sub-systems in a predictable manner.

In its dual role, the ultimate goal of SWA is to promote quality attributes of the design product (like usability, reliability, maintainability, efficiency, etc) one abstraction level up from the design phase. We think that next to the need to handle complexity, the need for architecture mostly comes from satisfying non-functional requirements (detailed later in this chapter) on a large scale. Usually, when we talk about structuring the software, we first think of its functionality. Actually we should first consider its non-functional characteristics that define it as a product. Much of the early SW was developed to solve very specific problems in mathematics, physics, chemistry or engineering. Therefore, many of the early attempts to SW engineering have been focused on functionality, attempting to understand what was going on in these programs (design after implementation). As a result, suggested methodologies represented whatever was best suiting a particular application (eventually complemented by a number of analysis tools). The extended consequence of the above was in many ways that today's SW systems are not properly architected, engineered, designed. They include a lot of kludges to provide survival for the SW houses - finally spending a lot more time getting things to work and fixing bugs than actually

[3] To quote the author [62], '(...) what users really want is SW that's cheap enough, fast enough, feature-rich enough, and available soon enough – i.e. good enough.'; the key qualification though remains 'soon enough'.

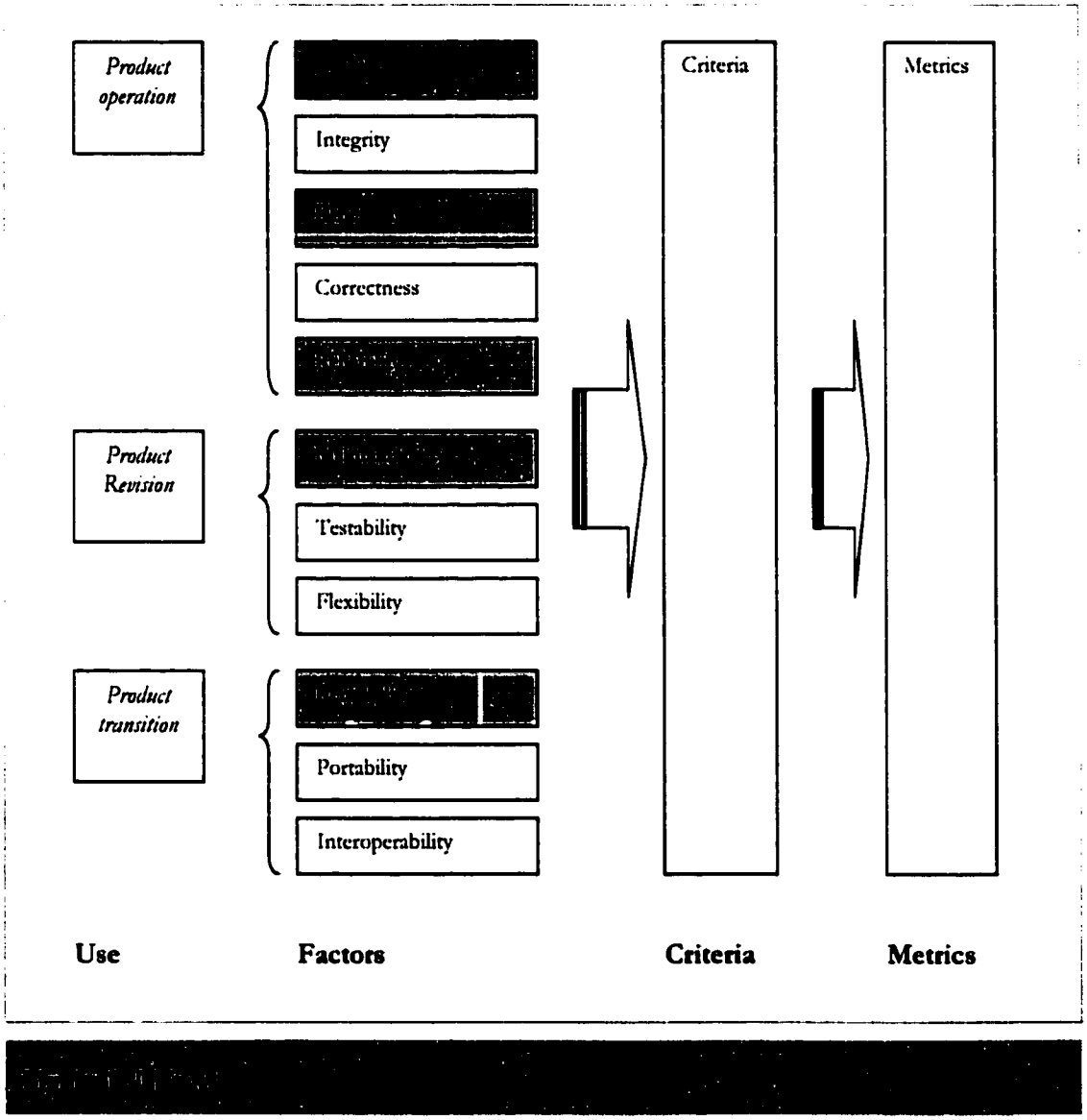
designing the system. That leads to bad products, not appreciated by their users, often not fulfilling the initial requirements, and prone to fast obsolescence. As M. Gentleman once pointed out in an informal meeting, discussing SW quality:

"Many people build and support programs in a way that seems adequate for their purposes. Companies in the SW business, when faced with technical difficulties have found solutions that allow them to get on with the job. However [...] SW developers and maintainers must become more effective."

Almost any system could be arbitrarily patched and quick-fixed in order to work; however, who wants to support the development costs and still deal with it after that? Obviously, SW has to be implemented as a product, hence it must be properly architected, and the design process should be such that (most of) the quality characteristics of a good product be implemented, followed, documented and to some extent measured. In today's SW, non-functional and functional characteristics are equally important.

There are a number of quality attributes that are typical for any product; since quality has many faces, the model to capture it encompasses a few interrelated characteristics. Amongst the early models developed, the McCall model seems to have gained a lot of popularity [22]. Using a decomposition approach, this model first establishes three *areas* where quality attributes can be used then a set of attributes in each area, called quality *factors*. In order to allow for quality measurements, further decomposition goes into lower level attributes called quality *criteria*, which are then associated to quality *metrics*. SW quality and its measurement are vast research areas, which developed a lot especially during this decade. It is not our intent to go into many details, this field being large enough in itself and out of the scope of this thesis. We will only look at what were considered to be the most important quality attributes.

People often talk about 'the four -ities', as key attributes of a well-engineered product. Without dwelling too much on them, these are *usability*, *reliability*, *maintainability* and *efficiency*. Keeping the same users/implementers perspective from before, we group these '-ities' into characteristics that are of interest mostly to the user (usability, reliability and efficiency) and characteristics of interest mostly to the developer (maintainability, reusability and, to a certain extent, efficiency). In addition to the McCall model, we prefer to look at efficiency not only from a user's perspective, but also from a developer's standpoint (see the reasons in the following paragraphs). As we mentioned before, SWA is intended as a vehicle to promote those characteristics to an abstraction level higher than design, moreover, allowing for decisions in any of the quality areas.



Usability and SWA

Usability, in a broad sense, describes how well the system performs its intended task. In a restricted way, it describes the 'ease of use', the capability of a system to provide the user with a mental model of the system behavior in order to

ensure the proper system-user interaction. This is an example where the restricted sense (the latter) became a lot more popular than the broad one (the former); we think usability is in a first place a measure of the degree to which we ended up with the expected (or right) product. It is not as important how intuitive and user-friendly this is, except for the case where easy user interaction is part of the driving requirements. For these reasons, we think usability is a user-related non-functional characteristic. User needs are generally specified in the early stages of design and can be reflected directly on a high-level design. Being so closely related to the requirement specification phase, SWA provides for convenient traceability of the main requirements on the first design models. As for the synthesis facilities, given the high level of granularity of the architectural specification, architecture could be closely fitted onto requirements from the very beginning, ensuring that the governing design decisions for later development will head towards the user needs.

Reliability and SWA

Reliability is concerned with the dynamic properties (behavioral aspects) of the system to be implemented. From a user's perspective in particular, it is concerned whether the system will be

- *complete*, that is, able to handle all combinations of events and system states consistently; behavior has to be as expected, repeatable and non-contradictory at any given time[4]
- *robust*, which means that faced with adverse or failure conditions, behavior will always be predictable (either recover or fail gracefully).

Reliability is definitely of importance to the user since it is related to the behavioral aspects of the final product. Between completeness and robustness, the latter seems to be most often associated to reliability. However, we believe that, for every SW system, the former is at least as important[5].

The contribution of SWA to completeness can be estimated from the synthesis and analysis perspectives previously mentioned. As a synthesis tool, SWA allows for the specification of dependencies and constraints to a desired level of generality, as such highlighting what is necessary and what is sufficient in the

[4] This particular attribute could turn extremely difficult to prove on an empirical basis; even extensive testing may not be enough to ensure a moderately complex system is complete.
[5] Completeness is sometimes connected to the requirement specifications in a sense that all the desired features are found in the final product.

initial design. By comparison with building architecture, SWA emphasizes what is 'load-bearing' and what is 'decoration' in the architectural specification. Further on, as an analysis tool, continuous consistency and dependency checking could be performed to determine whether requirements are satisfied: dependencies between requirements and design, between parts of the architecture or adherence to a particular architectural style. This type of activities, usually iterative and eventually involving formal methods, should lead to a complete design in the sense mentioned before.

Robustness could be considered an extension of completeness, in cases where the handling of failure scenarios is part of the initial specification; if the system is complete indeed, robustness becomes embedded in the expected behavior. Failing in a predictable way is a 'must have' requirement in the case of real-time systems. In the particular case of CBD, partitioning the system based on event responses at the architectural level allows for an early evaluation of robustness in the design cycle.

Maintainability and SWA

Maintainability is the capacity to allow for future modifications in order to either correct errors or introduce new features.

Maintainability is considered very important from a SW development perspective; most SW systems have to suffer a number of modifications (that is, 'maintenance') during their life cycle. Evolution[6] and customization[7] are two major factors cited as responsible for the high cost of SW development. Systems often evolve and are adapted to new uses; many times the price paid for this evolution is an increasing resistance to more changes or, worse, degradation of functionality. D. Perry and A. Wolf [50] talk about architectural *erosion* – due to violations of the initial architecture - and architectural *drift* - due to insensitivity to the initial architecture – as unwanted side effects of evolution. Usually the two condition each other. Parnas [49] discusses at large a lot more causes for the obsolescence (or 'aging') of SW due to evolution.

When it comes to maintainability, SWA can make a difference from the start through its analysis capabilities. For instance, one could develop a set of candidate architectures and then estimate on each of them the impact of changes that are

[6] New features are added as users find new needs.
[7] Many times systems are designed to provide for a larger user group and may have to be tailored to fit requirements of specific users.

most likely to occur. The architecture allowing for most (successive) changes is the better; [37] goes into more details on this type of scenario evaluation. In general, a good design requires careful consideration of possible changes; one way to improve system maintainability is separation of concerns, in a sense that will be discussed later.

Efficiency and SWA

Efficiency is seen as the degree to which the system makes use of its resources (how well, in other words) compared to the expectations. It is a relative measure, mostly of interest from a user's perspective.

At least some basic efficiency limits should be part of the initial requirements (for example, prevent a simple program from taking up all the system memory or hog the CPU). Efficiency also has to do with the implementers, given the fact that managing the limited resources available in the context of satisfying the functional needs is an important part of the engineering solution. Still from the implementation standpoint, efficiency may have a direct impact on performance. It is important to take a look at efficiency before getting involved with more detailed design, since the major parameters established at the architectural level tend to remain the same throughout implementation and even future SW versions of the product.

As a synthesis tool, SWA allows for high level decisions concerning the placement and accessibility of data structures, libraries or other system resources. Domain specific knowledge could be used at this early stage in order to customize system behavior to the application. On the other hand, as an analysis tool, SWA offers the high level perspective necessary for identifying major design flaws like system bottlenecks or deadlocks. By explicitly specifying the interfaces of the system along with their interactions, dependencies between modules are tractable more easily. If desired, running models could be developed at this level in order to establish efficiency benchmarks to check against later in the development process.

With today's low cost of HW resources, time tends to remains the only critical resource to be managed; nevertheless, some overall system efficiency guidelines must be provided[8].

[8] For instance, some of the word processors today take almost ten times more resources than ten years ago, for a rather little increase in functionality.

More 'ities' are associated in the literature to the SW product (some of them slightly overlapping or deriving from the ones we just described), such as security, installability, portability or understandability. For now, we choose to stay with the ones previously mentioned. As a matter of fact, they are connected to about 70% of the quality criteria in the McCall model. Although not necessarily terminated in '-ity' (one could say 'reusability' though), we think the potential for reuse is another highly desirable non-functional aspect of the SW product; we will look at it in the following section.

Reuse and SWA

Very often, the inability of SW developers to achieve low costs, high productivity and consistent quality has been attributed to a lack of SW reuse. While a universal SW reuse solution may seem unrealistic, significant improvements could be made focusing on well-defined areas of knowledge or activity (domains). SWA supports domain specific reuse by serving as a framework for understanding families of systems; the emergence of Domain Specific Software Architectures (discussed in more detail in the coming sections) is one significant step in this direction. The effectiveness of architectures as frameworks for reuse could be evaluated (again) through analogies with other established disciplines such as civil, mechanical or chemical engineering. One of the key attributes of a mature engineering discipline is the routine of using existing solutions for the development of new systems. Usually, innovative designs are first prototyped and implemented in several systems; if successful, universal acceptance eventually occurs. Successful solutions are then included in handbooks, technical publications, corporate standards, etc.; this leads to a high level of dissemination and design reuse. Current efforts in identifying and applying architectural techniques to SW could be seen as an attempt to reuse successful high-level solutions and a shift of SW development towards a mature engineering discipline.

From what we have discussed so far, two major requirements have fueled the development of an organized approach to SW design:

- the requirement to manage SW complexity in order to make large SW systems intellectually tractable
- the need for reuse (reuse of good designs/principles and where possible, code reuse as well)

It is worth mentioning that the need for reuse, in particular, fostered the occurrence of design patterns and the ongoing interest in them.

Although we named reuse at the end of our quality attributes list, it still is one of the major drives for SWA. As systems grow in complexity, in the context of shrinking budgets and schedules, what is called 'architecture centric reuse' becomes more and more important (so far this term has been mostly materialized in the form of frameworks).

Architecture is the most significant element for reengineering in a specific domain. The architecture is the key element in systematic rather than opportunistic reuse. Systematic reuse relies on a design made to match a wider category of users; customization then takes place, following pre-determined development paths (frameworks are a good example in this case). The final product could then closely fit the targeted application, with virtually no impact on the user. On the other hand, opportunistic reuse involves the use of assets (legacy code) in an ad hoc manner. In the case of opportunistic reuse, the burden that is left on the user is to understand the behavior of the software, the context in which it was developed, and how to fit it into the new application.

Most times, systems are not developed from scratch; they are evolved from (previous) other systems. Without architecture, the relationship between different sections of the existing code is hard to identify and therefore the assets are used individually rather than as a coordinated collection. The scope and benefits of reuse become smaller this way. In such a context, the architectural perspective could provide the relationships between all these features. Hence the advantages of architecture to the reengineering team include a larger scope for reuse and a coordinated guide to using the reusable code. The reengineering team does not have to determine how to put the code together since the architecture already provides the blueprint. However, the reengineering team must determine how to prepare the application for including the reusable assets: it must determine the interfaces between the application and the architecture as well as the reusable assets in the domain. In a phrase, the architecture allows the reengineering team to concentrate on interaction and interfacing issues instead of asset functionality.

The possibilities for reuse are greatest where specifications are least constrained, that is at the architectural level. Since architectures capture system constraints that are relatively constant in time, the architecture remains consistent between different versions of the system and could be reused for different applications within the same domain. In contrast, addressing relevant issues late in the development process (at the implementation phase) consequently reduces future

reuse opportunities because of relying on the poor reuse support afforded by particular programming modules.

Another side of architectural reuse is related to the design and coding of standardized interfaces and protocols, as well as the packaging of functionality on a high level of abstraction (to be further refined). Standardization is of particular interest since it facilitates a smooth integration of third party components with the ones designed in-house.

Let us mention in the end that, according to the specific application area the product is targeted at, some of the quality attributes could gain more emphasis over others:

- *reliability* (with the two aspects it entails) and efficiency are crucial for real time SW systems performing mission-critical tasks in adverse environments and with scarce resources.
- *maintainability* should be a common attribute to any piece of SW with a long life expectancy (that is, prone to subsequent maintenance work or upgrades).
- *usability* has a lot of visibility in the case of user driven (interactive) systems (nevertheless is a must for any successful product – sure proof that we ended up with the right one).
- *reuse* potential is very much sought after in every area of SW, given continuously shrinking development budgets and time to market constraints.

To various extents though, all the quality characteristics described so far are more than desirable in a SW product; a better (software) product is the major goal of SW design in general, architecture in particular.

The place of SWA in the design process

As mentioned in the introduction, in the early days, programs were small, with a restricted area of applicability, therefore easy to comprehend and implement. Furthermore, the programmers were scientists, who had a very thorough understanding of their application area. Because of this, there was no need for multiple levels of abstraction when writing a program. Only in the mid-seventies the notions of programming-in-the-large and programming-in-the-small were introduced by DeRemer&Kron [21] as a first response to the growing demands put on SW development.

Today there is a need for a higher abstraction level so as to consider SW from a system perspective. SWA comes as a normal evolutionary stage in our way of

looking at SW. Through a bottom-up approach, we have now left behind the concerns regarding algorithms and programming languages, in order to focus on the system perspective of SW. D. Perry and A. Wolf, pioneers in the study of SW development in general (and SWA in particular), recognized four phases in the SW design process, SWA being one of them [50]. In their view, starting from the requirement specification and down to the final implementation, SW design has four main stages:

- *requirement analysis* phase is concerned with finding the characteristics of the information and processing needed by the user. Along with the desired requirements one has to consider constraints related to resources, time or availability of information.
- *architecture*: is concerned with the gross partitioning of work, selection of architectural elements, their interactions and related constraints in order to provide a framework for design.
- *design* is concerned with the modularization and detailed interfaces of the design elements, algorithms, procedures, data types needed to support the architecture and meet the requirements.
- *implementation* is concerned with the representation of algorithms and data types that satisfy the design, architecture and requirements

We very much share the same view as the one above, with the exception of defining the interfaces. In our perspective most of the interfaces between SW entities should be defined in the architectural step, as being one of the important attributes in the high level modeling of the system. Once the interfaces decided upon, they could be refined at the design stage. Another aspect we want to emphasize is that before the requirement analysis phase (in the sense defined before), there is the crucial step which decides whether it makes sense to develop a SW product at all. In other words, one has to decide whether the given problem can be treated computationally, which usually involves

- a system to be modeled
- a system behavior to be expressed as a solution (and eventually provide a formal model of the solution)
- a translation of the above into a set of computations, executable on a computer (therefore the need to develop algorithms – or translate the solution into existent algorithms – and present this to the computer in an executable form: write a program).

This step may be part of the feasibility study, where one has to realize how a system can be modeled, a solution procedure developed and translated into an

algorithm and a program. The fact that the architectural step is considered the first look at the overall implementation of the system (in the context of the SW design process), after having established the requirements, fits with our user-related definition of Architect/SWA in the beginning of this chapter[9].

Depending on the system, the architect works with a user, management or sales group to undertake a requirement analysis and define requirements for a product or family of products. It is the architect's role to validate requirement specifications for consistency, completeness or applicability. In addition, due to his knowledge of the field, he might also suggest additional functionality not obvious to the user. Starting with the initial specifications, the architect does the design in the large by partitioning the system into implementable modules. At this point he may also consider general implementation aspects, such as production run size, system lifetime, maintainability and other constraints. This is done keeping in mind available technology, resources, etc. As part of this step, the architect has to validate the fact that the design in the large meets the functional requirements and system constraints. To allow for maintenance and later modifications, is imperative that all decisions be documented and preserved.

From the above outline, it becomes obvious that the architect has to balance many contradictory requirements (needs), therefore 'creativity' remains one of the main ingredients in architecting a complex system.

Points of view on SWA

As opposed to the accepted definition of Architecture as it relates to buildings (mentioned in the beginning of this chapter), there is no standard one to date for the term 'architecture' as applied to SW. In searching the literature, we found no shortage of definitions. For not having one commonly accepted, here is a brief outline of the most influential views with respect to SWA [64]:

Perry & Wolf (credited as pioneers in this field) tried to establish a model for SWA and came up with the following definition in 1992 [50]:

'a model of SWA (...) consists of three components: Elements are either processing, data or connecting elements. Form is defined in terms of the properties of, and relationships among the

[9] It seems that, applied to SW, the term 'architecture' was maintained probably as a tribute to the ancient discipline of buildings; we think that 'designer-in-the-large' or 'system designer' be more appropriate terms than 'architect' would.

elements, that is the constraints on the elements. The rationale provides the underlying basis (...) in terms of systems constraints, which most often derive from system requirements.

Architecture = {Elements, Form, Rationale}

In this context, SWA is '*a set of architectural (or, if you want, design) elements that have a particular form.*'

Along the same lines, but much later, Barry Boehm and his students at the USC Center for SW Engineering write that [64]

'*A SW system architecture comprises*
- a collection of SW and system components, connections and constraints
- a collection of system stakeholders' need statements
- a rationale that demonstrates that the components, connections and constraints define a system that, if implemented, would satisfy the collection of system stakeholders' need statements'

With all the merit it has for being the first focused attempt to define SWA, this definition is static in two ways:

- first, it doesn't make any mention of the dynamic behavior of a system at run time, as reflected on the architectural level.
- second, it leaves aside the basic ideas that stay behind a high level design, governing its evolution in time.

Some of these attributes are found in the definition advanced by Garlan & Shaw in 1993, although this is still centered on the structural issues. They maintain that SWA goes
'(...) beyond the algorithms and data structures of the computation; designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and control structure, protocols for communication, synchronization and data access, assignment of functionality to design elements, (...) scaling and performance and selection among design alternatives.'

At the other end, here is one other definition focused almost entirely on the dynamics of the system, advanced by Hayes-Roth in a paper for the ARPA-DSSA program:

'(SWA is) an abstract system specification consisting primarily of functional components described in terms of their behaviors, interfaces and component-component interactions.'

Aware of the fact that SWA entails too many aspects to be captured in one or two phrases, and that beyond these, things start to get confusing, authors started to look at SWA from different perspectives. It turns out that there are actually a number of dimensions in the SW development process where architecture plays its distinct role. These different perspectives were called 'views'[10] and Bass et al. come with the following definition in 1994:

'(...) the architectural design of a system can be described from at least three perspectives: functional partitioning of its domain of interest, its structure, and the allocation of domain function to that structure'

Further on, Soni, Nord and Hofmeister of Siemens Corporate Research based their approach on studies of several SW development projects. According to them, SWA is four-fold:

- 'the *conceptual* architecture describes the system in terms of its major design elements and the relationships amongst them.
- the *module* interconnection architecture encompasses two orthogonal structures: functional decomposition and layers.
- the *execution* architecture describes *the dynamic structure* of a system
- the *code* architecture describes how the source code, binaries and libraries are organized in the development environment'

This is indeed a good example of how views can be introduced (although their standpoints could always be debated). Nevertheless, the last bullet takes us far away from the architectural perspective for offering an over-simplified view of the system.

In 1995 Shaw brings some clarification and at the same time puts some order in the multitude of terms used to define SWA by classifying the views on SWA as follows:

- *structural* models showing SWA as composed of components, connections among the components, plus some other aspects including
  - configurations (captured in architectural patterns or styles)
  - constraints (captured in semantics)
  - analyses (captured by properties)
  - rationale (captured in requirements/stakeholders' needs)

[10] A separate section will be devoted to SWA views in the following pages.

- *framework* models, similar to the structural view but with emphasis on a specific predetermined structure of the system (as opposed to focusing on the composition of the system)
- *dynamic* models with emphasis on the behavior of the system at run time (eventual reconfiguration, enabling or disabling certain communication or interaction channels, etc.)
- *process* models with focus on the process to be followed in order to build the specific architecture

This is a serious attempt to capture the many aspects involved in SWA; the only thing we would argue with is the introduction of the framework models, which we consider to be a delivery option for a SW product, rather than an architectural view.

One recent definition worth mentioning belongs to [14]; the authors look at SWA as

'a description of the subsystems and components of a SW system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and nonfunctional properties of a SW system (...).'

The definition introduced below is a sample of what we consider a rather concise, though comprehensive one and was put forward by Garlan & Perry in 1995:

'SWA is the structure of components of a SW system together with their interrelationships, principles and guidelines governing their evolution over time.'

As already mentioned, structural issues in SWA refer to global gross organization and control structures, assignment of functionality to design elements, physical distribution and various other alternatives around them. The important thing this definition brings forward (and on top of the others) is the role of SWA as a repository of the governing design concepts to guide the system maintenance and evolution over time.

It is interesting to note[11] that many SW developers see the architect as somebody who

- is a generalist, not a specialist

[11] This position has been emphasized in various informal discussions with SW developers from the military, government or the high-tech industry.

- one that can advise both managers and specialists
- by and large, one that can sell ideas and concepts

The above definitions are not really complete in the sense that SWA cannot be considered as a stand-alone. Most often one has to consider the system level architecture where SW, in spite of playing a significant role, is one element of a larger system; the surrounding non-SW elements come as constraints and add new dimensions to SW interactions. Architecture is in this case a high level description of the system serving as a blue print for the system as it evolves, and allowing sub-system designers to understand the big picture of the system without having to go in any specific details.

Architectural views

Many authors insist on the fact that architecture is more than a high level design on how components might be used. Look at the boxes and arrows of an architecture diagram for instance and many times it becomes clear that its authors were trying to represent on the same drawing more than it was practical. The diagram could represent chunks of source code, or logical groupings of functionality or just physical computers. It usually tries to represent a bit of everything. This makes us consider architecture as including complementary viewpoints which offer a more in depth perspective on the problem at hand. Any system has a set of architectural viewpoints that have to be developed and resolved; these viewpoints are the ones to later drive design decisions. The question is now which viewpoints are worth considering; things are not quite settled in this area, different authors taking different approaches.

First of all, a view is considered as *'a partial aspect (of the SWA) that shows specific properties of a SW system'* [14].

By and large, one could say that any system has an 'architectural view' (in light of our previous discussions corresponding to the user's view and presenting the information needed to properly understand and use the system). A second perspective is the 'organizational view' (providing a high level functional description on how things work and interact). The latter also provides important requirement specifications for the implementers. Perry&Wolf, in [50] mention the multiple views of SWA as something inherited from the building architecture; their opinion is that SWA has three important views:

- the *process view*, mainly given by the data flow through the system, as well as the connection between processing elements
- the *data view*, with emphasis on the processing flow and less the connecting elements
- the *connections*, as mechanisms for moving data around

The three views are all overlapping and interdependent (especially the first two which are intimately intertwined, given that the state of the data relies on the process and the other way around). It is easy to see the bias towards a data oriented approach to SWA, influenced maybe by the OO advance in the early 90's.

Things could be filtered even further down; a largely accepted standpoint is the one taken by P. Kruchten in [39]. This time the approach taken is to describe SWA using five concurrent views (4+1 actually); four of them are 'static' in a sense that they are more or less self-contained, the fifth being 'dynamic' and acting as a glue between the four. Thus we first have:

- the *logical view* merely describes the logical grouping of functionality
- the *process view* describes concurrency and synchronization aspects.
- the *physical view* describes the mapping of the software onto hardware.
- the *development view* describes the allocation of software in its development environment (teams or individual developers, etc.).

Any of these views is not independent. Elements of one view are connected to elements in another view following certain design rules. Relevant instances of use cases, called *scenarios*, incorporate the connections between the views and are meant to prove that the four views work together as intended. The set of scenarios make up the fifth view (somehow redundant with the other ones) but playing two important roles:

- act as a driver to help discover the architectural elements during the design phase
- validate and illustrate the architectural design both on paper and in practice

As indicated by its authors, the entire process is iterative. It is out of our scope to further detail the multiple-view approach to SWA; nevertheless, in our opinion (not necessarily supported in the paper), the main contribution of this 4+1 perspective is that it tries to explicitly capture the behavioral aspect of SWA.

Domain Specific Software Architectures

The need for reuse on the architectural level lead to the emergence of Domain Specific Software Architectures (DSSA). This name is taken from a project put together at Carnegie Mellon for the Department of Defense. It first started with military applications (Avionics, Guidance and Control, etc.) but was also used as a starting point for other projects in the industry, mostly GUI design and telecommunications. As stated by its initiators[12], DSSA is a process and infrastructure that supports the development of

1. a set of *Reference Requirements* defined by a set of 'common' problems or functions that applications in that domain can solve/do (hence the term 'application domain')
2. a *Domain Model* characterized by a common taxonomy for describing problems or issues that applications in a particular domain typically address
3. a *Reference Architecture* (for a family of applications within a particular problem domain) as an architectural description for a family of applications describing functional components, connections or protocols

In general, the Reference Architecture consists of generic or abstract components that partially specify a system and which are replaced by real components when the architecture is actually instantiated. For example, the OO implementation of a framework is often a set of cooperating classes that could be composed or sub-classed in order to achieve the actual implementation. In their turn, the Reference Requirements are a set of standardized requirements for an entire family of applications.

The expressed goal of a DSSA is to support the generation of applications within a particular domain (also known as a product-line). Moreover, any architecture of existing systems could be generalized and placed into a DSSA infrastructure so as to ease maintenance efforts and reduce the cost of follow on activities. The most popular example of DSSA are frameworks. To paraphrase the definition advanced in [14], frameworks are a semi-finished SW (sub-) systems intended to be instantiated. They define the general architecture for a family of subsystems and provide the basic building blocks to create them. In the spirit of what was outlined already on DSSA, a framework captures the major design decisions that are common to an application domain; they emphasize design reuse over code reuse, going as high as the architectural level.

---

[12] The Domain-Specific Software Architecture Program, Special Report CMU/SEI-92-SR9.

Some of the main advantages offered by DSSA are very well instantiated by frameworks:

- frameworks improve key non-functional SW quality factors like reuse, modularity, scaleability
- they avoid traps and pitfalls usually learned only by experience; they capture expert knowledge and design tradeoffs
- communicate architectural knowledge among developers; codify good design (abstract the design process) therefore help manage SW complexity, even though often on a lower abstraction level
- frameworks help to structure SW systems into subsystems
- they are a good way of documenting SWA, therefore avoiding reverse engineering

SWA and other engineering disciplines

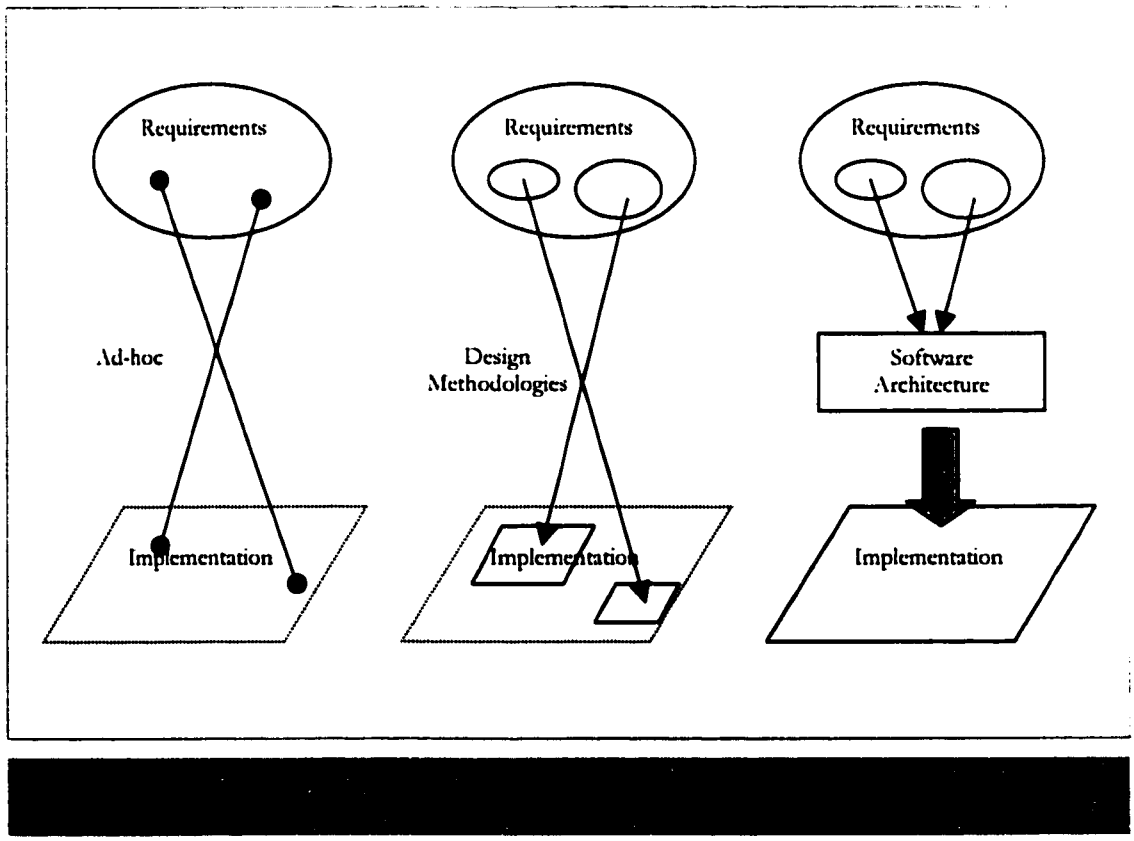
With respect to its place along with other SW related disciplines, SWA stays in the larger context of SW Engineering. Although both disciplines are concerned with bridging the gap between requirements and implementation, there are significant differences between them. SWA differentiates itself in the context of SW Engineering by and large, for the following reasons:

- the concern about algorithms and data structures is left aside in favor of the overall organization of large systems and their global control structure.
- the concern on implementation details like language, source code or definition sites is replaced by the configuration of the systems in terms of components and connectors along with the associated behavior
- the investigation on design methods is replaced by the one on architecture.

With respect to design methods though, SWA and design methods complement each other. SWA could probably develop on its own using whatever handcrafted techniques available; however, design methods come into play and improve the situation by providing a path (out of many possible ones) between specific system requirements and a specific system implementation. In its turn, SWA is concerned with the tradeoffs between the solutions chosen, acting like a filter for the most suitable one. Design methods and SWA complement each other for each design method may have its preferred architectural style; on the other hand, architectural styles may lead to new design methods that properly exploit them.

In order to contrast the perspective of well-established engineering disciplines to SW engineering, analogies have often been made between architectures in SW and those in other engineering fields. Most analogies have pointed towards building architecture; for instance, some commonalties between SW engineering and building architecture would include:

- architectural styles (Gothic, Victorian, Roman in building architecture vs. flow, layered, distributed, etc. in SW)
- notations and representations (blueprints, models, perspective drawings in civil engineering vs. structural diagrams, behavioral and informational views in SWA)
- standards/practices (building codes and inspections vs. Interface/protocol standards and code reviews)



Similar analogies could be made with chemical or mechanical engineering. Chemical processes can be seen as an equivalent of (some of) the SWA; due to the dynamic nature of chemical processes, some analogies here may hold for SWA but come in contrast with the static nature of building architecture. Just like a SW system, a chemical process has inputs (often of a random variation), processing (continuous in general) and outputs. Usually chemical processes have rich topologies and component types. In a point form, here are some of the analogies between SWA and 'chemical' architecture:

- *components* (chemical operation units - heat exchanger for instance - vs. SWA components - data base for example)
- *connectors* (chemical pipes and conveyors vs. procedure calls or SW pipes)
- architectural *styles* (chemical operations - batch, continuous - and topology vs. SWA styles - flow, call, etc.)
- *constraints* (precedence of elements in chemical reactions vs. precedence of data or conditions in SW)
- *notations* (flow charts and formulas vs. diagrams, ADLs)

Some of these analogies may seem somewhat limited or far-fetched. This is partly due to the fact that in other disciplines the structure is more closely related to the physical reality. SW has a conceptual nature by excellence; in SW there is no material associated to the product, we design things that we cannot see and we actually test effects, not the entity. Nevertheless, it would be useful for the SW engineer to estimate to what degree the initial requirement model is transferred into the final product (now that we established a number of ideas on product quality concepts). We step here into the complex area of requirement traceability, which has not been settled yet.

There are two criteria already in place to evaluate the partitioning of a SW system: *coupling* and *cohesion*. Both are concepts originally introduced as part of the structured design approach [58][13]. Coupling focuses on inter-module aspects and measures the degree of association between modules, established through a connection (higher coupling complicates things – it is loose coupling that helps reduce complexity). Cohesion measures the degree of connectivity between elements of a single module, emphasizing intra-module characteristics. The most desirable here would be 'functional' cohesion (elements working together to achieve a common goal) as opposed to 'coincidental' cohesion where unrelated abstractions are found into the same module.

A third specific attribute is *information hiding*, which consists in the concealing of details of component implementation from their clients in order to better

handle complexity and minimize coupling between components. Any details of a component that clients not need to know in order to use it properly, should be hidden (for instance specific data is made available only to selected modules or paths of code). Information hiding usually helps both modularity and simplicity.

There are (at least) two aspects that could be presented for measurement:

- a set of design attributes related to the properties listed above[13]
- a way to extract information about these attributes from the product documents available

With respect to the design attributes, one could mention *simplicity* (or its converse, complexity) as a way to meet objectives with no additional embellishments. Simplicity directly relates to maintainability/testability in the way that simple systems are easier to handle in terms of implementation, testing and subsequent modifications. Simplicity comes as a direct consequence of the attempt to handle complexity, which in turn lead to the emergence of SWA itself. Along the same lines, simplicity has a direct and benefic impact on the completeness part of reliability and, possibly, efficiency. Another possible attribute could be *modularity*, which starts with decomposing a given problem into smaller components. In order to promote a modular structure, one has to establish an approach based on separation of concerns. Functionality has to be grouped together in such way that module interdependence be minimized. This approach is convergent with the practice that has been established in the field of engineering by and large, to handle complexity by first partitioning the requirements space and then the solution space, according to the requirement partition.

Concluding remarks

With the increasing complexity and size of SW systems, the computational algorithms and data structures constitute no longer the main problems in SW development. Their place is taken by a new set of problems pointing to the overall organization of the system, along with the interactions between its constituent elements - that is the SW architecture.

[13] We restrict the set of design attributes to just a few, considered of interest in the context of the thesis; the literature mentions a lot more of them.

There still is a large context in which the term SW Architecture is used [64]. We have already discussed points of view on this topic, however, even from a slightly different perspective, some of the most significant formulations are:

- a synonym for design in the large
- prototyping or early implementation
- high level view of a system
- explanation on how a particular technology can be incorporated into a system

One could also say that the architectural level of system design represents the overall structure of a system as a composition of interacting parts. Other than the structure itself, this is the place for system level reasoning in terms of functionality assignment to architectural elements, their interaction protocols, processing rates, end to end capacities, overall performance, portability and scalability. Architectural design itself is concerned with composing systems from components and the interactions between these components. Such compositions provide an abstract view of the system so that the designer could perform system level analysis and reason about system level constraints. Architectural abstractions also let a designer associate multiple interfaces with components and express topological and semantically based constraints over a design.

From the discussions above is obvious that the role of the architect is more than the one of a supervisor. The architect is more of a generalist, who also gets involved with management aspects like cost analysis or design process. It is deemed to be easier to solve a specific problem but harder to find a solution to encompass multiple uses and versions of the product - therefore the need for a design process and especially an architectural perspective.

Some of the major benefits expected from the emerging discipline of SWA are:

- to act as a design framework trustworthy to satisfy requirements
- a technical basis for management of the SW development process
- a basis for dependency and consistency analysis
- a blueprint for design
- as much as possible, basis for reuse

It is our belief that SWA is targeted towards improving especially the non-functional characteristics of the SW product; once the architectural step has been successfully addressed, the design step can focus on meeting the desired

functional properties. A lot of progress has been made in recent years to promote SWA to common practice in SW engineering; however, ad-hoc still plays a major role when it comes to architectural design, leading to SW structures that are handcrafted, informal, hardly analyzable and inconsistent. On the other hand, SWA itself strives to become a discipline based on solid engineering principles. As a consequence, SWA receives a great deal of attention today, further expansion of interest being expected.

To conclude, we think that, in the case of complex systems, SWA offers the perspective to achieve basic requirements of proper product engineering:

- Improved productivity of development and maintenance
- Better predictability of production process
- Reduced time to market
- Better quality
- Larger market share through longer life and wider applicability

Architecture seems to be the most indicated vehicle towards improving both functional and non-functional properties of a system, as a whole. Well architected systems allow for a wider scope than originally asked in order to satisfy additional needs of present users (and attract others), with an emphasis these days to make extensive use of existing (off-the-shelf) components.

Chapter 2

SWA STYLES

The relatively short history of SW development has promoted a number of system organization concepts (such as client/server, data centered, object-oriented, frameworks, COTS, etc.), most of them evolved from ad-hoc solutions to applications in various areas. Only a few SWA models have been first developed and documented then advanced to the SW community for future use; the most popular example here are SW applications for telecommunications based on the OSI model. Given the exponential growth of SW complexity - which itself lead to the need for SWA - it becomes important to properly define and store system organizations that proved to be useful. Patterns are an important attempt to do it on the design level; architectural styles are part of the effort to achieve the same at the SWA level.

Just like in the case of SWA, one could start investigating SWA styles by first looking at the general use of the word 'style'. The same Webster Dictionary mentions a lot of meanings under this heading; we have chosen only the ones that are closest to the context of this thesis:

Style *[... from Latin, 'stilus'...]:*

1. *'[...] specific or characteristic manner of expression, execution, construction or design in any art, period, work, employment, etc.; as the Byzantine style, modern style.'*
2. *'the way in which anything is done; manner'*
3. *'sort; kind; variety; type'*

Loosely stated, a style is defined by a set of common characteristics in describing a group of entities, which makes them different from others.

In this chapter, we intend to first mention various styles and taxonomies as discussed by a number of authors; further on we will introduce our own unitary view on SWA styles along with considerations on the way to classify them.

It is largely acknowledged that, for any established engineering field, useful system organizations need to be stored and disseminated. In this end, with respect to the high-level organization of systems, two approaches are possible:

- an *observational* (bottom-up/passive) approach which optimizes and refines architectural solutions empirically, based on their continuous use by the SW community (a trial-by-error/post-facto approach)
- a *requirement-driven* (top-down) approach where patterns of architecture are first developed conceptually, then used in various applications in order to instantiate and ultimately validate their design

To date, the first approach has been the one of choice. Without highlighting the context that lead to a particular system organization, we think this approach doesn't necessarily provide for a proper match between the application and the solution to it. Our approach, which we called 'requirement-driven', sets out to bridge this gap by first identifying specific sets of problems, then suggesting SWA styles as solutions to them. This comes close to the [25] approach of organizing patterns on the design level, by use of the context/problem/solution template[14]. It is worth to note that, in the case of SWA, when one mentions the word 'style', usually ends up by also defining a taxonomy. Taxonomies (by the Webster: 'the science of classification; the laws and principles covering classifying objects') are generated by looking at existing implementations and try to classify them; which is all right, but only to some extent. Just as in the case of patterns, the development of SWA styles was based on an inventory of recurring SW development techniques used throughout the years. Although this is a good start in every field of research, it is highly desirable that we come up with newer solutions that are first designed and then tried out. The academic use of taxonomies in SWA could be more restricted in favor of the ones based on actual types of requirements. Still being far from having a commonly accepted classification of architectural paradigms, we are at least able to identify and discuss a number of styles that currently shape the basic content of SWA.

The use of SWA styles

As part of introducing SWA as a mandatory step in the overall SW design process, [50] also introduced the notion of architectural style, again, by analogy with the building architecture. The authors see architectural styles as being important from both a descriptive and prescriptive perspective; [57] shares

[14] C. Alexander was actually the first one to suggest this criteria, for building architecture patterns, first in [5], then in [6].

basically the same view. Along with some of the points made by the latter, we could summarize the two perspectives as follows:

- *descriptively*, architectural styles provide a codification of formal arrangements and design elements (such as components and connectors).
- *prescriptively*, styles provide restrictions on the descriptive part, that is some limits on the repertory of design elements and the way to arrange them.

The intent of SWA styles is to act as a set of guidelines on the architecture of SW. Styles only capture some of the important decisions on the architectural elements, at the same time emphasizing relevant constraints on the architectural elements and their relationships. At the same time and to various extents, the level of constraint can be left up to the architect.

Some authors maintain that any SWA style needs to provide a specialized design language for a particular class of systems; in addition, it is highly desirable that every style allow for a number of sanity checks on successive versions of the design. In this end, one can expect for instance

- a vocabulary of architectural design elements (component/connector types)
- design rules or constraints that determine which compositions of these elements are allowed within one style or another
- design analyses that could be performed on a system built along a particular style
- semantic interpretation to decide whether, once the design rules satisfied, the composition has a well-defined meaning according to the initial requirements

The four bullets above seem to lean towards formalizing architectural styles, especially in order to promote constraints characteristic to a specific structure. Along these lines, significant attempts have been made in this area, first by developing Architecture Description Languages (ADLs)[15] and, based on the above, development and simulation environments for architectural design.

[15] The study on ADLs has been initiated jointly by the Department of Defense and several academic institutes in an effort to address architecture representation issues. ADLs are intended to assist the development of new systems, mainly in the areas of real-time and distributed applications. Some of most popular ADLs are Micro-Rapide, UNAS, Meta-H and Control-H, etc.

There is another school of thought though (M. Jackson [35], M. Gentleman[16], ea.) which tries, at least for SWA, to keep things to an intuitive level, even at the expense of losing some of the formalism. The reasoning behind this approach is to keep complex systems intellectually tractable, since this is actually one of the main reasons to promote SWA. We also tend to share this latter view.

Regardless of the formalism level, one would be able to connect the use of architectural styles to a number of significant benefits, in close relation with the need for SWA itself:

- styles promote design reuse by abstracting and storing previous design experience. Also, styles can lead to code reuse when it comes to invariant aspects of an architecture (for example a family of pipe and filter implementations could use UNIX operating system facilities)
- styles are a very effective way to communicate key design ideas behind the organization of a large system; it is easier for others to understand the organization of a system once agreed on the conventions to be used
- by constraining the design space, an architectural style provides guidance for the future evolution of a system and allows for style-specific analysis
- the use of standardized styles supports interoperability, which should come naturally especially within the scope of the same style
- just like SWA itself, styles allow for different views of the same entity[17], each view emphasizing specific structural aspects. This way, one can expose and estimate benefits or drawbacks of a particular architecture.

Points of view on SWA styles

As already mentioned, the first ones to investigate SWA styles, and actually apply the word 'style' to SWA, have been again, Perry&Wolf. In 1992 they defined the architectural style as being [50]

'(...) that which abstracts elements and formal aspects from various specific architectures. An architectural style is less constrained and less complete than a specific architecture'

---

[16] In M. Gentleman's opinion, ADLs are too formalized; the SWA perspective is at a level that is too high to allow for such formalism.

[17] For instance, some database architectures could be looked at as batch-sequential; on the other hand, there are authors who introduce databases as layered architectures. In their turn, layers can also be viewed as a flow architecture; and so on... These styles will be discussed later in this chapter.

A more complete and precise definition was attempted by [14] in 1996; they see the architectural style as being

'a family of SW systems in terms of their structural organization. An architectural style expresses components and the relationships between them, with the constraints of their application, and the associated composition and design rules for their construction'

D. Garlan, who emerged in the last years as one of the authorities in the SWA field, provides in his turn the following definition of an architectural style:

'an architectural style characterizes a family of systems that are related by shared structural and semantic properties'

This definition emphasizes the fact that it is actually a particular class of architectural descriptions that has been called an 'architectural style'. It is an aspect that makes the definition a lot more versatile with respect to a broad class of applications.

From the definitions for both SWA and architectural styles, one could see there is a fine line dividing the two: what may look like an instance of architecture in a given context, may actually be an architectural style to a broader approach. An architectural style is always less constraint and less complete than the actual architecture; it may very well focus only on certain aspects of the architecture. As [50] mentions[18], styles capture those high level design decisions that are more likely to suffer from erosion and drift, especially when it comes to a large project with cooperating teams working on it. SWA styles are meant to provide visibility to the important aspects of the architectural design (just like in the case of design patterns) so as to prevent their violation. Let's proceed and take a look at some of the most influential points of view on SWA styles.

SWA Styles by Perry&Wolf

Along with setting the foundations for the study of SWA, Perry and Wolf also introduced SWA styles, according to the definition we have seen before. Nevertheless, they did not come up with a taxonomy per se.
[50] analyses examples of compiler architectures, organized sequentially or around shared data, however no attempt was made at that time to introduce any taxonomy of SWA styles.

[18] This particular paper only makes reference of sequential and parallel processes as architectural styles.

SWA Styles by Garlan&Shaw

Garlan&Shaw focused a lot on putting together a taxonomy of SWA styles most commonly used up to date. In [28], along with a brief definition of SWA as

'a collection of computational components (...) together with a description of the interactions between these components – the connectors',

a number of the most popular architectural styles is examined; nevertheless, no clear definition is given, on what an architectural style really means. The architectural styles identified on this occasion are:

- pipes and filters
- object oriented (and data abstraction by and large)
- event based (implicit invocation)
- layered systems
- repositories
- table driven interpreters

The authors make note of other important categories like Distributed Processes, Main Program/Subroutines, Domain Specific SWA, State Transition Systems and Process Control Systems, before exploring how appropriate different styles are to specific applications. A lot of the ideas in the paper are later revisited in [57]. We postpone any further details about the individual styles until the end of this thesis; the merit of the paper though, is to open the perspective on some of the main architectural patterns currently used. No further classification of these styles is attempted; some of the styles mentioned in the paper might be debatable (for instance, we think OO is a design methodology, rather than a SWA style).

SWA styles by Shaw

M. Shaw did a further refinement of the architectural styles in [56]. SWA styles are viewed here as a set of patterns that occur regularly in the overall organization of the systems or component interaction, system design often making use of several such patterns combined in various ways. The following

styles are identified as the major players in architectural design, along with a common framework to contrast them:

- Pipeline

System model	mapping data streams to data streams
Components	filters (local processing, purely computational)
Connectors	data streams ('pipes' as named on other occasions)
Control structure	data flow

- Data abstraction (object-oriented)

System model	localize state storage (encapsulation)
Components	managers (servers, objects, data types)
Connectors	procedure call (method invocation = procedure call + dynamic binding)
Control structure	decentralized, usually single thread

- Implicit invocation (event based)

System model	independent reactive processes
Components	processes that signal significant events, without knowing the recipients of the signal
Connectors	automatic invocation of processes with registered interest in events
Control structure	decentralized, components are not aware of the recipients of the signals

- Repository (including databases and blackboard systems)

System model	centralized data, usually richly structured
Components	one memory, many computational processes
Connectors	interaction between computational units and memory via direct data access or procedure call
Control structure	depends on the type of repository: may be external for databases (depends on the input data stream), predetermined or internal (depends on the state of computation as for blackboards)

- Interpreter

System model	virtual state machine
Components	one state machine as the execution engine and three storage entities: current state of execution engine, program to be interpreted and current state of program to be interpreted
Connectors	direct data access and procedure call

Control structure	state transition for the execution engine, input driven for the selection of what to be interpreted

- Main program and subroutines

System model	call and definition hierarchy
Components	procedures
Connectors	procedure calls
Control structure	single thread[19]

- Layers

System model	hierarchy of opaque layers
Components	most often collections of procedures
Connectors	most often procedure calls; might also be client server
Control structure	single thread

Each of the styles above is considered as making use of particular component-connector matches, which uniquely define it. It is important to note from this paper that the characteristics of a style have nothing to do with the functionality of the components, performance or any other properties included in the specifications. It is the way components interact amongst themselves, (that is, the abstractions behind the component's interface) that uniquely define a style. This comes along the same lines with our assumption, made in the previous chapter, that it is better to define the interfaces in the architectural step.

SWA styles by Katzman

Very close from the Carneggie Mellon school of thought is R. Katzman. In his former SWA course, he attempts a first classification of SWA styles as shown below. With some additions, this classification is very much based on the Garlan&Shaw findings (the styles mentioned by Garlan&Shaw are written in italic):

- independent components:
  - communicating processes
  - *event systems* - *implicit invocation*
  - explicit invocation

[19] We think the granularity of threads is too low to be visible on the SWA level.

- data flow
  - batch sequential
  - *pipes and filters*

- data centered:
  - *repository*
  - blackboard

- virtual machine:
  - *interpreter*
  - rule-based systems

- call/return:
  - *main program and subroutines*
  - *object oriented*
  - *layered*

It is interesting to note, looking at the last two taxonomies, that while Shaw seems to be more concerned with the functional grouping of components, Katzman seems to rely more on the nature of communication amongst components as a basis for his classification.

SWA styles by Go5

As mentioned before, [28] outlines that SWA styles can also be viewed as a set of patterns that occur regularly in the overall organization of systems or in component interaction. With the overall development of design patterns in the late years, this pattern-based perspective of SWA has been investigated more and more seriously.

[14] takes a very methodical approach to patterns on all stages of the SW development process. Concerning SWA in particular, 'the Gang of Five' believes that architectural patterns express

'fundamental structural organization schemas for SW systems; they provide a way to organize generic modules, specifying their type of responsibilities, together with rules and guidelines for organizing the relationships between them'

Accordingly, [14] considers four categories of patterns on the architectural level, based on the goal they are trying to achieve:

1. support for decomposition of a system task into cooperating subtasks *(from mud to structure)*[20]
    - pipes and filters
    - layers
    - blackboard

2. support for creating *distributed applications*
    - broker

3. support for creating systems that provide for a lot of *human-machine interaction*
    - model-view-controller
    - presentation-abstraction-control

4. support for applications that require extensive adaptation to changing functional requirements *(adaptive systems)*
    - microkernel
    - reflection

We think that the first category very much overlaps with the work of the Carneggie Mellon folks. The next three groups fall rather into the DSSA field (especially the man-machine applications) and, in our opinion, their components could be very well placed amongst the subdivisions of the first group. It is worth to note that the Microkernel and Reflection patterns are very good examples of how an architecture can emphasize up front one of the quality attributes (maintainability in this case) more than the rest. Having said this though, the [14] taxonomy is missing some important SW organization patterns on the architectural level (for instance the 'main program and subroutines', which is probably the oldest and most popular way to organize SW). However, the book makes very good points on the advantages and drawbacks of various SWA patterns, with respect to the topics that are debated.

Sources for architectural styles

It is quite hard to establish precisely where the aforementioned architectural styles come from. For sure, software folklore was the media to disseminate

[20] R. Johnson used to call a 'ball of mud' the situation where you have to meet a large set of requirements and are aware of a multitude of related constraints, but don't really know where to start from.

them; however, in terms of releasing a new architectural layout, 'architectural visions' and 'black magic' are often referred to as being the starting point.

P. Krutchen makes an interesting argument in [40], by identifying (at least) three potential sources for SW architectures (we would say these hold for SW designs in general). In his view, any given architecture is, in various proportions, a combination of intuition, theft (we would say rather 'reuse') and method:

- *intuition* as applied to our case is the ability to immediately apprehend the requirements and conceive around them without reasoning. A lot of the new architectural designs came out this way. Intuition, based on experience, plays an important role especially in the case of seasoned SW designers appointed as SW architects. After recognizing a pattern or adopting a new point of view, a new architecture (or architectural element) appears, which is then confronted to the requirements or fitted onto the existing system.
- *reuse* seems to be the method of choice in developing SW systems; most of the times a SW architecture (or just elements of it), is literally 'lifted' from
  - a previous application the designer happens to be familiar with
  - systems with similar characteristics
  - models found in the literature

  Reuse is the main reason behind building SWA taxonomies, and there is nothing wrong with it as long as it is carefully tailored to the particular application at hand.
- *methods* encompass the systematic (and that is, conscious) way of building architectures, based on system requirements and taking into account possible constraints. A number of methods have been documented so far, still, in many cases SW architects make use of heuristics not completely defined.

Again, the weighting of these three sources varies according to the novelty of the application and the experience of the architects designing it. The paper suggests (from the author's experience) some typical profiles of the way the three sources contribute to architecture in general:

- classical project: 80% reuse, 19% method, 1% intuition
- unprecedented project: 30% reuse, 50% method, 20% intuition

There is no doubt that intuition, which is usually associated to creativity, will be always an important factor in any type of engineering. At the same time though, intuition has to be integrated with rigorous methodologies and validated by experience; the more we rely on intuition, the higher the risks.

COTS

We have already pointed out that requirements are constantly evolving in today's dynamic SW market. Consequently, the ability to design large (and complex) SW systems to closely match user needs, becomes a challenging task; aggressively shrinking time to market pressures only add to this challenge.

One of the solutions tossed on the table was tailorable and adaptable SW, which means SW that can be modified or further developed (for requirements not accounted for originally) after delivery and during its use. The other solution involves integrating Commercial Off the Shelf SW (COTS) and has gained a lot of momentum over the past few years. The component-based concept for developing SW is not new. As early as the mid-seventies module and interface definition languages were brought forward in an attempt to better aggregate the SW into well-defined and well-structured modules. These languages provided execution units with well-defined interfaces, apart from interconnection mechanisms for gluing the modules together; nevertheless, the architectural perspective of the system was left behind, in favor of component description. A serious step forward was made through the emergence of open architectures in SW, attempting to emulate their counterparts in HW; the open concept here meaning support for both scaleability and heterogeneity. This means moving from monolithic, custom made (and many times proprietary) designs to providing for multi-vendor SW components working together in the same system. [20] mentions a few examples in this area; the ones below are a more unified view:

- *application frameworks* which are sets of guidelines for developing components that can work together. They usually consist of application programming interfaces (standardized inter-component calls) along with guidelines on how these calls should be used. The early releases in this category were GUI toolkits like X-Windows or Motif.
- SW *bus architectures* based on a similar design principles as the HW bus[21]. The bus could be described as an abstract agent who de-couples its clients by taking over all interfacing and coordination needs. Since individual components are only prepared to talk to the bus, heterogeneity of architectures or languages is easily supported. The first accepted model of this kind was CORBA, released by OMG in 1994. Special varieties of the SW bus model are coordination models/languages, which only provide

[21] This might be a very sensitive statement...

abstract primitives coordinating or interfacing SW entities; these primitives rely usually on a host language at implementation time.

Out of the two above, the bus architectures have become increasingly popular since they can quickly enable a system put together out of heterogeneous components, without writing coordination SW other than the bus itself. However, there is a significant exposure to embedding interconnection assumptions into the execution modules, in light of our previous discussion on this topic.

The biggest difficulty in building COTS applications is to 'glue' these applications together, that is, design and implement the coordination SW that manages interconnection requirements and compensates for mismatches. As previously discussed, let's not forget though that the flexibility of integrating COTS is inversely proportional to the degree to which interaction assumptions have been embedded into components. The coordination SW used (and adapted) to glue the components together cannot always overcome coordination assumptions already built into the modules; this fact has been proven by experimentation on architectural design environments like Aesop or Synopsis [20][27]. Usually, the same set of components could be used to generate different architectures, only differentiated by the selection of a different coordination mechanism. It turns out that the latter, rather than component implementation is a defining factor in the system architecture.

Architectural decisions

The architecture of a system, like any other engineering endeavor, is the result of a set of decisions, which, in their turn are mostly influenced by the system requirements. This fact was apparent even from the early works of Perry&Wolf, the rationale for architecture being part of their very first SWA definition.

Architectural decisions are responsible for choosing a particular architectural style, and are the motivation for component selection, connection or coordination mechanisms. It is fairly obvious that there would be different sets of decisions made for a real-time system as opposed to a general-purpose system. Along the same lines in the case of a real-time system, there are significantly different sets of decisions in a case where the deadlines are soft (soft real time system) as opposed to when the deadlines are hard (hard real-time system).

Even though Software Architecture has just passed the folklore stage, one could still draw correlation between influences and decisions in Software Architecture. Ideally, this correlation would be independent of the order in which architectural decisions are made or the influences they could have on each other. Under the direction of D. Garlan, the Software Engineering Institute at Carnegie Mellon conducted a set of case studies on large system architectures, in another attempt to extract and document successful architectural design practices, based on previous experience. Most of the ideas expressed below are based on their findings.

It is difficult to trace and classify all the factors that influence architectural decisions: they rank from the most obvious, system requirements, to technical expertise, previous experience or organizational culture. To start with the requirements, one could categorize decisions as follows:

- *Functional* requirements that are time-independent: given specific standards of correctness, these are requirements verified by watching the system in all its states and comparing its outputs. Mathematically speaking, as pointed out by Parnas, these requirements are expressed as relation of the inputs and state of the system on one hand, and the outputs on the other, where the state of the system is not a function of time.
- *Performance* requirements are the ones specifically checked against time, more formally, relations in which the state of the system is a function of time.
- *Non-functional* requirements are measurements against the development process rather than against how the system performs (they are independent of the functionality intended or performed by the system). As mentioned before, non-functional requirements mostly include the '-ities': maintainability, scalability, etc.

Most authors consider reliability and availability on one hand, and safety, security and fault-tolerance on the other hand, into non-functional requirements, and bundle them under the name of 'quality attributes'. However, we support the SEI view, which prefers to tag these attributes as functional requirements, a start-up set for any well-engineered system. SEI classifies them as 'functional quality attributes'.

Similarly to M. Jackson's suggestion that it is worth to try and analyze the problem before attempting to find a solution, the SEI focused on requirements before attempting to find what decisions make the architecture of a system. Intuitively, one could assume the following requirements structure:

- *driving requirements*: requirements that architects deem are the hardest to meet. Usually these are the ones responsible for the most ambitious architectural decisions.
- *requirements* (by and large): those that are relatively easy to meet by several architectures. Out of these architectures, the one that better fits as many driving requirements as possible is the starting point for further investigations.

Based on the cases under study, the SEI makes the following observations:

- The driving non-functional quality requirements are a major influence in the architectural decision.
- The driving functional quality requirements are a major influence in the architectural decision.

There is one additional observation to be made in the case of embedded systems, (where performance usually plays in important role): the driving performance requirements are a major influence in the chosen architecture.

It may look odd, but although functional requirements seem to play the key role in system development, taking up significant design and verification resources, they don't necessarily play a major architectural role. Basically, to have driving functional requirements means that functional requirements will be hard to meet and some of the reasons could be the following:

- difficulty in specifying what outputs are required, this being a specification problem.
- difficulty in implementing software to achieve the functional requirements; this is a design/implementation problem.
- difficulty to meet performance/accuracy constraints - most often this could be related to algorithmic issues (for instance, cases where small, accumulative errors lead to malfunctioning of the system only after running for a long period of time). Accuracy is very likely to stay the same if computations are contained within the same module or spread out over a few.

For the reasons above, SEI concludes that driving functional requirements (except for what we called functional quality attributes) are usually not a major decision factor in the architecture of the system. Eventual exceptions might be for instance distributed systems with an intensive data exchange, where the right data has to be in the right place at the right time.

The fact that the experience in system development of the organization, and in particular of the individual, plays a role in the design decisions by and large, has been well emphasized by the pattern community in [14][25]. The same holds for architecture as well. From the perspective of an organization, a bias towards a particular architecture can be induced by factors like:

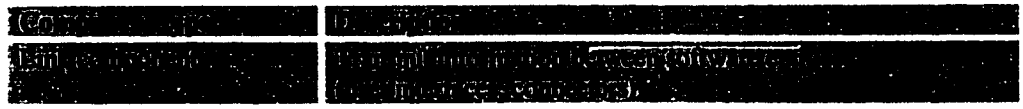
- the existence of tools/frameworks geared towards a specific architecture
- incentive to use/re-use a particular architecture for business reasons are usually a major decision factor
- development history (especially success stories) with a specific architecture would have a minor influence

From an individual's perspective though, past development experiences with a particular architecture will exert a major decision influence, proportional to the estimated success for this time around.

Further choices lie in decisions like whether to bundle everything into one process or several, component selection or component interaction. In contradiction with SEI we trust that component selection/interaction is a far more important decision compared to how many processes we should span. Especially when it comes to structuring the system along event responses, component selection and their interaction is the first aspect to be considered; the number of processes is only a consequence of this first step and could be very well deferred to the design stage. Once the functional partition has been made, it is desirable to take advantage of concurrency as much as possible; this could be achieved on either single or multiprocessor, the latter leading to the more challenging task of assigning the right processes to the right processors.

Garlan and Shaw make some interesting suggestions with respect to component selection (far from being exhaustive, given the fact that many of the components are domain specific) which we adopt here in a slightly different format:

Computation components	**Simple input/output relations; no states (functions, filters, transforms)**
Controller components	**Coordinate event sequences of other modules (scheduler, manager, synchronizer)**



As for interconnection mechanisms, there are a large variety of them available, some of the most popular examples being: broadcast, blackboard, mailbox, semaphore, etc. What is important though about interaction is not the mechanism being used but how much of it is carried and how it is structured/restricted. The classification of interaction types associated to our style taxonomy matches very well the one suggested by SEI. Accordingly, interactions could be classified as:

- *unrestricted*, where any component can communicate to any other component
- *managed*, where there is a way to restrict communication amongst components based on a specific discipline (on a hierarchical basis for example, in the case of Restricted Object Based Design – see appendix)
- *forbidden*, that is, no interaction at all amongst specific components (on the same hierarchical level in the case of Restricted Object Based Design)

Obviously, it is mostly the case of the last two where coordination comes into place.

It is not easy to correlate the multiple influences exerted on architecture with the final decision; nevertheless, it is possible to identify relationships between groups of influences and groups of decisions. The major observation here is that there might be relationships amongst components, identified during the development process, that disappear at run time (and the other way around). Take for example the concept of a module itself: although it is a conceptual and well related grouping of programs at development time, at run time one can only see individual programs running (eventually in parallel) without being able to tell which module they come from. There is no development methodology able to take into account all the interactions and dependencies that appear at run time; the study of how architectures behave dynamically is a whole research area in its own [see papers by Sylvia Stuurman]. Therefore it is worth making a difference between static and dynamic decisions, the latter being actually a subset of the former:

- *static* decisions are basically all the decisions made at development time; for now, future dynamic decisions are only assumptions.
- *dynamic* decisions are decisions whose effects are only seen at run time

Having said that, we could sum up our previous arguments on the need and importance of Software Architecture by concluding that static architectural decisions are only geared to non-functional properties; therefore they tend to be motivated by the driving non-functional requirements. Ideally, at any given time there would be traceability between the driving non-functional requirements and the architectural decisions. On the other hand, the algorithmic structure and the amount of interaction amongst components is expected to have a significant influence on the performance of the system; other examples point to the relation between reliability or security and the interconnection strategy/mechanisms. This leads to the assumption that dynamic architectural decisions tend to influence the performance properties of the system. Again, driving performance requirements will motivate dynamic architectural decisions and it is desirable that the correlation between the two always be traceable.

All of the above are empirical observations (as in the case of patterns) taken from a study conducted by SEI and based on existent legacy systems; more information needs to be gathered and eventually models be developed in order to help Software Architecture step from folklore to engineering.

Concluding remarks

The purpose of this chapter is to actually pave the way towards a coordination-based approach to SWA. We introduced architectural styles, as they have been reflected in the literature and provided an outlook of various taxonomies on this topic.

In support of SWA as a mandatory step in the SW design process, the intent of SWA styles is to act as a set of guidelines for the architecture of SW. Styles capture relevant decisions on the architectural elements, and emphasize relevant constraints and relationships amongst those; nevertheless, to various extents, the level of constraint could be left up to the architect. A style is always less constraint and less complete than the actual architecture and it is not uncommon for a style to focus only on certain aspects of the architecture. This is a major difference between architectural styles and patterns, a subject to be discussed in more detail throughout the thesis.

Following this brief outline on the most important directions in the study of SWA styles, we compiled a list of what we thought to be the most important styles identified so far. The short descriptions attached to each are actually ours,

in absence of any definitions from the authors in question, but hopefully close to their view.

	Only examples, no attempt of a taxonomy.	*First attempt of a style classification*	*Further refinement of Garlan &Shaw*	*Significant contribution based on the Carneggie-Mellon school.*	*Taxonomy oriented very much towards DSSA.*
Batch sequential	examples mentioned as 'multiphase'.	Degenerated case of 'pipes and filters'.		Independent programs performing data transformatio n in steps.	
Implicit invocation		Components invoking registered operations via broadcast. Also appear as 'event systems'.	Independent reactive processes making invocations without knowing the recipients. Also appear as 'event systems'	Only 'event systems' mentioned (objects or processes registering procedures with events announced non-deterministica lly)	

Repository		Central data structure operated by independent components.	Structured, centralized data directly accessed by more computation processes (blackboard and database included)	Mentioned as database (large central data store controlled by incoming transactions)	
Interpreters		Virtual machine produced in SW.	Virtual state machine.	Execution engine in SW (subset of rule-based systems).	

Chapter 3

COORDINATION AND SOFTWARE ARCHITECTURE

In close relation with the explosive evolution of computer programming, as well as the increase in size of computer systems (locally or spatially distributed), the theory of coordination appeared lately as a significant emerging discipline. Nevertheless, the notion of coordination is by all means not restricted to computer applications. From leisure activities to high-end technology, most of us have at least an intuitive understanding of what coordination means; however, we may not be aware that, although good coordination is nearly invisible (and often taken for granted), it is mostly perceived when it is missing. In general, coordination becomes important in very complex systems; we have experienced this fact, at least from a social perspective, starting with early human organizations (civil or military)[22] and ending with today's large corporate. Nowadays a great deal of work in different research areas, spanning various disciplines, from biology going through mathematics, economics, management, psychology or finally computer science, contributes to the overall view of coordination.

In the following pages we set out to take a brief look at coordination as different areas of research see it[23], in order to pave the way towards a coordination-based approach to SW architecture.

On the definition of Coordination

As pointed out in the introduction, coordination is definitely part of every day life, with or without us being aware of it. If the intuitive meaning of coordination is

[22] Take oligachical organizations, for example.
[23] A lot of the ideas in this chapter are based on [42].

usually sufficient for common use, when it comes to critical assignments, the term must be well understood and applied. To better define the meaning of coordination, a good start is (as always), the Webster Dictionary:

'coor'dinate – (...) bring into order as parts of a whole (...)' [Webster's New Dictionary – 1994]

In general (and with respect to computers in particular), coordination could be looked at as the process of managing dependencies between activities; in this context, dependencies could be either shared resources, relationships between activities, or timing/simultaneity constraints.

Of course, there is no shortage of attempts to define coordination; some of the significant ones are outlined below [National Science Foundation – 1989]:

1. *'The operation of complex systems, made up of components'*
2. *' The emerging behavior of collections of individuals whose actions are based on complex decision processes.'*
3. *'The joint effort of communicating actors towards mutually defined goals.'*

We agree more with the third definition, since it is the only one that brings forward the common goal coordination is pointed to; the operation of complex systems made out of components is not necessarily coordinated. On the other hand, collections of individuals do not necessarily carry coherent actions, moreover, lead to convergent results.

Going back to the third definition above, other authors have been even more concerned with the 'common goal', which is part of coordination:

4. *'(...) composing purposeful actions into larger purposeful wholes'* [A. Holt – 1989]
5. *'The additional information processing performed when multiple connected actors pursue goals that a single actor pursuing the same goals would not perform'* [T. Malone – 1988]
6. *'The integration and harmonious adjustment of individual work efforts towards the accomplishment of a larger goal'* [B. Singh – 1992]

However, there is another standpoint we are inclined to agree with. The definition below equally emphasizes the existence of a common goal, or just the need to manage shared resources when the focus of effort within a system is not the same (very much the case of distributed systems):

'Activities necessary to maintain consistency within a work product or to manage dependencies within the workflow' [B. Curtis –1989]

Just as in the case of the Software Architecture, there is no generally accepted name for coordination. Therefore, when referring to coordination by and large, many authors in the computer field use the term of 'coordination theory', in contrast to coordination as applied to computers, when they simply use the term 'coordination'. And again, like in the case of Software Architecture, one has to go and make analogies with some of the well-established research disciplines, in order to apply some fundamental coordination mechanisms or come up with some new ones. For example, lessons learned about coordination in human systems could help understand computer-based systems as well; on the other side maybe, new coordination models, once successfully applied to computer systems, could be implemented in human organizations[24]. On the other hand, facilities offered by massively parallel computer architectures are used in genetics to model DNA algorithms. The bottom line is that coordination could occur in many types of systems, as seen in previous examples. In the particular case of computer systems, there might be specific interactions which resemble more to others outside the computer field; it only makes sense to identify and apply these analogies where and when appropriate.

In order to talk about coordination between any kind of entities, first of all there have to be some sort of dependencies between them; without any interdependencies, there is nothing to coordinate. The term coordination is usually used with an inclusive connotation: notions like collaboration, cooperation or competition, in spite of having each their own meaning, highlight in fact different aspects of coordination.

The broad perspective of coordination

Having defined coordination as basically the process of managing dependencies between activities, we have also mentioned that the coordination theory actually calls for an interdisciplinary study of coordination in order to develop theories about how coordination can occur in different kinds of systems. Since coordination in a broad sense spans well beyond the boundaries of computing, the current status of this emerging theory is more like a collection of results, analogies and partial frameworks. Further development of coordination theory

[24] ... but only to some extent; while in computing the incentives of a program module can be controlled by a programmer, in human systems there are also motivations and emotions added to a specific incentive. This makes the mapping from computers to human organizations not very straightforward.

requires more characterization of the various kinds of dependencies and identifying the coordination processes that can be used to manage them.

Let's now take a look at some of the facts that triggered the aforementioned analogies, with the strong belief that the identification of coordination similarities across disciplines would facilitate a useful exchange of ideas in this area.

Biology

Biology deserves the first mention here, nature being in general the first model when it comes to man-made things. SW is no exception; from concepts like 'anthropomorphic programming' to distributed applications, many of the coordination examples offered by biology triggered a lot of interest in how to apply them to computer science [17]. Take for instance coordination patterns that were found in *insect behavior*[25]. In this case, despite the simplicity of the individuals, groups of 'social' insects exhibit a rather complex behavior using a variety of simple rules. Most of the studies center around distributed optimization cases, extremely effective and susceptible to be applied in computers to performing complex tasks (by distribution of activities over massively parallel systems). Common examples come from the life of insect colonies and deal with the allocation of different workers to various tasks such as searching for new food sources, gathering supplies from existing ones, regulating the group temperature, or guarding the hive.

Let's first consider the example of ant colonies and how they manage to find (almost blindly) the shortest path from their colony to a feeding source and back. In this case, the media to communicate information about paths is a substance called 'pheromone'; the use of two simple principles lead to finding the optimal path to a food source:

- a moving ant lays certain quantities of pheromone in its path; it is also capable to detect the presence of it.
- another ant, otherwise moving randomly, encounters this trail and uses it as a stimulus to go that direction, this way reinforcing it with its own substance

As a consequence, the more ants follow a specific trail, the more that path becomes attractive to be followed. If an obstacle appears in the shortest path

[25] Various observations have been made, at different evolutionary stages, starting from bacteria to ants, caterpillars, bees and many others.

and cut it off, ants would go around it and, following the two principles above, a new optimal path is found.

Similar examples come from the life of bee colonies. In particular, there are simple local rules to control the allocation of food collector bees to particular food sources:

- first, nectar is unloaded (from forging bees) by nectar storing bees, at a rate depending on nectar consistency
- second, if bees are unloaded too rapidly, more bees are recruited to that particular food source

The final result is that more bees collect nectar from better sources; this is a good example of decentralized control with very flexible response to local stress (adaptive control).

Other interesting studies involve *group behavior* and coordination amongst groups of animals, dealing for example with optimizing the size of the hunting pack (that is, more chances to catch) against the cost of a lower share for each individual. And such examples could continue.

In the end of the paragraph, let us just mention human *physiology* (as still related to biology and coordination applied to it). From a coordination perspective, this field can be also viewed as studying the way in which the activities of different parts in the human body are coordinated to keep the person alive and functional.

Economics

A great part of economics involves coordination (and the study of it), mostly concerning information flow, interaction between supply and demand, or its impact on resource allocation. For instance, one of the major results of this type of studies concerns *resource allocation* as a balance between utilities and services:

- consumers tend to maximize each their utilities and, on the other hand,
- service providing companies want to maximize their own profits

The result is that global resource allocation becomes optimal (in other words, no individual utilities could be increased without decreasing another).

Some other areas of study, like transaction *cost theory*, deal with the conditions under which a hierarchy is a better way of coordinating multiple entities than a market. Along the same lines, *agency theory* studies ways to create local incentives in the form of actors behaving in such a way to attract interest and action from other actors, so as to achieve a specific goal without hierarchical supervision. In terms of information flow, there are scenarios (in *team theory*) to analyze how information should be exchanged amongst cooperating entities of a system in a case where all have a common goal. Moreover, there are situations where it is possible to make entities reveal information they posses even though they have conflicting goals. A short example of the latter is the so-called 'second price auction', a case where participants submit a confidential bid, the highest bidder being required to only pay the amount of the second highest bid. This mechanism motivates bidders to show the real value they place on an item, rather than focus on what they think would be the next highest bid.

But the one research area in economics, with correspondents in other fields of science, is the *operations research*. It analyses various coordination mechanisms, with special emphasis on optimal techniques for coordination decisions. A number of scheduling and queuing policies and techniques (like linear or dynamic programming) are used here, in order to make resource allocation optimal.

Sociology

Sociology and psychology put together their own separate research area, which deals with coordination: the *organization theory*, which studies how people interact together in formal organizations. To summarize some of the most important results in this field (most of them established in the 60's and early 70's), one could simply say that all activities including more than one executor require:

- some way to *divide activities* amongst different executors
- some way to *manage interdependencies* amongst different activities.

Interdependencies could be of at least three kinds:

- *pooled*, where (otherwise) independent activities share or produce common resources
- *sequential*, where some of the activities rely on the completion of others before beginning
- *reciprocal*, where each activity requires input from another

Instances of the coordination mechanisms proposed to handle interdependencies as above are:

- *standardization*, in which case a set of predetermined rules govern the performance of each activity
- *direct supervision*, where an activity manager handles interdependencies between activities on an individual basis
- *mutual adjustment*, where each entity makes ongoing adjustments to handle dependencies

The mechanisms described above can be extended to manage dependencies not only between individual activities, but also between groups of activities[26]. Activities could be grouped into units such as to minimize interdependencies among groups. This could be achieved by grouping together activities with tight interdependencies, thus generating the smallest units; these units are grouped together with other units they have weaker interdependencies with, and so on. Combinations of this kind give rise to various organizational structures, hierarchical (vertical) or matrix (horizontal) common in human organizations. Let us mention here that organization theory became lately a very hot topic in computer science, along with the interest for patterns. Some of the preeminent figures in the pattern community went as far as studying the interaction between organizational patterns amongst developer groups on one side, and software patterns on the other side. In other words, this kind of studies focus on how the architecture of a software system could be mapped on the developer's organization and the other way around.

Coordination as applied to computer systems

We will now turn towards applying concepts of coordination theory to the design of computing systems (distributed or not). Before going any further though, it is worth mentioning that there is no silver bullet in terms of analyzing and partitioning components of coordination in a system; one might very well consider factors outside of the coordination area.

For a proper analysis, we should start by identifying the components of coordination in a specific situation. In light of our definition of coordination (as being the management of dependencies between activities), first we should find

[26] Similar concepts are used by the Multiactivity paradigm for partitioning system responses.

all activities that are interdependent, then see what the dependencies are (and eventually who performs them). It is also indicated to come up with metrics and evaluation criteria[27] on how well dependencies are actually managed and watch how these goals are met following different coordination scenarios. The approach of selecting goals could easily work in a case where goals are convergent, but the partitioning becomes more challenging whenever we have to deal with conflicting goals (which is probably the case most of the time; at least some amount of goal conflict is always present). In this case it is indicated to identify which goals are conflicting and look at the performance of the system as an entity; it is hard to provide a detailed evaluation right now, especially since conflicting goals could actually lead to increased overall performance.

A good way to help evaluate alternative designs of distributed application systems is to apply... coordination methods. Various authors have developed mathematical models for distributed applications, going from probabilities to chaos theory. It was demonstrated for example that in the case of a system with a large number of processors (say, over 20), where any processor can exchange tasks with any other, the behavior of the system becomes unstable with increasing the number of nodes[28]. Nevertheless, when the processors are grouped hierarchically into clusters, frequently exchanging tasks amongst themselves and only occasionally with the exterior, the system becomes stable for an arbitrarily large number of processors. This very important result comes along the same lines with the Multiactivity paradigm, which is also based on a hierarchical organization, with responses being the equivalent of processor clusters here.

From our perspective, the purpose of studying coordination is to apply its principles to computer systems. A lot of useful analogies could be made for sure with non-related disciplines, but let's focus now on distributed computing architectures; connecting systems together is easy, coordinating their activity is hard. As outlined before, one of the main problems encountered in distributed computer systems is how to assign tasks to computing entities; we also said that competitive bidding was a suitable solution when it comes to distributed computing. One of the interesting features of this solution is the extent of decentralization and flexibility for both clients and servers. Each of them is able to use its very own criteria in order to participate to the bid[29].

---

[27] Examples of evaluation criteria would be establishing measurable goals, at least for some of the key interdependent activities.

[28] We tend to make the same extension to architectural modules.

[29] Servers can be selected on their estimated completion time of the job, clients to be served may be selected on the basis of load and how long have been waiting to be served.

The same concept could be applied to workstations connected on the same LAN for instance, with the result of locally coherent load sharing and scheduling according to various priorities. Databases (data storage by and large) are a place where competitive bidding also finds a place; for instance if a software entity wants to keep data or maintain pointers to another software entity, it has to 'pay rent' to the owner of that memory space. The rent could be determined by competitive bidding and once a 'subscriber' fails to pay, memory is de-allocated. There are studies on how to implement such mechanisms without excessive overhead. This model may not be effective on a small scale but it certainly pays off in complex networks where direct access between software entities is complex or prohibited (inter-organizational networks for example). Another important problem in parallel systems is how and when to route information between work units. In AI for example, where programs have to search a large variety of possible solutions or work with incomplete data and accept partial solutions (see the Blackboard architecture), it is essential to efficiently exchange information, so that pieces of knowledge can be put together at the right time. This way, redundant or unnecessary work can be avoided. One way to achieve coordination in this case consists of a number of 'sprites' (we can call them work units) that work in parallel and interact through a database. In order to be activated, each such work unit requires certain conditions to be true in the global database. When all the conditions are met in the database, and once the work unit is triggered, it can do one of the following:

- *compute* new results to be added to the database
- *create* new work units, in their turn awaiting the right conditions to trigger them
- *inactivate* work units whose work is now known to be unnecessary

Having each such unit supported by a 'sponsor' solves most of the resource allocation problem; without such support no processing time would be allocated to complete the work. Moreover, a 'sponsor' can support multiple work units (say to work on proving the validity or invalidity of a specific assumption) and, according to which work unit is successful, withdraw the support from the others. Experiments of sharing intermediate results with this model (even executed by time sharing on a single processor) showed dramatic improvements in the execution of algorithms.

Computer science in its turn contributes to the interdisciplinary understanding of coordination. Research in computer science focused on topics like shared resource management, information flow management (including unreliable

sources), or task partition and assignment[30]. Before going into more details, we will briefly consider the areas just mentioned:

- *allocating work* to executors is one of the most important problems in the world of computers. Gelernter and Carriero suggest three ways to divide parallel programs into units:
  1. By the type of work to be done
  2. By resource availability
  3. By subparts of the final result
- *sharing resources* is more and more an area of concern in software/computer engineering, given the need to efficiently manage resources like processors, memory or access to I/O devices. Various mechanisms able to handle contention are now in place: semaphores, mutexes, etc. Specific applications in database systems developed their own mechanisms like locking or time-stamping to allow multiple access to shared data.
- *information flow management* is another important class of issues. The Coordination Based Architectures (discussed in this thesis) are one way to allow parties to share information, even without knowing the source of it or whether any other parties need it (see again the Blackboard Architecture). Also, with respect to *unreliable information sources* (not in terms of credibility but in terms of reliability of the transaction), protocols are in place, to ensure that either all the operations in a transaction are completed, or none (if this is desired).

Coordination patterns

Once we accept coordination as a way to manage dependencies, a first step to study it is to identify the most common dependencies and then the ways to manage them. [42] brings forward a set of dependencies encountered usually (and spanning beyond the boundaries of computing), which are a good start for such an analysis:

- *Shared resources* – usually managed via priority order, 'first-come-first-served' mechanisms, time division (budgeting), bidding, etc.
- *Task assignment* – managed similarly to shared resources (tasks are also resources in the end). *Task/subtask* relationships deserve a special mention here, for using techniques like goal selection or task decomposition

[30] Of course, the study of coordination, as applied to computer science, took advantage of results coming from other more established disciplines (thus with more experience).

- *Producer/consumer* relations – further subdivided in
  1. *Prerequisite constraints* – managed by notification or tracking
  2. *Transfers* – managed by inventory tracking
  3. *Usability* – managed by standardization, user inquiries, joint design, concurrent engineering in the case of design for manufacturability
  4. *Simultaneity constraints* – managed via synchronization or scheduling

Having said this, we come back to the idea of identifying *patterns of coordination* and eventually develop *coordination handbooks* to store and transfer knowledge about coordination – an approach well proven by other (already) established engineering disciplines. The list mentioned above is by no means exhaustive, however it could be a good start to systematically analyze dependencies and their associated mechanisms.

Shared Resources

Sharing limited *resources* and managing their allocation is probably the most common dependency, encountered in areas of activity like economics, management and organization theory or computer science. In economics for instance, markets have been observed to have interesting properties in terms of decentralization when it comes to resource allocation, both achieved through bidding/pricing mechanisms. A lot of independent decision-makers produce a globally coherent allocation of resources, by interacting with each other only locally and without any centralized control. On the other hand, there is a built-in mechanism of incentives (we already mentioned some of the findings in this area) stating that if all participants try to maximize their own benefits, the overall resource allocation becomes 'optimal'. Organization theory ties the control of resources especially to organizational power: those who control resources have 'power' (and the other way around). The theory suggests a hierarchical allocation of resources, where managers at each level would allocate resources amongst subordinates. Nevertheless, practice shows that managers may try actually to increase their own power by attracting resources away from other activities, this resulting in a sub-optimal use of resources as a whole. One way to balance this trend is to introduce costs to each type of transaction, which leads again to a market-like environment, this time on a hierarchical basis (we won't get into any details here, but there are certain conditions to make this scenario work).

A particular case of resource allocation is task assignment, especially when time becomes a resource as well; one could analyze in depth whether it is better to assign tasks based on a managerial decision, as opposed to prior assignment, pricing mechanisms or even bidding.

Let's not forget though that the area of particular interest to us is computer systems; an eloquent example here are operating systems which need to cope with memory and processor allocation, task assignment or scheduling access to other devices. Although the computer field in general may not come with significantly new models, there is room for plenty of experience sharing with other fields. Analogies with markets helped to develop new resource allocation models for computer systems; similarly, the study of distributed computing could help in its turn to better understand the evolution of human organizations (markets in particular).

Producer Consumer Relationships

Producer-consumer relationships are another type of coordination relationship extremely common whenever related activities are involved (that is, an activity produces something, which is used by another activity). [42] identifies a few types of dependencies, connected to this type of relationship:

- *prerequisite constraints* is a case where the producer activity has to be finished before the consumer can start (see for example batch-sequential processes in the SWA Styles chapter). Obviously, provisions have to be made for some kind of notification when the producer is ready; therefore, managing this type of constraints also entails sequencing and tracking of processes. In process management, prerequisite constraints are often encountered on an assembly line and many times when it comes to handle information (which has to be complete in order to be passed on). In the case of computer systems the most important aspect of this relationship is determining which activities could be done in parallel and which ones depend on the result of some others, so as to take maximum advantage of concurrent execution.
- *transfer,* in the case of a 'producer-consumer' relationship where the end results of the producer have to be carried to the consumer. In many cases one has to actually deal with physical transportation, but when it comes to carrying information, we would rather call the process 'communication'. Other interesting aspects connected to this type of dependency relate to the storage of the entities being transferred; they could be made available 'just in time', through tight sequencing (no storage being needed), or put aside as an inventory of finished items. There are studies dealing with the level of stocking necessary in order to minimize costs. To a large extent, the same problems arise in computer systems as well. In the case of parallel processing, the execution rate of processes has to be adjusted so that the producer/consumer do not overwhelm each other (with or without

buffering). Network protocols face the same problem, especially amongst communicating processes that don't share any memory; in this case some flow control mechanism is needed.

- *usability* is also an issue related to the producer-consumer relationship; whatever is produced on one side has to be usable to the other side. In order to facilitate the information flow, a standard format must be used, creating interchangeable outputs (eventually within a mutual agreement with the user, on what format is expected).
- *simultaneity constraints* are extremely common in human organizations, given the fact that humans are by nature inclined towards serialization. Computer systems are no exception from the fact that a resource cannot be allocated twice at the same time: shared memory can only be accessed one at a time, as well as processor cycles. Starting from the low granularity of executing processor instructions and going up to the process level, synchronization primitives have to be in place in order to control the sharing of data. Typical examples are the producer/consumer problem (that is, data is used only once) — or the mutual exclusion problem (which prevent simultaneous writes to shared data).

Task assignment

Task/subtask dependencies are fairly common whenever it comes to having multiple activities working together. Various disciplines deal with this aspect in their own way:

- Organization theory makes use of strategic planning, management by objectives or other ways of grouping people together
- Economics analyze the scale and scope of individual economies and global economy as a whole
- Computer Science makes use of modularization techniques in programming (or planning in AI) in order to partition the work

The most popular dependency amongst activities is when a group of activities are all subordinated to others in order to achieve a common goal; we name this *'top-down goal decomposition'*. Usually, once the goal is known (as a result of the process named 'goal selection'), it is decomposed in 'sub-goals' in the process of 'goal decomposition'. The goal decomposition techniques have been for long a topic of research in Organization Theory and many of these techniques have made their way into Computer Science. In computer applications the goal is often pre-determined (maybe with the exception of Artificial Intelligence); the

important problem becomes how to achieve a suitable goal decomposition and map it to activities that can be performed separately. Modular programming, routines, objects and so on, are good examples of structured goal decomposition processes; in AI the same thing is achieved through planning where goals are decomposed into activities based on a set of prerequisites, current knowledge and forecasted effect of those activities. The Multiactivity paradigm falls also into the top-down goal decomposition category. The result is a layered structure of managers and work units, given the decomposition of the goal (named here response) in either sub-responses or directly into activities. Managers are able to selectively direct requests to the layer below. Very similar patterns are followed by management structures in human organizations.

The opposite of the top down decomposition is the *bottom up decomposition*. In this case, one realizes that a specific group of activities (already in place) could work together, with small modifications, to achieve a new goal. This is the situation especially in intelligent systems (and again, human organizations for example) where the bottom-up approach to selecting a new goal needs a lot more involvement and commitment from the participants than the top-down delegation.

There are two main solutions when it comes to task assignment: assignment by a central coordinator or assignment by competitive bidding. In the case of the central coordinator, this is the only decision-maker to decide which server would perform which assignment. The option of competitive bidding is more decentralized and in general, follows the steps below:

1. one of the clients broadcasts a message asking for a specific service along with the required qualifications to do it
2. the servers analyze this information and then decide whether to submit a bid for this assignment
3. every server submitting a bid also submits a description of its qualifications for the job
4. the client then uses all these bid messages to decide which server will get the assignment
5. once the decision is made, the client sends again an award message to that particular server (who gets the job)

The communication overhead is obvious but the advantage of getting an optimal and flexible assignment to the job is not to be left aside. Malone used formal techniques from queuing and probability theory to analyze these two models with respect to production costs, coordination costs and vulnerability to failure. The results show that the centralized approach has lower coordination

costs, but is very vulnerable to processor failures for instance; the de-centralized model, on the other hand, is less vulnerable (and always follows low production costs) but has significantly higher coordination costs.

As stated in the beginning, there are many more dependencies to mention (if not to study), along with the coordination mechanisms associated; we thought this would be a significant subset of the most common ones. Please note that we mostly mentioned activities as being subjected to dependencies, when actually any entity could be; still, it looks like activities themselves are good enough to model coordination in a meaningful way. The bottom line here is that most concepts used by the Coordination Theory can be used to highlight similarities between different disciplines in terms of coordination, and, once these being abstracted, easily transport them across the boundaries of their original field of application

Coordination in Software Systems

After a brief introduction on the way coordination could be applied to computer systems, let us develop this topic and explore the ways it could help solve some of today's problems in SW development. As often stated, in our view, the most important of these problems is how to handle the complexity of growing SW systems in the context of shrinking schedules and resources and continuously expanding requirements. Various solution have been used so far, ranging from the ad-hoc ones (unfortunately the first being put in practice) to extremely formalized approaches (ready to yield complicated results even for simple problems). We argued that a first step would be to introduce architectural analysis as a mandatory stage in the SW development process. Once this done, some way of decomposing the complexity of requirements is needed, followed by putting the parts back together as an implementable solution[31]. It was already hinted that a coordination-based perspective to SWA in general, and the CBD methodology in particular could offer a viable solution. Both will be discussed in more detail in the following pages.

[31] We have already discussed examples of architectural patterns that have become popular (captured as 'styles'); unfortunately these are a direct reflection of rather informal solutions that have been proved so far, and not a result of a systematic approach to SWA.

From static to dynamic: another case for SWA

The evolution of SW development over the past three decades means a lot more than the evolution from mainframes to workstations and networks: we went from SW programs to SW systems. Besides shifting focus from algorithms to gross system organization, we need a clear paradigm shift from algorithmic to actually interactive systems. We have already quoted the high interaction demands put on today's SW systems as one of the main reasons for their increasing complexity. Proliferation of this type of requirements made SW design techniques once considered specific to real-time systems to now be applied more and more to general purpose products. There are rigorous logical proofs that interactive systems are a lot richer in behavior than their algorithmic counterparts (also known as Turing systems). In light of our classification of SWA styles, interactive systems are able to interact with an external environment they cannot control (that is, with events out of the initial design) at least by harnessing its power. The Multiactivity paradigm performs this task at the coordinator level by coordinating and delegating tasks between response managers, without necessarily understanding their computational details. Along the same lines with our static/dynamic view of SW systems, [61] defines interactive systems and interestingly contrasts them to parallel and distributed systems. Paraphrasing this paper

- *interaction* occurs when dealing with an environment the system cannot control
- *parallelism/concurrency* occurs when computations of a system overlap in time
- *distribution* occurs when components of the same system are logically or spatially separated

The argument here is that in the absence of interaction, both parallel and distributed computations can be reduced to algorithmic computations. As a result, the key to a behavioral view on system composition is interaction rather than parallelism or distribution. This comes in contradiction with the popular belief that associates the last two mostly to real-time or intelligent applications.

The most important consequence of the discussion above is the fact that we must give up the unrealistic goal of completely specifying the behavior of a system[32] all at once (which may take a psychological adjustment as well). The acceptance of such incompleteness makes for a very important step in SW

[32] Many times, a source of incomplete specification could be the requirement specifications themselves; it often happens that even the user does not know precisely and ahead of time what are the required characteristics of the system.

engineering, which could be compared to the large acceptance of empiricism at the expense of rationalism in philosophy, mathematics or physics. It is worth noting that many of the established engineering disciplines we referred to in this thesis (i.e. building or mechanical engineering) have not been faced (yet) with this type of decision. Abstraction is a key ingredient in coping with system complexity; especially on the architectural level of SW; we want to focus on a subset of the relevant attributes and ignore the irrelevant ones. In this context, incompleteness is both a principal consequence but also a mechanism between formal (rationalist) and non-formal (empiricist) abstraction. Completeness is only possible for a restricted class of comprehensible systems, where the appropriate semantics to describe them are in place, and where behavior could be safely restricted to a subset describable by algorithmic rules. On the other hand, incompleteness is the price paid for modeling independent domains whose semantics and behavior are far richer that the ones that can be modeled; this is the case of most complex systems. One can use incompleteness as the key criteria to distinguish between an algorithmic and an interactive model of the world. The SW community has already taken important steps in this direction through AI systems, based on the premise that a combination of algorithmic and interactive techniques are closer to human behavior that the algorithmic ones. Other emerging concepts along the same lines are anthropomorphic programming and symbiotic systems.

If not able to describe the entire behavior, we should at least be able to specify parts of it; complete specification has to be replaced by partial specification of uses or views. On the design level this is being handled by use cases (especially in OO); at a higher level, this is usually done by specifying interfaces, as already discussed when we introduced SWA. Interfaces play the role of harnesses in their dual role of containing system behavior and also direct it to useful purposes. Incomplete specification fits SWA very well since SWA seeks to flesh out unimportant details from a system's specification and focus on the relevant properties of the system. Multiactivity in particular achieves these goals by dividing the system behavior in system responses, with the response managers acting as interfaces. In this case, behavior specification through responses translates into system specification through response managers.

The important paradigm shift from algorithms to interaction comes as a focal point of similar trends in the study of human-computer interaction, SW engineering and system architecture by and large. As often mentioned before, SW development evolved from small scale programming dedicated to perform purely scientific computations (especially in mathematics); due to the formal nature of these tasks, they lent themselves very well to an algorithmic approach. With the penetration of SW in virtually every area of human endeavor, the

increasing demands and expectations from SW products, this shift from algorithms to interaction identifies computer science and especially SW engineering as a distinct disciplines from mathematics (which they evolved from). At the same time it provides the rationale for calling computer engineering by and large a science on its own.

From formalism to empiricism

There is a general intuition that empirical models are a lot more expressive to be understood than the formal ones (although the latter might be better at stating and proving the problem). It seems today that SW implementations in particular have become the language of choice for modeling. They allow applications from various fields of science or engineering to be modeled and *expressed* in a common language, so that we are able to talk precisely about more abstract concepts. Since most of the modeling takes place at the architectural level of the particular application, attempting to formalize it at the expense of expressiveness simply defeats the initial purpose of having a model.

There are always tradeoffs to be made between formalism and expressiveness in many other disciplines, but especially in computer science. Restricting ourselves just to the latter, we strongly believe that promoting formal methods on the architectural level of SW design would adversely impact the entire design process and the quality of the final product. Programming in the large in general is highly interactive and cannot be reduced in any way to algorithms, which are usually characteristic to programming in the small. On the contrary, SWA being one level of abstraction above design, it could enhance (and simplify) algorithms with interaction, leading to what can definitely be called 'smart systems'[33]. Overemphasizing formalism in SWA is most often done at the expense of expressiveness, which is one of the reasons for introducing SWA in the SW development process.

Coordination and Software Architecture

Very much like SW engineering which gradually evolved from programming languages and algorithms to program composition and later SW architecture, coordination theory made its entrance to SW engineering once again from the language perspective. The broad opportunities opened by massively parallel

[33] Algorithms are considered to be inherently 'dumb' for not being able to dynamically adjust the results of their computations.

systems called for new ways to deal with concurrency and cooperation of a very large number of SW entities working as a single application. Classical views of concurrency, using standard programming languages (basically extending the sequential programming paradigm), were not enough. Therefore the interest shifted towards languages in order to enhance modularity, reusability of existing components, interoperability and ultimately support for coordination models. These languages were able to exploit parallelism at different granularity levels (parallel statements, objects, etc), perform communication (by message passing or shared data) and above and beyond, be fault tolerant. After exposure to a variety of applications, it soon became clear that no unique language was able to deal with the number of facets involved in developing a complex system: composition (and heterogeneity), scalability, reuse and many others.
Consequently, coordination languages for massively parallel applications remained in sort of a niche market, given the fact that these systems are exclusively suited to algorithmic and not at all to interactive computations. The latter continued to be developed with no explicit (that is, embedded in the development environment) support for coordination, most commonly used languages having to rely at least on the facilities offered by the operating system, if not construct a coordination infrastructure on its own. This last fact ultimately led to an increasing interest in coordination theory within the SW community. We think coordination is crucial, especially on a large-scale perspective, being one of the key handles to manage complexity.

The heterogeneity issue mentioned above fostered the development of 'multi-paradigm programming' able to support interaction between multiple programming paradigms and at the same time isolate unwanted interactions The multi-paradigm approach to heterogeneity in SW systems had important ramifications to coordination. On this particular occasion, two avenues were basically followed:

1. Create a meta-language to integrate heterogeneous SW entities
2. Provide appropriate interfaces to the heterogeneous entities so that they can communicate with the other

The second solution is the one that was adopted on higher abstraction levels of SW design, starting with module interconnection languages, out of which MIL75 is best known. Over the years, these models evolved up to widely accepted interfacing standards (like CORBA), versatile architectural styles (like the Blackboard style) or looser paradigms like the actor model.

When we discussed SWA styles we already expressed a concern that most of the styles commonly used today do not consider coordination at all. Coordination

seems to be indispensable when it comes to addressing issues with respect to the development of complex systems (not to mention distributed systems). In this end, any SW system could be seen as a combination of two activities:

- The *computational* (execution) part, incorporating the core functionality of the system
- The *coordination* part, which takes care of the communication and cooperation between the components of the SW system

Most of the SW systems developed today do not make such a distinction and the two activities are found intimately interleaved within the system components. In contrast, our design methodology sets out from the very beginning to keep these two activities separated throughout the entire design process. As we already mentioned, the concept of coordination is closely related to the one of heterogeneity. By keeping the coordination component separate from the computational one, the former sees the latter as black boxes; therefore heterogeneity amongst the execution components is actually encouraged from the programming language up to the architectural level.

In the previous chapters we have already defined coordination in general; on top of that definition one could define a coordination model as 'the glue that binds separate activities into an ensemble' [31]. Coordination models can be further defined in a similar way to the one [50] did it for SWA, that is, as a triple {**E**, **L**, **M**} where

- **E** represents the set of entities being coordinated (system components essentially)
- **L** the coordination media (shared abstractions, SW bus, data streams, etc)
- **M** the semantic framework the model adheres to (event-based state transitions, I/O abstractions and others)

Of course the SW community has come up with more coordination and configuration description formalisms in this area. It is not our intent to go into details on how coordination could be formalized[34]. Instead, we prefer to take a

[34] [46] offers a comparison between formal coordination models based on:
- entities being coordinated
- coordination mechanism
- coordination medium
- semantics, rules or protocols for coordination
- use of a separate coordination language
- application area
- degree to which the framework is integrated

more liberal position[35] and see coordination as dealing with architectures in terms of *components* and *connectors* (in line with our previous descriptions of SWA). At this stage one could say that such a perspective on SWA is a first step in the direction of separating coordination from execution in a SW system (in this case distinguishing the computational component from the structural one). In our view of coordination though, when referring to SWA we find it more meaningful to at least separate interaction from execution, thus coordinating the interactions between SW modules.

Discussing SWA styles, we based our classification on the locus of coordination and considered the central and distributed case. However, now that we discuss coordination in more detail, it is definitely worth to mention a more versatile classification, which takes into account the driver for changes in the coordinated processes:

- *Data-driven coordination* where the evolution of computation is driven by the properties of the data involved in coordinated activities
- *Control-driven coordination* where the evolution of computation is driven by events explicitly requesting for a specific change in the state of the coordinated activities

Of course, there are more criteria to classify coordination models, some quick examples being the kind of entities being coordinated, the SW architectures assumed by the models, the important issue of scalability and openness, and others. Nevertheless, as [46] suggests, the great majority of coordination models could fit into one of the two categories above.

Data driven coordination models

In the case of the data driven coordination models, the state of computation is determined at any moment in time by the values of the data being exchanged, correlated with the state of the processes exchanging this information. It is up to the participating processes to either coordinate themselves or coordinate other processes.
Most of the data-driven coordination models are centered on the concept of 'shared data space'. Processes involved in the computation can only

---

The literature considers even more criteria.

[35] We will see that some degree of formalism is still present in our coordination-oriented development environment introduced later in this thesis. That formalism (in the form of PAL expressions [2]) is used for design validation purposes rather than decision making.

communicate indirectly through this shared data space, which in a way ensures their independence. One would say processes are de-coupled

- *in space* since inter-process communication is only done via the shared medium
- *in time* since the content of the shared data space is independent of the history of the surrounding processes

Around these two types of separation we also note the fact that a certain process does not need to know the identity of its interlocutors in order to communicate to them. Also there is no need for synchronous retrieval of the data; this could be read at any time after it has been made available. We tend to see this type of coordination as rather distributed (and in some sort of permissive way), since coordinated processes usually make their own decisions; nevertheless resource management is centralized.

The shared data offers a lot of flexibility in terms of the structure of the data that is being exchanged, ranging from flat to nested records. There is also flexibility for restricting the coordination flow by letting specific processes to coordinate others and not the other way around. Linda [31] is the most popular coordination model in the data driven category and historically the first in the family of coordination languages. Let us note though that not all data driven coordination models share the data in a common repository; message passing could also be used, still coordination remains driven by data in the sense defined above.

Control driven coordination models

The control driven type of coordination is the one of choice for most of the SW systems. In this case, the state of the computation is decided not directly by the state of the data (the actual values of the data are not involved), but by a certain coordination pattern defined previously. Control driven processes evolve their framework by reacting to state changes of their components or event receive. In contrast to the case of data driven coordination where the components were actually looking at the data values in order to make decisions, components here (coordinators or computational modules) are seen as black boxes; from a coordination perspective, data is of no concern. Inter-process communication is usually point-to-point (eventually limited broadcasting) and is performed through well defined interfaces (input-output ports). Once again, the coordination flow is seen as bi-directional, any components being able to send

out control messages or events to inform the outside about their state. Of course there is room to restrict the flow of control information (therefore the roles of components) nevertheless this is an important difference from our view on coordination which advocates a strict separation of coordination from execution. In terms of functionality, [46] describes the relation between the coordinator and its coordinated processes as a producer-consumer relationship.

The main difference between these two types of models is the degree to which coordination is kept separate from computation. In this respect, the control driven models seem to do a better job than the data driven ones. Despite not necessarily restricting the coordination flow amongst components, data does not get involved with coordination in the control driven case, which (stylistically) means a separation of concerns between coordination and computation. This could be brought to the extent where a separate coordination language is used, which sees units of work as black boxes; it is easier this way to separate components into coordination and computational ones. However, we think this separation is not enough; the fact that any modules are able to exchange control information back and forth and could make their own decisions based on event observation, does not lead to a clearly separated coordination structure. As for the data driven models, the fact that any process is capable of both manipulating/analyzing the data as well as coordinating itself or others does not indicate once again a clear separation of coordination from execution. At least from a stylistically/syntactical perspective, there are a number of coordination primitives which are embedded with the computation code, and lead to an amalgamation of computational and coordination code. Usually, processes cannot be easily tagged as purely computational or associated to the coordination process; it is up to the programmer to make a clear distinction between the two aspects and implement its own coordination framework.

In terms of the application area, each of the two seems to have its own field of expertise; while the data driven models seem to be mostly used to applications which require a high degree of concurrency, the control driven models tend to primarily be used for ...modeling. This is a reflection of the fact that the first category is suited to process data, whereas the second is better at coordinating SW entities. To conclude, the two coordination paradigms discussed above are definitely useful for a better understanding of how coordination could be applied to SW systems. Both offer great hints in identifying and separating coordination from execution; nevertheless, a lot in this direction is left up to the programmer and additional coordination frameworks are needed on top of these two models.

Coordination vs. execution in Software Architecture

SW production is still confronted with build-from-scratch development, in contrast to other mature engineering disciplines where (at least) routine problems are reliably solved by reusing existing designs to a large extent. The end result is that designers are able to build relatively complex, and what is most important, high quality systems out of existing, previously proven parts. There are two reasons why SW lags behind when it comes to reuse:

- Difficulty to *identify and locate* the appropriate components, for still missing desired component libraries or SW design handbooks[36]
- Interoperability *mismatches* at low level (different procedure names, data types, parameter orderings), going to the high level of SWA. (usually, different assumptions on the structure of the application components are integrated into)

The second issue is the most serious since it is hard, if not impossible to overcome. In spite of its seriousness, to date there was no systematic attempt to establish techniques to deal with architectural mismatches; designers still rely on their own experience and intuition in an ad-hoc manner in order to confront it.

In light of our initial assumption that on the architectural level, a SW structure is basically comprised of connectors and components, we think that components and their interdependencies should be separated from the very beginning. If in a regular[37] SW architecture components aggregate the core functionality, interdependencies amongst components should be considered a concept *orthogonal* to the problem domain[38]. SWA mismatches are directly related to the failure to consider component implementation orthogonal to component interconnection and coordination; most of these mismatches are generated by important coordination assumptions that are embedded into the modules. By doing so, the possibility of reusing these components to other applications is being drastically reduced. On the other hand, there are significant benefits from

[36] It is true SW development environments come to alleviate this need with own function libraries; however, these promote reuse on a very small (code level) scale and are not necessarily interchangeable.

[37] We do not discuss here massively parallel architectures.

[38] Research within both MIT and Carneggie Mellon Computer Science departments shows that most SW interconnection requirements can be expressed (if not implemented) through a specific set of concepts independent of the problem domain.

keeping these two design aspects orthogonal; beside the opportunities for reuse one could definitely mention:

- Solid grounds to keep system development under control; coordination (or the lack of) is one of the main ingredients to system complexity and a principal source of unpredictable system behavior. By making coordination orthogonal to computation, the two gain at least equal visibility starting as early as the architectural design stage (moreover, focus could be easily shifted to the one whose constraints are harder to meet)
- The complexity of coordination is not enhanced by computation and the other way around
- Dependency management could become routine as soon as design frameworks for coordination are being developed. As a consequence, initial application development can experiment with and choose the most suitable coordination model
- Application maintenance is favored by easy replacement of components with updated implementations; the effect of maintenance is localized around the replaced component, with no impact on the initial architecture
- Portability is also encouraged, especially if making dependency management a routine activity (as much as possible); even though different coordination processes might be required, the basic architecture remains unaffected.

There is no widely accepted methodology yet, to partition a particular application in an orthogonal way[19]. Nevertheless, [20] comes up with some hints on estimating how much a specific module is purely computational:

1. Interaction with the environment is only done through input/output ports
2. Every port is (could be) independently managed in terms of resource availability
3. An arbitrary number of clients can be connected to any port
4. No assumptions are made inside the component with respect to resource access or ownership; any such issues should be handled outside the component

In a case where some of these requirements cannot be achieved or unwanted functionality has been inherited, components could be augmented with wrappers to achieve desired properties.

---

[19] [20] suggests an iterative one, used in conjunction with a coordination design environment; in this case, activities and dependencies are successfully refined so that all activities are only associated to purely executable components and interconnections can be exclusively managed by coordination mechanisms.

There are basically two reasons why activities need to communicate with another; both could be considered as based on resource use:

- Some activities use resources (i.e. data) produced by other activities
- There are activities that share resources with other activities

Without going into the details of dependency classification, let us mention that the more complex case where resource users are independent offers two other typical sub-cases of activities *competing* or *cooperating* for the same resources. The latter sub-case embodies the more popular timing dependencies in the form of either *mutual exclusion* or *prerequisite*. If it does not look already complicated, let's only add the fact that each case above could be further subdivided by the number of activities interacting, reported to the total number of activities (that is: one-to-one, one-to-many, one-to-one-of-many and so on). This discussion lets us point out some ideas on how to represent and classify coordination processes. There are two sets of questions to be answered here (basically characterize coordination by means of dependencies):

- the degree of generality of the coordination processes
- the type of dependencies we are dealing with

The latter deserves more attention, with a special focus on the cause of dependencies, possible ways to organize (or identify common ways to manage them) and not at least, track them as the system evolves. One could always go into more detail with analyzing the specific dependencies.

[30] advances a taxonomy of interconnection assumptions considering two dimensions: the *design level* and the *location* of the assumption within the component. Accordingly, design level interconnection assumptions can be identified as

- assumptions that are found at the design/implementation level, qualified as *design assumptions*; the common example here would be protocol assumptions
- *architectural assumptions*, a lot closer to the specification level; an example would be modules designed with a flow architecture in mind which later cannot be reused in a call architecture

Location assumptions are more specific and involve interfaces, could very well be embedded in the code or be implicit (that is, entirely rely on the communication properties of the environment the SW was initially placed in).

Today's SW development practice confirms the fact that, as we get closer to the implementation stage in the design process, current design and programming tools tend to focus increasingly on components, leaving the coordination and communication aspects out of the picture. Support for complex module interactions is buried in the semantics of programming languages or operating systems or, even worse, is fragmented in order to be embedded into the computational modules.

Concluding remarks

The purpose of this chapter was to provide an introductory view to coordination, starting from its dictionary definition, going through the way it is reflected in various research areas, exploring coordination patterns and ending with an emphasis on the domain we are interested in – Software Architecture.

We consider coordination a major player on top of SWA which, as often stated, emerged as a central concern in order to make today's complex systems intellectually tractable and exploit successful patterns of system organization. The main argument in this chapter was that as early as the architectural step in the SW development process, coordination aspects should be considered orthogonal on the computational ones. In order to center a specific design on coordination, the most important issue is the identification, synchronization and communication of concurrent activities. The traditional approach to SWA is based on functional or object-oriented decomposition criteria, leaving important issues as the ones mentioned above to be dealt with at implementation time. This view also builds on our earlier assumption that components and connectors are the major abstractions to describe SWA, considering components as representative for the computation part and connectors mainly associated to the intercommunication/coordination aspect. We think coordination deserves at least as much attention as the execution part on the architectural level. If decisions regarding the computational aspects could be deferred to the design level, the coordination structure has to be decided early, along with the architecture.

Another important argument we brought forward was the fact that no interconnection and coordination assumptions should be built into the computational modules. Current programming languages do not offer separate abstractions to represent component interconnections, not to mention coordination. Therefore these mechanisms are implemented together with the core functionality, deferred to the operating system and often disseminated within the execution modules. Actually, these arguments are tightly related and follow

one to the other; separation of coordination from execution is the main prerequisite to leaving any interconnection assumptions out of the execution modules. The conventional view on module interconnection is one where connectors are considered passive entities (solely used to transfer information from one end to another). With interconnection specifics being migrated out of the execution components, connectors become more complex, incorporating additional functionality ranging from simple storage to decision making.

We strongly subscribe to the arguments brought by the Coordination Institute at MIT (T. Malone and others) that an interdisciplinary study of coordination is needed in order to properly apply its principles to computer engineering by and large (in fact, the same approach was recommended for SWA). Once again, it is our belief that the key to managing complexity in large SW systems is to successfully address their coordination issues. Based on the principles discussed here, we will introduce Coordination Based Design as a first step towards a coordination-based approach to SWA and then take another look at SWA styles.

Chapter 4

CODE: A COORDINATION ORIENTED DESIGN ENVIRONMENT

In the beginning of this thesis we have made a case out of considering SW as a product before anything else. By and large, providing quality products consistently and at reasonable cost (design, implementation and maintenance) is the basic goal of engineering. In this context, the need to integrate SW development along with established engineering disciplines has also been outlined. We have also started by emphasizing the need for quality in SW. Even in relatively simple products it is recognized that one has to design for quality from the very beginning; for any reason, quality cannot be considered a subsequent activity. In the case of complex systems, usually developed concurrently by multiple teams, quality (or the lack of it) becomes even more visible. Therefore, the entire development process, the methodology and system architecture must provide for quality assurance. All of them must plug the holes were quality may be lost; a frequent example is the case of multiple design teams where gaps occur most often at the interaction of the various teams or in the case of system integration. Quality has many facets, some of them purely qualitative in nature, others being measurable. These facets are all highly interdependent and contribute to various extents to the overall perception of quality; it is a fact easily seen in complex systems, where the various components (modules) generally have different quality requirements, therefore emphasizing their own dimension of quality. Part of this could be seen as a consequence of the major need to satisfy a large variety of constraints, a characteristic of properly engineered products. Let's not forget that, when we discussed quality, we have made a distinction between quality attributes that are directly related to the application requirements and attributes required by a well-engineered product in general. The methodology and system architecture must facilitate the specification and balancing of both functional and non-functional requirements and constraints.

CODE: Motivation

To a large extent, the development effort in present-day computer systems is geared towards the SW aspect ('SW engineering'). This is a rather biased attitude; it is true that SW has to be engineered and not hacked, and it is true that HW today (especially the data processing one) is both inexpensive and reliable to accommodate applications spanning a broad quality range. However, in a lot of applications (control, embedded, event-driven) one has to develop a complete computer-based system, in which the HW component (processing, I/O, networking) cannot be ignored. Another aspect is related to the broad expertise needed to develop a complex system. Computer-based systems are a good example, being inherently quite complex; one needs various levels of expertise to properly develop such a system and in this context SW engineering is only one of the areas of expertise required[40].

It seems to be largely recognized that one cannot handle complexity without taking into account modularization and making use of components. Nevertheless, the simple use of components is not a guarantee of a successful product; the success achieved once in making them work together has to be reproducible. Components have to be defined and designed in a way that allows for easy assembly; modules have to be able to comply to the requirements of a manufacturing-like process, so that less expertise is needed to 'handcraft' and match them together. Furthermore, this has to hold at all levels, from system modules, to components and parts. Nowadays, short development cycles, due to the reuse of existing components with minimum expertise, are another important step of engineering a product. SW has been looking at modularization for a long time; starting with the early attempts of information hiding and separation of concerns, procedural programming, then OO and patterns, all these attempts were targeted at both reducing overall system complexity and enhancing reuse. The side effect was that with encapsulating an increased abstraction level and complexity inside the modules, it becomes a lot harder to control their interactions and ultimately reuse them. One actually requires significant expertise to develop quality SW from these more complex entities.

CODE attempts to overcome the above problems in computer systems by using a hierarchical management model similar to the models in complex human organizations. In this end, by coupling management with coordination, the

[40] In the same line of thought, if one considers the development of a ship or airplane, would not leave the entire development in the hands of a metallurgist since both are made out of metal, or to an engine designer since both involve a number of engines.

interaction between system components becomes easier to handle. One does not distinguish anymore between HW and SW, the two being now considered as resources assigned to a specific response.

CODE: Approach and Overview

The Coordination Oriented Development Environment (CODE) is an extension of the work on Coordination Based Design and its associated methodology [2]. It is believed to make a difference in contrast to other present-day development methods which as a rule, do not cover all the development phases throughout the product life cycle and are not necessarily consistent from one development level to another in terms of methods and tools. CODE extends and integrates the phases of the development process (from requirement acquisition to implementation, delivery and further to product retirement) with matching methodology, system architecture and modeling[41] tools. Most methodologies so far are not that concerned with the modification and upgrading of large SW systems, once they go out the door. The fact is that a significant part of the design activity is devoted to modify/upgrade existent SW systems, rather than develop new ones from scratch. As a result, a large majority of designers spend most of their time trying to understand their own part of the code; most often, the overall picture of the system is neglected, producing serious side effects [49].

Basic principles of Coordination Oriented Design

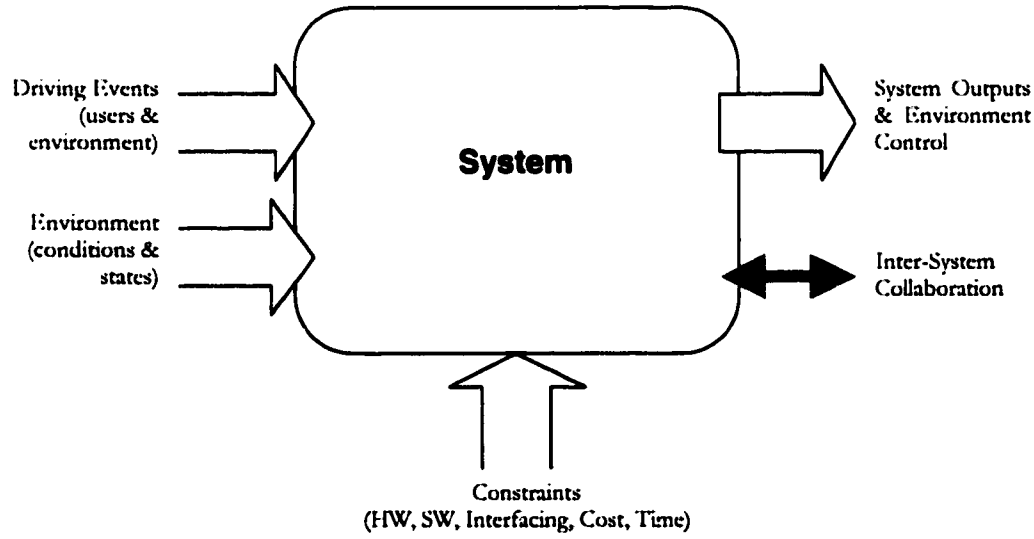
COD is an integrated development environment with a well-defined development process, matching methodology, system architecture and a tool-set specified for the development of complex, critical, and multi-user computer based systems.

- The *development process* corresponds to a set of well-defined rules and guidelines that help the designers and implementers to identify the various choices needed to craft a well-engineered solution to a specific problem. As such the development process, which involves a number of distinct phases, could be looked at as transformations between various viewpoints (models)

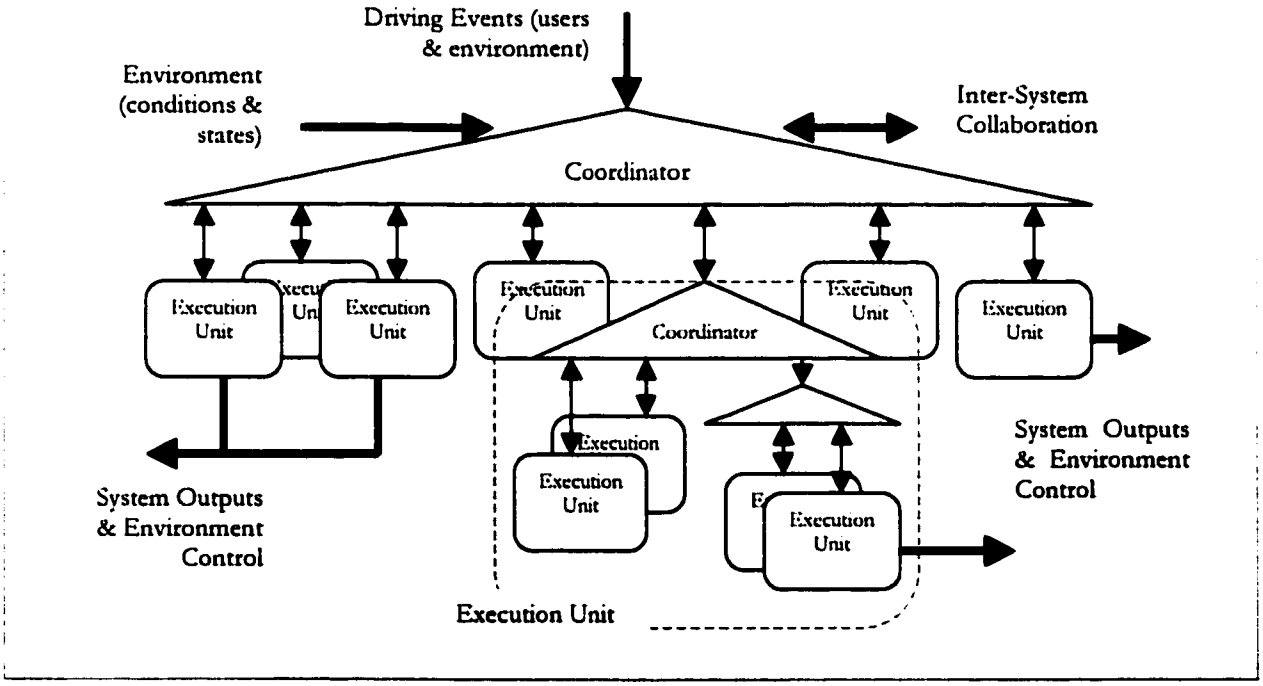
[41] The CODE architecture diagram offers the first modeling level; a lot of reasoning could be made here, where one could shuffle things around to find a better solution. The validity of the solution could be then verified via PAL/PAD tools, which are part of the development environment.

allowing the developers (architect, designers, implementers, testers) to proceed from a problem-oriented model (the 'what'), through a set of solution-oriented models (the 'how'), to implementation.

- The *methodology* corresponds to a set of unifying principles needed to describe, analyze and make decisions about a specific system. Therefore, it must rely on a set of basic 'concepts' that characterize the system and a 'language' (or notation) to define/describe the properties of the system in terms of these basic concepts.
- The *system architecture* basically corresponds to a high level structural and behavioral description of the system, which allows for system level reasoning. As such, it serves as a blueprint for the users to validate system requirements and for designers, implementers and testers to understand the overall system without having to go into the details of each of its modules.



The methodology in CODE uses a behavioral view that characterizes the system as being event-driven; the system executes 'work' as a response to events that drive it. Furthermore, the work associated with a system response is partitioned into relatively independent units of work called 'activities'. In other words, each response is viewed as a set of activities which have to be executed in a given precedence relation (represented by a precedence graph) to provide the required outcome to an event. As a consequence, in specifying the event responses, *execution* (what has to be done, reflected in the units of work) is separated from *coordination* (when to do it). Note that the overall system specification is performed through stepwise refinements. The specification as a whole is definitely top-down however, there could be optimizations that are made in a bottom-up fashion.



Furthermore, the methodology views the computer system that has to be implemented as a set of *assignable resources* (software objects/modules, hardware

components and interfaces), used in an orderly manner to execute the activities (that is, the units of work) that configure the responses of the system.

The corresponding architecture views the system as one or more coordinating units ('coordinators'), specifying the sequence of activities that has to be initiated in response to an event, and a number of 'execution units' able to execute the different activities of the various responses. Note that the functional view of the system (referred to as a functional diagram) corresponds to an *outright separation of concern between coordination (management) and execution (work).* In contrast to making this feature a requirement of the architectural design (as recommended lately in the literature), the separation rather comes for free as a natural result of describing the system behavior in terms of responses. In the above organization, the lower level coordinators receive from the environment only conditions or states (not events, since these are already part of an event response), which only affect the lower level execution[42].

Before going any further, it is important to clarify what are the boundaries of the system under consideration; every time a design task is approached, it makes engineering sense to be able and define the scope of the approach. One could argue that a development process could be scaled to any level, especially in the case of distributed systems. We consider this an unreasonable claim, if one has to meet constraints on various working conditions and provide meaningful quality assurance. In the case of our particular design methodology, the need to define the boundaries of the system becomes even more important when it comes to distributed and especially with single coordinators; therefore, we will consider the following cases:

- If the system under consideration involves a number of subsystems that are highly coupled and/or a common critical resource that has to be managed, it is indicated to combine them into a single system by defining a higher level coordinator over them[43]. Nevertheless, the systems may include their own critical resources to be independently managed.

---

[42] Suppose, for instance, the coordinator assigns the task of guiding a robot from point A to point B to an execution unit. That execution unit owns the response until it is carried to completion. Meanwhile, the low-level coordinator may be involved for example in avoiding an obstacle on the way, in which case it receives conditions and states about the obstacle and makes decisions without the involvement of the central coordinator.

[43] We will make the same call as part of the Case Study, when analyzing the Hospital Information System.

- Conversely, if the level of coupling/communication between subsystems is low, they should be developed separately with a well-defined protocol of communication amongst them[44].

An overview of CODE development phases

Turning to the development process associated with CODE, at the highest level, it consists of six phases. Essentially, these development phases are along the same lines with the SW development model described at the beginning of this work (requirement acquisition, architecture, design and implementation).

A. The *architectural phase* relies heavily on the so-called 'requirement engineering'. It includes both *requirement acquisition* (by and large) and *architectural design*. On a second thought, one could actually rename the requirement acquisition phase as *'requirement management'* and divide it into two closely related steps:

- requirement *acquisition* per se, where the architect, along with the user, come up with a large set of functional requirements as well as constraints, various options to be considered and so on
- requirement *specification*, where the initial set of requirements is restricted to the relevant ones, clear deliverables are set and eventually requirements get a more formal description

At this stage, the requirement management and architectural design are closely coupled, since for a proper specification of requirements, it is not enough to know 'what' has to be done but also 'why' there is a need to do it and what is the associated cost of doing it.

B. The next step consists of incremental *design phases*, involving multiple design teams. The use of coordinators and executors allows for separate design, implementation and deployment of specific system functionality. Both system partitioning and module integration should become fairly straightforward once they are done separately at the work unit level, for each execution unit. Given the incremental character of the design phase, coordination facilitates a vertical development dimension so-called 'design by refinement' (adding details) as well as a horizontal one called 'design by extension' (adding new functions).

---

[44] In the case of the HIS, this would be the case of the patient information system on one hand and the administration information system on the other hand.

C. The *implementation phase* involves a number of implementation teams as well and has a recursive character. Just like the design phase, modules could be implemented one by one, this being at the same time a consequence of the clear separation of concerns between coordination and execution as well as a result of specifying functionality in terms of event responses. The use of coordinators and executors also allows for a straightforward integration and testing of various functionality and constraints.

D. Deployment phases involve both piece-wise deployment and version upgrades. The major improvement brought by our strategy in this area is that delivery could start very early in the development cycle, as opposed to most other development processes. Since functionality is so clearly separated, the developers could consider a phased delivery in terms of:

- *system responses*, in which case a version upgrade may be necessary given the fact that changes in terms of system responses usually involve changes to the coordinator
- lower *work unit* levels, in which case an upgrade may not be necessary, communication from the coordinator to the missing work units just being bypassed

E. The *maintenance phase* includes both standard maintenance and upgrades. With respect to this phase of the product life cycle, our coordination-based architecture allows for easy maintenance or replacement of faulty modules and eventual extension of functionality, all with extremely localized effect on the overall system. Moreover, by instrumenting the system at the coordinator and execution unit interfaces one could readily detect faults or introduce various types of fault tolerance mechanisms[45].

F. As the requirements, usage, technology or interfaces evolve, there is a point beyond which upgrades become impractical and parts or the entire 'old' system must be turned into a new one. Usually the new system is being developed in parallel, for a smooth migration between the two. In the case of our coordination-based architecture, the *retirement phase* could be performed piece by piece, analogous to the delivery. The two systems still interact during the transition, by providing services to each other via their highest level coordinators.

---

[45] **This is another aspect that was highly emphasized in the Hospital Information System case study.**

Last but not least, CODE could also make a presence in the field of DSSA. As a first approximation of such applicability, the following modifications could be made to the development process:

- The architectural phase is being replaced by a 'sales' phase. Once the specific area of application has been properly understood one does not need to architect a new system, but only customize an existing design to more specific user needs. After identifying generic requirements in the particular application area, a set of domain-specific PAL expressions is presented to the user (via the PAL simulator). Possible modifications are outlined and specific user constraints are discussed. After a number of such meetings between users, sales people and designers, the specific features to be delivered by the system is finalized as a set of cohesive PAL expressions. This now becomes the contract between the users and the developers.
- The design and implementation phases are now collapsed into a modification and integration phase. As such, the various coordinators are developed and the resources needed are integrated and modified so as to meet user needs. At this stage one also defines the required system instrumentation and maintenance as well as executing system testing.
- Deployment now corresponds to transferring the system on customer premises and executing acceptance testing.

We would like to stop here with the overview of CODE principles and development phases. Since a more detailed approach to Coordination Based Design has already been taken in [2], we set out to now go into more detail with respect to the CODE development process. Before proceeding though, on a general note, let us mention that this process cannot be assimilated to a waterfall model for at least three reasons:

- Each phase may include a number of sub-phases and generally requires a number of iterations and checks. Some of the possible checks are review meetings, inspections and testing.
- Each phase has a set of well-defined deliverables, with specified time-lines.
- Consecutive phases are usually interleaved and interdependent.

Coordination Based Design in general is also a step forward towards the component-based design of computer systems. Research on the COTS topic has already been underway for almost two years by now, with concrete results outlined in a number of graduate works under the same supervision.

The Architectural and Design phases in more detail

As previously mentioned, the architectural phase in CODE includes two interdependent activities: *requirement management* and *architectural design*. We have also divided the requirement management phase into a requirement acquisition phase and requirement specification. Requirement acquisition starts with the system architect obtaining from the 'high level' users/clients (managers, administrators, etc.) an overall (and often informal) picture of the significant events and activities in the environment where the system will be placed. Based on these exploratory discussions, the system architect evaluates the critical resources to be managed on which based, he gets a feel of the primary events and the high level constituents of the system. It is important to emphasize once again that at this stage the architect defines the high level architecture based on the critical component that has to be managed. This first draft of the architecture is presented to the users for validation and eventual approval. At the same time more specialized users of the various modules could be identified. In a case where the first draft was approved, the architect meets with different user groups and the architecture is reviewed, along with the integration and required interactions between the modules[46]. At this stage, one could also define the order in which the modules would be developed and, in the case of piece-wise delivery, deployed.

For each module a new requirement acquisition phase is initiated by eliciting from the more specialized users/clients the primary events of the module, a rough behavioral specification of each event response and the relationships/dependencies between them, along with a listing of the various constraints. The constraints could be

- *component* constraints (hardware, software, etc.)
- *quality* constraints (as a minimum: usability, maintainability, efficiency, reliability)
- *performance* constraints (timing constraints, number of events that have to be handled, etc.)
- *interface* constraints (user interfaces, interfaces with other systems, etc.)

Based on the data received from the users, the system architect develops a rough requirement specification as a functional diagram in terms of coordinators

[46] Smaller systems may not have multiple 'high grain' modules, so one could proceed immediately with the module architecture.

and executors and matching/corresponding formal event responses expressed as Process Activity Diagrams (PAD) or Process Activity Language (PAL) expressions [2]. These requirement specifications are refined and expanded via a number of iterations during review meetings where the system architect acts as the presenter and the users/clients as reviewers.

As previously mentioned, in many information systems the definition of need is vague and open-ended (such as a 'hospital information system to improve patient care'), providing only guidelines and not true requirement specification. It is the role of the system architect to firm up these guidelines during the review meetings by presenting different possibilities along with the implications and costs associated[47]. Still part of the requirement specifications, the architect and the user group will decide on the delivery strategy, that is the sequence in which the product is to be implemented and delivered.

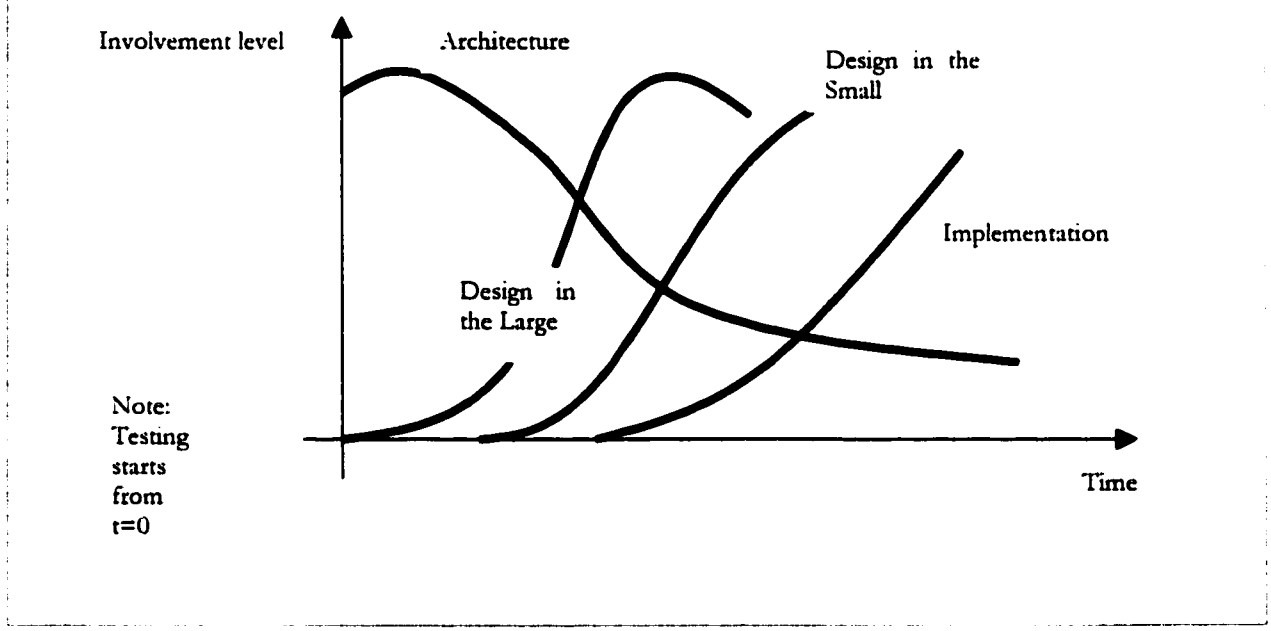
As a final phase of the requirement specifications, the event responses are refined until one reaches the smallest response component/unit of work of interest to the users. We call these units of work 'activities' and any response would consist of at least one (if not a few) activities. Moreover, for larger systems there might be a need for an additional layer of combined responses; in this case, we call *elementary responses* the ones made up only of activities and *combined responses* the ones consisting of two or more elementary responses. The corresponding PAD or PAL expressions as well as the functional diagram, annotated with pertinent constraints, are used as the specification document that is inspected for errors by each user in the group, first separately, then in another review meeting [48]. Existing PAL-based tools such as editor, verifier and simulator could offer an important support in such meetings.

After a few review meetings with the user and based on the event responses agreed upon, the system architect partitions the system into implementable modules, which correspond to the execution units of the high-level functional system view. At this stage, one could also specify the design groups required. Once again, this partitioning is done first according to the critical resources that have to be managed in the implementation and then according to the design expertise required. The system architect then prepares a modified set of event responses and annotated functional diagrams that represent requirement specifications at a level of detail that is suited to the design community. Usually this represents a level of detail/refinement above the one of the user

[47] Depending on the user community one could have a single representative user group or a number of different user groups.

[48] It is assumed that by this time the members of the user group would have obtained the expertise needed to inspect the document.

community; nevertheless, a direct mapping between user and designer oriented requirement specifications is imperative. As a follow-up, the system architect then refines these requirements through a number of review meetings[49], this time with the design group leaders acting as reviewers.



The final product is an architectural definition that serves as a blueprint for the development of the system; it allows various module design groups to understand the overall system (as we mentioned, allows for system level reasoning), without developers having to understand the details of other modules. With this architectural definition, CODE goes beyond the gross system structure, as represented by the functional diagram on the architectural

[49] At this stage, the various system level user constraints are translated into more detailed module level design constraints.

level. It also includes the operational properties of each module (functional diagram of the module and event responses) such that their interactions provide a description of the global operational properties of the system[50].

The second phase in the development process associated with CODE is the design phase. At this point, the development involves a number of distinct activities, some of which are done in parallel. Also, it is important to note that some of the early design activities are done in parallel with the later architectural phase activities. This way design could provide more feedback and clarification for the architectural phase and, on the other hand, this interaction helps keep the design consistent with the decisions made along with the system architect. We will continue with a qualitative view on the involvement of the architectural step in the development phases that follow

In the beginning, the leader of each design group acts as a module architect, refining the module architecture, detailing the module event responses (via PAD or PAL) and the possible interfaces[51]. The design group leader conducts now a number of review meetings with him acting as a presenter and the group members as reviewers. The outcome of these meetings should be that the module event responses are specified to the required level of granularity. At this time the specific decisions for reuse and/or use of components are introduced; another aspect decided at this time are the various types, levels and parameters of fault tolerance. Once the module design has been agreed upon, an inspection meeting is organized, with the group members as inspectors. To help the inspection one can use again the PAL-based tools mentioned before: editor, verifier and simulator.

As a consequence of the module design being completed, the design group leader assigns the detailed design of the various module event responses to individual group members (again, only the module event responses included in the first product deployment are considered). From this point on, there might be a number of additional review meetings on the way, were individual designers present and explain their design options with the team leader and other members of the group acting as reviewers. The detailed design[52] of the module being completed, there is a final detailed design inspection meeting after which

---

[50] This is possible because each design module (as outlined above) corresponds at a higher level to an execution unit.

[51] At this stage, only the modules required in the first product deployment are considered in any detail.

[52] The detailed module design also includes relevant hardware aspects: specification of the number and types of activity executors (processors, memory, interface, etc.) along with the activity to executor allocation.

the group could basically proceed to implementation. The attainment of the relevant module constraints is checked at this stage as a prototype, using the PAL simulator.

In parallel with and at the end of the detailed design phase, the system architect remains still involved in the process. He calls integration inspection meetings, with the group leaders as inspectors, in order to make sure that the modules still comply with the initial architecture agreed upon and that those modules could be properly integrated. At this level one has to integrate the hardware design of the modules into a single system. This is a good time to make decisions between the system architect with the SW and HW design leaders on what functionality would be implemented in HW and what is taken care of in SW. Once again, the relevant system constrains are checked using the PAL simulator as a prototype. At any of these meetings the various instrumentation and fault tolerance decisions are continuously kept in perspective and reviewed.

An important aspect of the development process so far is that the various documents produced for the architectural phase and the design phase represent almost complete system documentation. When properly annotated, in a well defined, understandable format, with an outline of the major discussions and decisions by the system architect and the design group leaders), they also help with traceabilty of the system implementation to the initial requirements.

With the previous two phases successfully accomplished, let's only take a brief look at the implementation phase. It basically involves two concurrent activities:

- the realization of the required base platform and interfaces (HW and SW included)
- the coding of the various coordinators and activities.

The implementation of the coordinators and their integration with the executors will be the main focus during this phase. The coordinators are basically the specific part of the application; the implementation and coding of coordinators mainly requires entering the relevant data captured during the definition and simulation phases of the event response. Implementation of the executors should not pose significant problems, since some of them are expected to be available off-the-shelf; however, customization and interfacing with the coordinators might require some effort. If not readily available, the coding of the activities corresponds to writing relatively small segments of straight code, activities being inherently more algorithmic in nature. Obviously, a major requirement of the implementation phase is to do proper testing. The test

activity is expected to be a lot more deterministic, given the use of coordinators and the modular aspect of the code.

Case Study: A Hospital Information System

We set out to exemplify the CODE process by following through its steps in a case study that specifies the architecture of a Hospital Information System. Given the fact that in most hospitals computing facilities have been introduced one at the time, many of them remained separated and confined to their initial application area: patient registry, clinics, accounting, and so on. In this particular case study, an integrated hospital information system is needed which is capable to assist with some of the routine actions performed in a hospital such as patient admission, transfer or release, management of medical records, consultations and other day-to-day activities. Once again, we will restrict ourselves to the significant steps of the architectural phase for the following reasons:

- Rather than implementing a Hospital Information System, the goal of the case study is to illustrate CODE concepts and the fact that they could be applied in a systematic manner.
- In preparing the case study we used some work previously done[53] for separate units of the system. Nevertheless, no architectural approach had been taken so far with respect to the overall requirements and how to partition the system. Besides illustrating CODE concepts, this is what we think this case study brings new to the development of the Hospital Information System.
- We have not pursued any contacts with real customers in order develop this system; of contrary, most of the information used here comes from papers in the medical technology field, previous work on this topic or personal experience. Therefore, some of the information required to go into more detail was missing, which impacted the development of certain event responses.
- Because of the size and complexity of the system there is a large amount of effort involved only in the architectural phase, not to consider the entire development. As a consequence, we stopped short from developing PAD/PAL expressions and simulating the event responses by means of the PAL simulator; these were considered out of the scope of the thesis.

[53] P. Cousineau worked on the Registry unit, R. Robichaud gathered requirements on the ICU [52], the cardiac unit being outlined in [1].

It has been mentioned before that the architectural phase in CODE includes two interdependent activities: requirement acquisition/specification and architectural design. The requirement acquisition and requirement specification phases could be folded into one phase called requirement management. At this stage, there are continuous iterations between the two phases and a dialog between the architect and the user, having as result a draft (for now) of the requirement specification. Based on this draft, the first version of the architectural design is done, along with a first model of the anticipated event responses. This model is then advanced to the user for a first estimation, any further changes going back through the requirement acquisition phase in order to be modeled. To a large extent, we will follow a similar path in the following example, highlighting as much as possible the iterative character of this process. The entire case study will be carried at the architectural level of CODE, mostly between requirement management (acquisition, then specification) and architectural design, without going into the modeling phase.

The overall system (first pass)

Based on exploratory discussions with the high level users (senior hospital staff, managers, administrators) the system architect gets a first estimation of the requirements, decides on the critical resources and finally the high level architecture of the system:

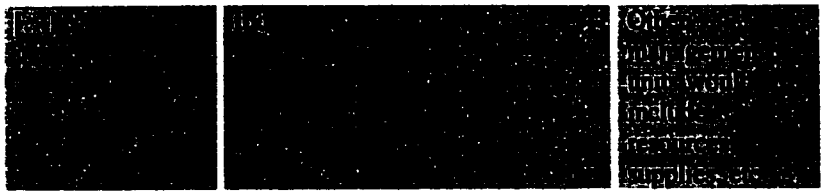
- The critical resource and primary events of the system

Following the initial discussions with high-level users, here is a list of routine activities to be covered. This list is put together by the system architect based on user input; it is grouped by area of activity following the unit organization in a typical hospital (for now, only significant examples in each category have been considered):

Clinic	Cardiac unit	• Retrieve/examine patient record on admission • Patient consultation[54] • Prescribe treatment/medication • Update patient record on release

[54] This step will be expanded once we discuss this unit in more detail

Ward	Cardiac care	• Retrieve/examine patient record on admission • Order medication • Apply medical treatment (daily) • Monitor/record patient evolution (daily) • Update patient record on release	
Services (see note 3.)	Registry	• Admit new/former patients • Book appointments • Handle emergencies • Transfer patients • Archive/de-archive records	
	Inquiry	• Identify inquirer • Retrieve patient record • Display patient record	(allows for inquiries from friends, relatives, etc)
Management (see note 3.)	Patient accounting	• Retrieve patient record on admission • Calculate medical fees • Update patient record on release	



Notes:

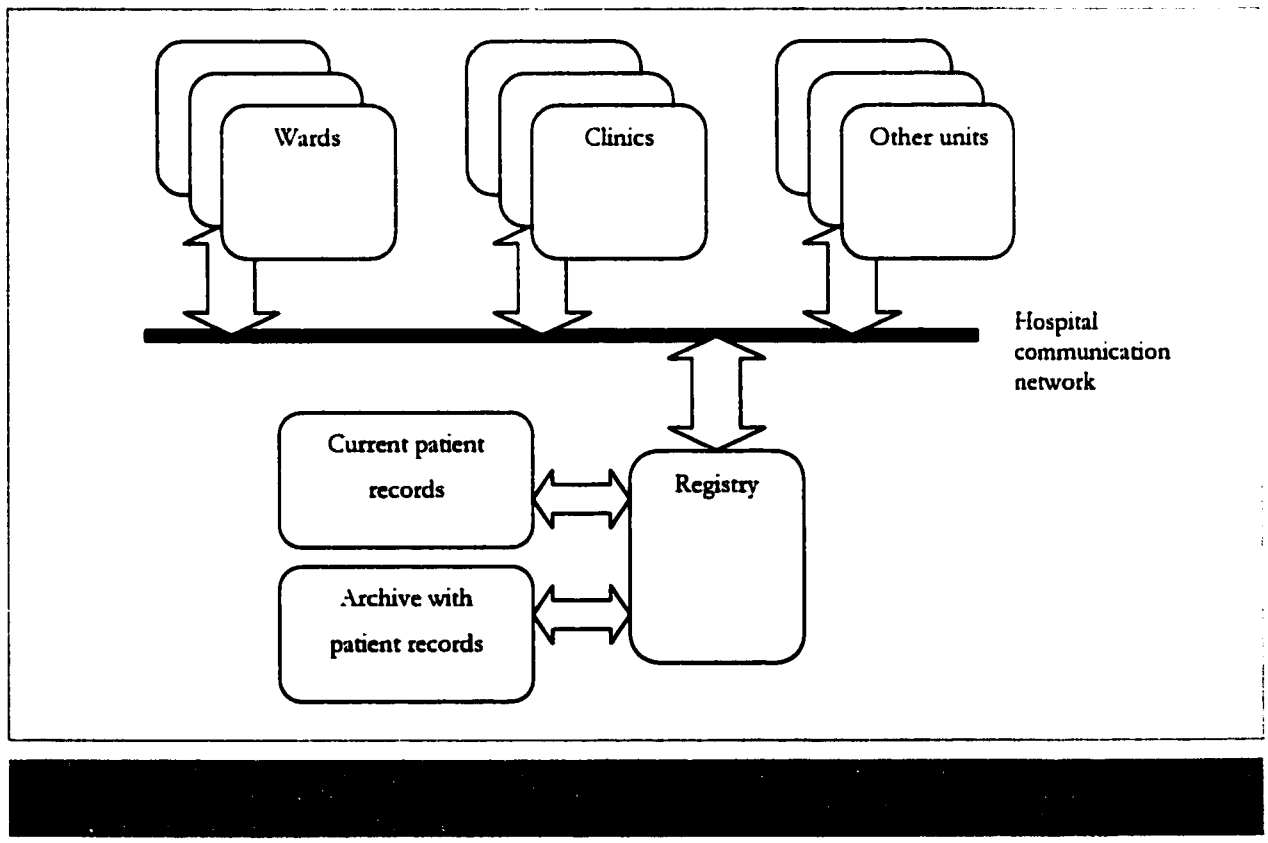
1. The items that have not been detailed yet could be subject to piece-wise delivery (i.e. later additions).
2. Surgery units are organized differently according to the size of the hospital. Small hospitals have a generic surgery unit, handling routine operations. Cases that require special assistance are sent to larger hospitals, which have specialized surgery units attached to each department. For this study we only mentioned the generic case.
3. One could consider at least two facets of management in a hospital:
   - one being concerned with issues directly related to patients (patient billing, booking/releasing resources for an admission/release, etc)
   - another one concerned with the day-to-day issues related to running any institution (maintain inventory, insure supplies for different needs, pay hydro bills, etc).

In this case we only consider the management part that is patient-related. Same holds for services though: there are services directly related to patients care (like the ones mentioned in the table) and services not necessarily related to patients (teleconferencing for example). Again, we only focus on the patient-related ones. A second integrated system could be considered in the cases we just described, separated from the system under discussion and devoted to administrative-type activities, which are not directly related to patients. Being loosely coupled, communication between the two systems could be based on a peer-to-peer relationship. Such an arrangement makes sense in the case of a large hospital; for a small one the two facets could be integrated into the same system since the additional load on the central coordinator would be easier to accommodate.

It appears from the summary above that virtually any unit in a hospital relies on access to patient information; with this type of information missing or being mishandled, patient care could barely take place. Therefore, when it comes to the information flow in a hospital, patient records are we considered as being the critical resources. Consequently the high level architecture would be

designed around coordinating the management of these resources. The fact that we set out to build an information system already suggests it could be organized around a database; this is only confirmed by the particular hospital application we are looking at. Given the fact that in this case the information being handled consists of patient records, the database would be a patient record database. With this specific data storage in mind, the admission of a new patient is considered a primary event. In addition, there are other operations that are necessary in order to maintain the desired information flow, such as record retrievals, updates or transfers. Even by analogy with a paper-based filing system, it is convenient to archive records currently not in use (that is, upon patient release).

- Subsequently, the system architect and the user agree on the primary units of the system.



[55] It makes a lot of sense, since it is always desirable to tailor the system to the application and not the other way around.

A first look at the system requirements (once the above steps completed) suggests a rough organization following the hospital departments[56] and organized around a central repository of patient records. In terms of the repository itself, it will definitely be comprised of an archive containing all patient records not in use, as well as a current database, containing all records of patients currently under observation. As indicated by the list of activities, the Registry department would be the only one to have access to the archive.

- The system architect and the high-level users agree on a rough behavioral specification of the event responses and the relationships between them. At this point more specialized user groups of the event responses are identified, which will be consulted for further refinements.

Once the events identified, we will only focus on the primary events (in italic) and some of the secondary ones, which were considered relevant at this time. As outlined before, any patient admission is considered an external/primary event.

Standard admission of new patient	• Identify patient • Open/fill in new record • Make appointment • Book resources • Admit patient	• Database • Database • Database • Booked resources	Additional DB records: appointments and resources
Standard admission of former patient	• Identify patient • Retrieve/update existing record • Make appointment • Book resources	• Database • Database • Database	Additional DB records: appointments and resources

[56] It makes a lot of sense, since it is always desirable to tailor the system to the application and not the other way around.

	• Admit patient	• Booked resources	
Patient transfer[57]	• Agree on patient transfer • Retrieve all patient data • Copy data to new site • Execute patient transfer • Update record and archive patient data • Release resources	• Database • External connection/database • Database • Database	The patient transfer can be internal or external. Additional DB records: resources, transfers (eventually)

Notes:

1. The few event responses specified in the table confirm once more the patient records as critical resources and therefore the database the main resource to be managed.
2. At this step, the right hand column suggests that the database structure still needs to be refined by addition of new data types as well as connectivity to exterior.
3. Transfers could be external (in which case an external data transfer is involved, along with patient transfer) or internal (in which an internal data transfer would be required, along with the transfer of the patient).
4. There is a lot of similarity between the admission responses, as well as the transfer/release responses. One could think of optimizing these cases by

[57] The layout of this response has later been modified on a more detailed architectural level.

combining them together where appropriate (we will discuss the details later).

- The system architect and the high-level users discuss a listing of various constraints:

1. Quality constraints (at least usability, reliability, maintainability, efficiency)

Usability:
The main feature offered by the system is efficient tracking and management of patient records. A most important aspect in this hospital information system is that data be consistent, secure and error free. The system is expected to provide limited access to the data as well as error detection mechanisms, so as to minimize the risk of malpractice suits; at least some basic error checks should be performed especially when applying medication or treatments.

The system and its components are expected to be user-friendly especially at the man-machine interface; they should be intuitive enough to quickly provide the user with a mental model of the system as a whole. As a consequence, GUI based menus are a requirement. Help and navigation menus are also a requirement and should easily be available at any time (context sensitive help is a bonus).

A minimal patient monitoring system along with means of alarming of medical staff in case of emergency is also a requirement; customization of this monitoring system for particular medical applications and connectivity to expert systems should also be kept in sight. Features like

- Interpretation of test results and diagnostic assistance
- Trend prediction and predictive alarms

are highly desirable.

Reliability:
The system is expected to

1. Be able and handle any combinations of events and system states consistently, as expected, repeatable and non-contradictory at any given time
2. always be predictable (either recover or fail gracefully) faced with adverse or failure conditions

At least partial redundancy is required (that is, at least for the critical parts). The effects of any failures should be localized and not be rippled through the system or outside of it.

Maintainability:
The system should allow for non-disruptive maintenance to existing units in order to either correct errors or introduce new features (SW upgrades); if the case, loss of functionality due to maintenance should be minimal and localized just to the affected area. Same holds for HW upgrades. An open structure that allows for future additions and developments is also required.

Efficiency:
The system is expected to make proper use of the provisioned resources; working at maximum capacity should neither decrease performance under the limits initially prescribed nor necessitate additional resources. At the same time, the system has to integrate with existing resources as much as possible (like computing facilities, medical equipment or communication infrastructure for example).

Performance (timing, number of events to be handled, etc)
The record storage will mainly set the performance of the overall system through a number of parameters as follows:

- Storage capacity (current and archive)
- Average access time to the storage (retrieve, update or transfer) under average load
- Number of simultaneous transactions being handled
- GUI performance
- Availability (or its converse, downtime) measured over five years

Being a system primarily targeted to human interaction, performance constraints tend to be rather soft.

2. Interfacing constraints (with users, with other systems)

It is desirable that the number of terminals correspond to the number of users; in this end, each hospital unit will have at least one terminal access to the system. Today's low cost of computer HW makes it possible to actually install 'intelligent' terminals/PCs with the additional possibility of external data storage/retrieval via floppy units. At least some of the sites would have printing facilities as well.

The system is expected to interconnect to similar systems in other hospitals, therefore standard communication protocols should be used, along with insuring traffic security. Similar requirements are set for the connectivity within the hospital; standard interconnection protocols would insure compatibility with third party equipment. The existing communication infrastructure should be used as much as possible for both internal and external connectivity.

3. Component constraints (HW, SW, ...)

HW/SW components should be as much as possible off-the-shelf, with large availability and within reasonable costs. Future upgrades to more performant HW/SW should be taken into consideration as well.

The overall system (second pass)

With this rough image of the system in mind, it is worth to review and add more details at least to some of the items discussed so far. It is not too early to make forecasts on further system partitioning, possible implementation solutions or set out performance parameters. In order to keep a consistent picture with the items discussed in the first pass, we will review all of those regardless of the amount of detail added.

- The system architect along with the high-level users reviews the choice of critical resources and primary events of the system.

Some further refinements are needed at this stage, only in terms of event responses. It has already been agreed on the patient database as being the critical resource to be managed and on patient admission being a 'primary' event for the system[58]. Nevertheless, it turns out that additional events related to making appointments should be considered. Appointments are primarily made for clinics and involve consultation and diagnosis of the patient along with further recommendations[59] on what actions to take. There is a distinction between external appointments (made for an external patient) and internal appointments

[58] Through 'primary' events we mean external events that 'load' the system, triggering a response; all other events relevant to the response and occurring during response execution are considered 'secondary'.

[59] Appointments could be made for services as well; examples include radiology, laboratory, and physiotherapy. These kind of appointments though are usually the result of an initial consultation, and this is one aspect that could be enforced through the system.

(for patients already under observation), with the former being primary events and the latter considered as secondary events.

The detailed event responses will be discussed further on; the addition of a new data item for appointments has been already made visible on the first pass and will also be discussed along with the second pass over the event responses. On a general note, we wish to point out that all the events mentioned so far also have their 'opposite' events, that is any admission, appointment, transfer, etc. could be cancelled as well. We have chosen not to detail those responses in order to keep this study to a moderate complexity level.

- Subsequently, the system architect and the high-level users agree on the primary units of the system.

While the overall system organization remains the same, further refinements could be made concerning data organization, so as to add more resilience to data corruption or even data loss. In this respect, the system architect suggests that individual units keep their own copies of patient records and other relevant data, as long as they are in use at that site; with this new architecture, we evolve the system from a centralized database system to a replicated/distributed one. The units work on their own copies of data which is then transmitted to the central database of current data, following a specific algorithm (which depends on the implementation). Even though a centralized database system is a lot simpler to implement (especially on general-purpose computers), there are major advantages to the replicated solution, like:

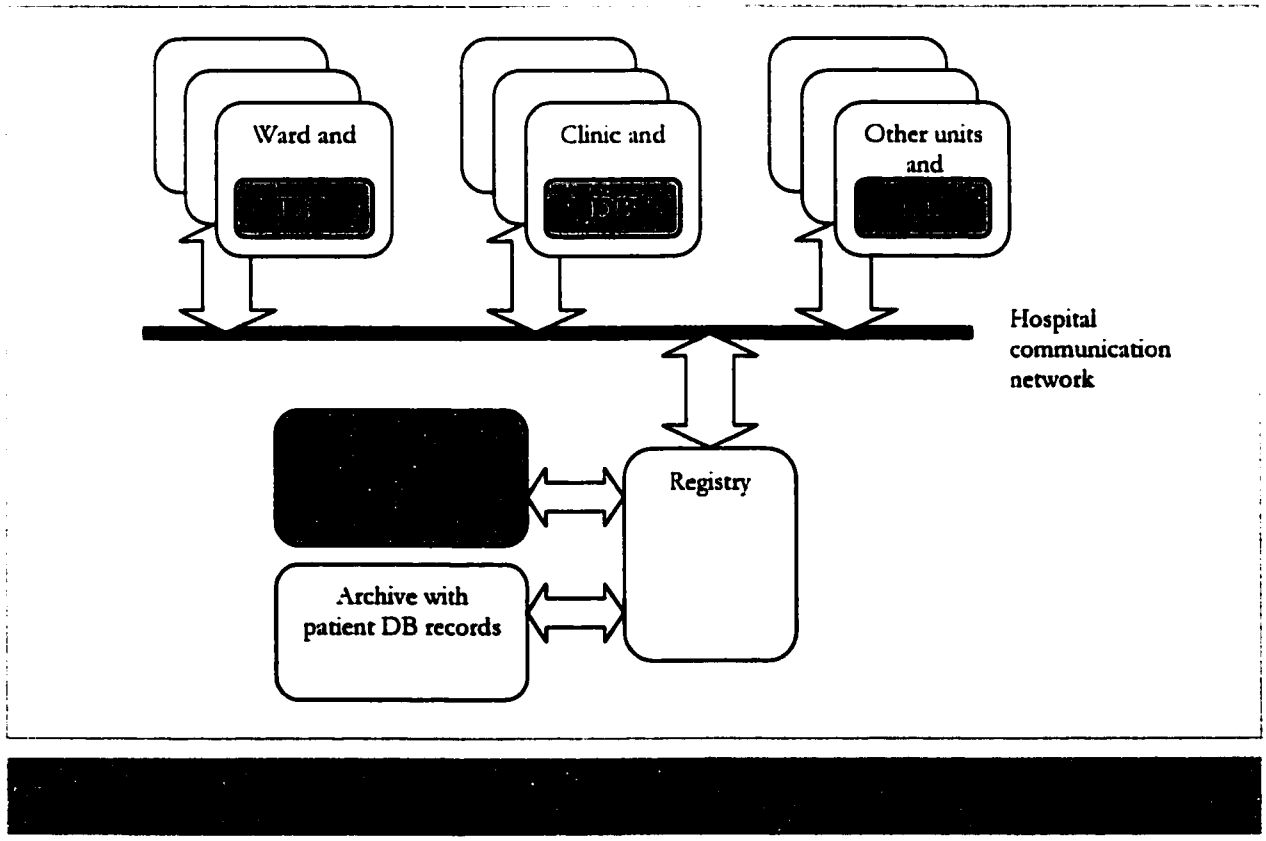
1. Higher *availability* of data: users, from different locations on the network, can access their data locally and in the format they need
2. System *performance* improvements: since data is stored at locations where it is most frequently used, access is faster and at a higher rate, compared to a centralized storage. This solution proves especially useful in the case where the communication network is being already shared with other applications (which is our case, in the context of using the existing communication infrastructure in the hospital).
3. Higher *reliability*[60]: the possibility of complete or partial data loss is significantly reduced a centralized storage is a single point of failure therefore more vulnerable. On the same front, additional security features can be embedded in the transactions between the databases (for instance security, consistency and error checks at unit level as well as central level)

[60] This aspect will be further discussed along with the reliability requirements.

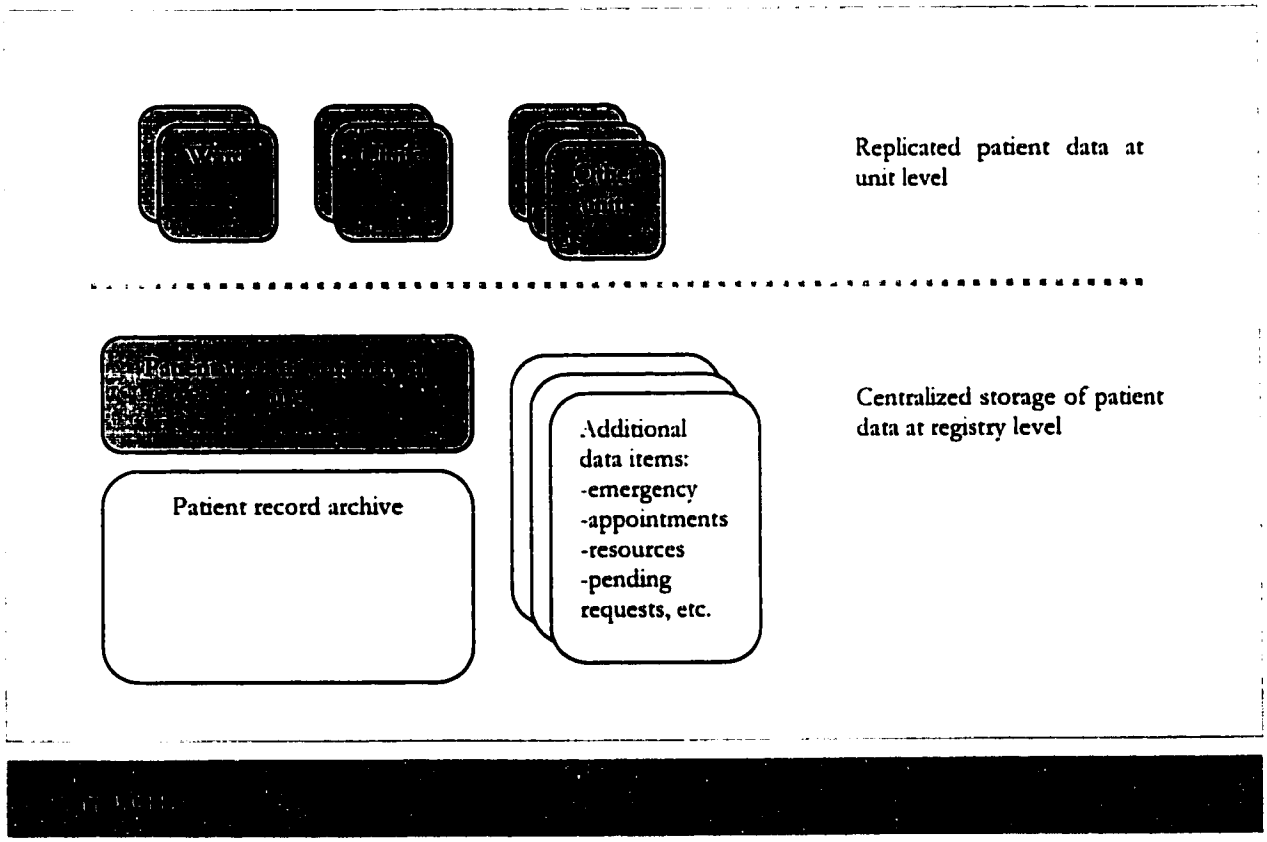
4. Increased *flexibility*: the subsystems corresponding to individual units could be independently developed and customized to specific needs
5. *Maintainability*: the subsystems being loosely coupled this way, it is easier to perform maintenance or upgrade various components of the system

On the other hand, new design problems arise, the most important being

1. data *consistency* between the replicated records. This aspect will be discussed in more detail when we talk about the database manager, but there are various ways not only to overcome this issue, but use it to our advantage by introducing additional error checks.
2. more *storage* is needed; with present day's inexpensive HW, this should not be a real problem.
3. communication *overhead* due to maintaining data consistency may still become a problem; an optimal update schema taking into account update frequency, response delay and other factors has to be worked out. The result should reflect a major improvement from the centralized solution



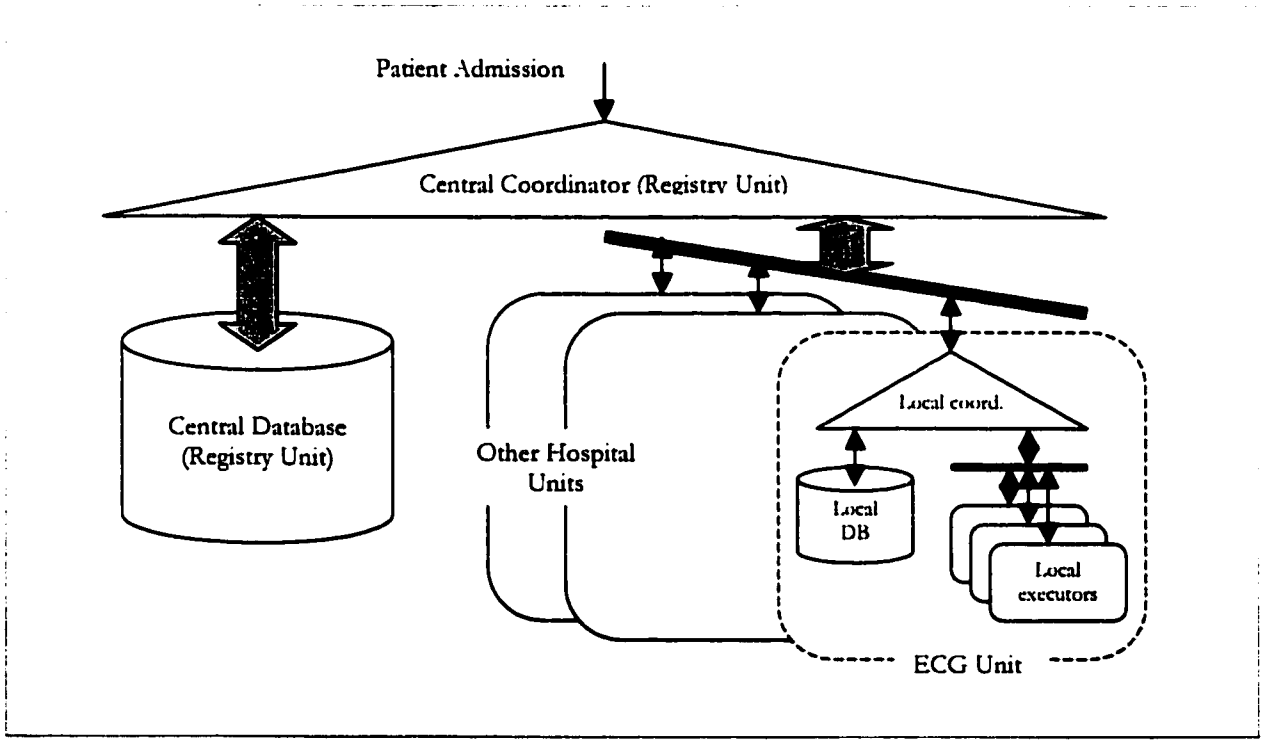
Comparing the benefits to the drawbacks, the replicated database architecture is still considered as a better solution to implement. As a consequence the new system diagram will look as shown in Figure 7 and Figure 8. More aspects related to the decision to replicate the data will be discussed along with the quality parameters.



Based on the overall system architecture suggested by CODE and in support[62] to the data architecture outlined above, two basic system elements could be identified:

[61] It happens in this particular case that the critical resource to be managed is a repository of patient records; one should not understand from this example that the CODE system architecture *has* to be applied only on data.

1. A database manager which controls the access to the current database and the archive as well as interactions between the local databases and the central one. Due to the nature of its tasks, this manager is associated with the registry service.
2. Unit database managers associated to the individual units (wards, clinics, services, etc.) which control their local database and the operations associated to it.



The design of the associated subsystems is based on the CODE general concept of separating coordination from execution, in order to provide a system, which

[62] It happens in this particular case that the critical resource to be managed is a repository of patient records; one should not understand from this example that the CODE system architecture *has* to be applied only on data.

is reliable, efficient, and simple to design and maintain. These subsystems are interconnected through a common bus ('the central bus' for now) and communicate via broadcasting of tagged messages.

- The system architect refines the rough behavioral specification of each event response and the relationships between them. At this stage, the more specialized user groups identified on the first pass start getting involved.

As a follow up to the first pass over the most significant event responses, the system architect along with the registry staff will now take into consideration further improvements. In terms of additions of new data items to the HIS database, the following records can be considered:

1. *Appointments*: which could be simply organized by days (in the form of a regular planner) and in addition could be linked to resource bookings, patient records and staff availability; various query criteria should be available. Further decisions are postponed for a lower design level.
2. *Emergency*: the emergency procedure involves first aid measures and a summary investigation of the patient (along with fast retrieval of important history data in the case of a former patient). Separate records are needed in order to capture information like circumstances of the emergency, patient state, treatments applied, etc; relevant parts of this information will later be folded under the patient's medical record.
3. *Resources*: it is useful to separately keep track of resources as opposed of collapsing them together with appointments (since medical staff can be seen as a resource as well). Nevertheless resource occupancy has to be cross-referenced to patient records.
4. *Transfers*: a transfer record is needed in order to track items that are necessary for patient transfer but don't necessarily need to be part of the patient record (this includes for instance resources allocated at the new site). Such a record is even more useful for external transfers.
5. *Pending requests*: it is useful to keep track of actions to which completion has been deferred, until the user gets closure on them. One such example would be issuing a prescription to the pharmacy; this request is being queued in a pending request queue under it is either acknowledged or rejected (for lack of medication for instance).

On this second pass we have made additions with respect to events and event responses, by considering appointments as a separate event. While these

additional events have been outlined already, their event responses are detailed below (primary events in italic):

Appointment of new patient	• Identify patient • Open/fill in new record • Make appointment • Book resources • Update/save record • Issue appointment slip	 • Database • Database • Database • Database • Printer	Could be folded under external appointments.
[illegible]	[illegible]	[illegible]	[illegible]
Appointment of internal patient	• Validate appointment request • Retrieve existing record • Make appointment • Book resources • Update record • Issue appointment slip	• Database • Database • Database • Database • Database • Printer	

Similarities between the different types of admission suggest some optimization of these cases by combining them together where appropriate. Basically all admissions could be folded into one generic category called 'admission', with an option made by the operator for an emergency procedure or not. Further on, in both cases the system could ask for the patient name and qualify the admission as for a new patient or not, depending on whether the name of the patient is found in the database or not. As a consequence, instead of four primary events, we could only consider one[63]. One can make a similar judgement for the patient transfer/release: transfer can be considered as a case of patient release, from the perspective of the releasing unit (same holds for an external transfer seen from the hospital level). However, it is believed that collapsing these three cases together would actually complicate things, therefore we choose to maintain two separate categories: transfers (external or internal) and releases. Under the same

[63] This requirement was later modified on the lower design levels; after assessing the database manager event responses it was decided that two types (standard and emergency admissions) would better fit the registry needs.

circumstances, combining together the two external patient appointments could reduce the three categories of appointments to only two.

One more mention on the event responses is that we foresee three categories of tasks to be performed, given the structure of the system:

1. Functions that could be performed locally and are initiated by the user, such as retrieval of locally available data or producing and displaying various reports.
2. Functions that require interaction between various subsystems and are initiated by the user, such as downloading a record once a patient has been admitted to or transferred between units.
3. Periodic functions initiated by the system, involving data updates between the central and local databases, error and consistency checks, checking for pending requests or garbage collection.
4. Periodic functions initiated by the system and performed locally, such as local garbage collection.

Functions that are local and do not require interaction with other subsystems are likely to be easier to implement; the ones that require such interactions are inherently more difficult.

- The system architect along with the high level users review the listing of system constraints:

1. Quality constraints (reliability, maintainability, usability, efficiency)

Usability:
As mentioned on the first pass, one of the important features of this information system is to prevent malpractice suits as much as possible. While there is always the option of using an expert system to verify the validity of the treatments issued, there are more simple checks that can also be effective. Let us assume that upon receive of new medication at the pharmacy level, the directions prescribed from the factory are entered in the pharmacy database Here is an example of tracking the application of a prescription once it has been issued:

- the doctor enters the prescription
- the following process can be implemented at pharmacy level:
  - upon receive of a prescription the system can compare the doctor's prescription against the directions prescribed from the factory

(administration, allergies, doses); any mismatches are double checked with the issuer of the prescription then the prescription is validated.
- before delivery the medication delivered is double-checked again against the validated prescription
- at ward level, upon receiving the medication, the nurse enters what she received and the system checks it out again against the validated prescription; same checks can be performed every time she applies the treatment. Items to be checked are name of the medication, quantity, administration doses, appearance, etc.

Again, there is always the option of using an expert system.

No other usability improvements are foreseen at this time; of contrary, we decide to leave open the features related to trend prediction and predictive alarms (eventually would have those processed by a third-party expert system) and make the first delivery with just a basic alarm monitoring system. In this case, medical staff can set monitoring thresholds and then put the patient under observation; once the medical equipment detects any threshold crossing, an alarm will be raised to the staff on duty. Nevertheless, interfacing to third party equipment is an important requirement.

Reliability:
The replication of patient data brings important improvements to reliability:

- the possibility to introduce additional consistency/error checks when performing database transactions. In the case of directly operating on the central database, error checking can only be performed at the interface to the user (upon submission or retrieval). With the individual unit databases, in addition to the error checking just mentioned, similar error detection could be implemented in the transactions between the unit and the central databases.
- redundancy to partial system failure, with the possibility of data rollback.
    - in the case of corruption of the central repository, at least the current data can be retrieved from the individual unit databases. This eliminates the need of having a central database backup at another site, only for the current patient data; the archive has to be backed up[64]. For even more safety, it would be indicated that the current database not be on the same storage as the archive.

[64] One can think of nightly backups of the archive, stored on separate media.

- in the case of a unit database failure, the data can of course be retrieved from the central repository, up to the last transaction transferred there (data rollback).

The mechanisms to update the central repository with the latest transactions from the units could be either journalling (immediate transfer as soon as the update was performed) or updates at specific intervals (regular polling, end of shift or end of the day). We consider the end-of-shift updates more attractive[65] since it spares on the network traffic, given that the requirements of this particular application are not real time. A second machine should be considered, working as a backup to the one where the central database is located; the central database is a single point of failure for the entire system and it is important to insure redundancy at least at this level.

Maintainability:
Maintainability is strongly supported by the clear separation of coordination from execution. Any of the execution units can undergo maintenance or upgrades with minimal impact on the rest of the system. This way, the effect of modifications is strictly kept under control. Another benefit of this type of architecture is that it keeps an open structure allowing for future additions and developments; in this case any changes are mainly restricted to the direct coordinator.

Efficiency:
No further specifications regarding efficiency.

Performance:
As mentioned on the first pass, the record storage will set the performance of the overall system to a large extent. This is a good time to set some basic performance targets to be met (more details following with the design of the individual units):

2. Performance constraints

a) Storage capacity (current records and archive)

A rough estimation of the storage capacity needed could be done based on the expected number of patients and the average size of a record. We will consider for instance

[65] This option was reviewed when discussing the number of simultaneous transactions handled.

- 1000 patients under observation (that is, 1000 records in the current database)
- 50000 archived patient records
- a maximum of 15kB needed to store a current record (considering structured records, with limited text input - the average is ~10kB actually).
- a minimum compression rate of 50% for the archived records (the average here is ~75%)
- a maximum of 50kB to store an archived record, given that the archive contains more than the current information. At an average of 50% compression rate, the uncompressed information would amount to (more than) 100kB[66]

Based on these numbers, the size of the centralized current database is 1000 * 15kB = 15MB (maximum 25MB including the overhead for the database SW). With today's HW, this is easily accommodated on any PC (by today's PC standards, in could easily fit into the RAM...). In the case of the archive, the maximum size is as follows: 50000 * 50kB = 2.5GB again, not a serious challenge for standard PCs.

An additional aspect to be considered is the manipulation and archival of graphics[67] (various graphical records provided by medical equipment, which might be considered worth to archive). Considering there is a way to transfer them into the JPEG format (which is already compressed), such a record would need ~25kB of storage regardless whether it is in the current database or the archive. Considering an average of five such records being stored for each patient, we ask for 5 * 25kB = 125kB/patient. For 51000 patients the storage capacity could go as high as 51000 * 125kB = 6.4GB, still readily available on PCs. As we already suggested, the archive should be maintained on a separate workstation; the same is recommended for the graphic storage, along with backup facilities. The graphic records could be separately stored as attachments to the patient record. However, we do not consider this addition as a requirement, at least for the initial delivery of the system.

The local databases need to be considered as well, each having its specific needs therefore different data requirements for local use. If clinics could basically use the common format and require 10-15kB per record, wards may require as

[66] Keep in mind that only relevant information from a current record gets archived.
[67] One could always think of using workstations dedicated to graphical intensive applications (graphic workstations) which have plenty of computing and storage facilities for this task.

much as 20-25kB per record. With a maximum of 50 patients per unit in both cases, this would amount to 50 * 15kB = 750kB storage for a clinic and 50 * 25kB = 1.25MB for a ward. Service units like laboratories, ECG or radiology are graphic intensive units and may require more storage capacity[68]. Assuming 150kB needed per consultation and about 20 patients a day visiting the unit, a total of 20 * 150kB = 3.0MB needed as workspace. Once again, storage does not prove to be a problem at all, even in worst case scenarios. If standard PCs are used (with an average of ~6 GB disk space), there is plenty of room available to archive the graphic attachments to patient records. It is always an option to replicate the graphics storage of the central archive; for example, all radiology attachments could be archived on the computer in the radiology lab.

b) Number of simultaneous transactions being handled

The number of simultaneous transactions being handled is expected to be extremely low, most likely one at the time. This is a direct consequence of individual units having their own database and therefore localizing transactions at the unit level. Nevertheless, updating the central database with all the transactions performed over an entire shift may prove to be a major bottleneck; therefore a polling mechanism has to be implemented by the database manager in order to pick up local database updates at regular intervals of time. As a consequence, the initial assumption that all new transactions would be transferred at the end of the shift has to be modified in favor of regular polling, which would spread out over the entire day the traffic load on the network.

c) Average access time to the storage (retrieve, update or transfer) under average load

User interaction considerations set a maximum limit of seven seconds, under maximum load, for any given transaction (in a case where the transaction cannot be performed within this interval, the user should be notified of the transaction in progress and get the final result later). The fact that we implement local databases helps once again to cope with this performance constraint, since no major problems are expected in dealing with the local database; nevertheless a transaction-buffering scheme should be taken into consideration.

[68] In this case for instance, a study on the processing performance would also be useful. This is considered out of the scope of this case study, since we set out to illustrate the CODE methodology rather than implement the system.

d) GUI performance

GUI performance is set again by user interaction considerations; it should not take more than two seconds to open any form in a case where there is no data to be retrieved and displayed, or data is not readily available. This leaves a maximum of five seconds to retrieve data if the case. Let us mention again that the system is expected to perform a lot better than this due to local data storage.

e) Availability (or its converse, downtime) measured over five years

Expected availability is above 95% over five years.

3. Interfacing constraints (with users, with other systems)

In addition to the items discussed on the first pass, we will now refer in more detail to the interconnection mechanisms. As mentioned before, we intend to use the existing communication infrastructure; assuming the hospital has an Internet network already installed, the communication protocol of choice would be TCP/IP[69]. A direct consequence of having access to an Internet connection is the possibility of using the standard Web browsers for interfacing to the user (along with HTML/Java user interfaces). The use of the Internet connection provides an integrated communication media for use inside and outside the hospital. However, Internet communication outside the hospital campus raises security problems, therefore all external access has to be passed through firewall machines; this level of security is considered outside the scope of this system. Specific standards would be used for interfacing to third party medical equipment (therefore some drivers may have to be written). One example of a standard protocol for communication with medical equipment is the Medical Interface Bus (MIB), based on the seven layers OSI reference model.

4. Component constraints (HW, SW, ...)

As already hinted, PC platforms should easily accommodate the entire application. Separate storage machines are recommended for the archive and eventually storage of graphics. In terms of using off-the-shelf SW components, there are quite a number of databases readily available, which can be customized to accommodate this specific application. Consequently, the focus of the

[69] A lot of hospitals today have access to ATM (Asynchronous Transfer Mode) routing, especially for remote medical assistance or videoconferencing; this would require a more performant network (coaxial or optic fiber), different from a regular infrastructure, since twisted pair is not good enough for ATM connectivity. For this case study though we only consider the most popular communication protocol.

development would be on the central coordinator and response managers as opposed to the databases[70].

This very much concludes the first two iterations through the requirement management phase; further refinements will probably occur after the system has been modeled and passed to the user for a first evaluation. For now though the system architect proceeds to a second, more detailed requirement specification phase[71].

The central database and the database manager

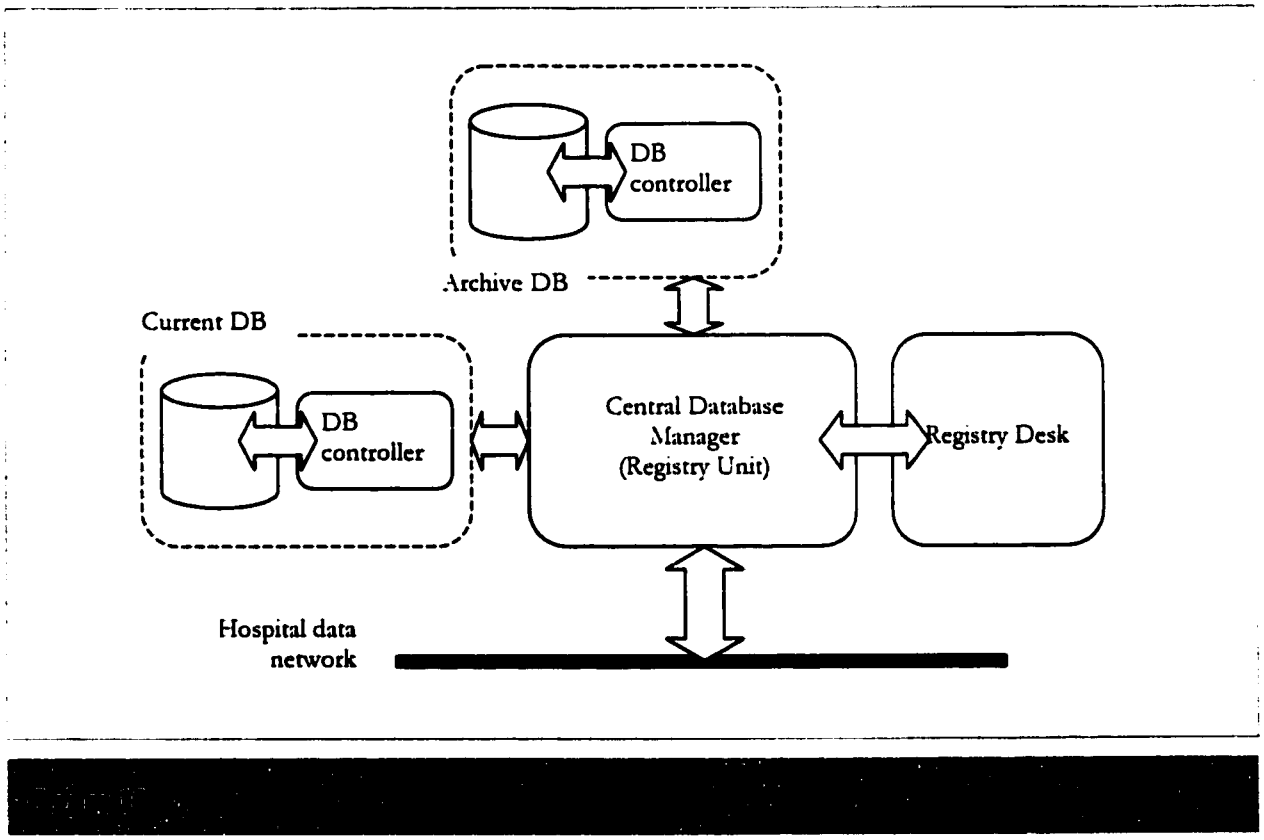
The central database subsystem has been roughly described before, along with the overall system description. It mainly consists of the two databases (current and archive) and the database manager. Nevertheless, we will now detail this picture a little bit more.

In the case where off-the-shelf components are used, it is very likely that the two databases would have their own database managers; however, an additional management level needs be in place so as to control the two managers as part of the central database subsystem. The additional manager will be the overall database manager and is part of the SW that has to be written, following the structure recommended by CODE. Other than these entities, one would still want to mention an admission interface to the user at the registry desk and a bus controller as an interface to the system bus (and hospital network at the same time).

As previously decided, what will be called 'the database manager' is actually the control element of the centralized database as well as the main system coordinator (in charge of coordinating the various elements of this subsystem). The system architect along with a representative of the registry staff now proceed and review the responses assigned to the registry/central database subsystem, in light of the decisions already made when discussing the overall system. It was mentioned at that time that the four types of patient admission

[70] It is interesting to note that we started developing this application similarly with developing applications from code libraries. Once the coordinators are in place, one could think of reusing the same coordination structure for other applications, in which case the databases would probably have to be replaced; our application has become now a framework.
[71] We will only look at a few significant examples in this case study, those being the database manager, the ECG unit and the ICU unit.

could be summed up in a single one with subsequent checks for whether it is a standard admission or an emergency one and whether it is for a former or a new patient. However, it is now believed that keeping the standard procedures completely separated from the emergency ones would better serve the user needs (emergencies having a much lower incidence rate than regular admissions). Consequently, we will further consider two separate responses: standard and emergency admissions.



Standard patient admission:

This process is initiated by the registration unit (admission desk) and involves the following actions:

1. Once the patient identification entered, the database manager looks up the archive to retrieve the appropriate record.

- If a match is found, the record is uncompressed and the patient profile along with relevant medical history is retrieved, following the retrieval strategy[72] specified by the admission desk. The retrieved data is combined with other admission information to form the current patient record.
- If no match is found, the system assumes this is a new patient and a new current medical record is generated by means of a special questionnaire to determine the profile of the patient and its medical history.

2. According to the specifics of the admission request, the registry staff makes the appropriate appointments and resource bookings.
3. The current medical record is updated, transferred to the current database and at the same time is broadcast over to all the units so that they can extract and store relevant information in their local databases.

It appears, based on our initial assessment of the event responses as well as this detailed description, that appointments and resource booking will have their own separate event responses, associated to the database manager.

Emergency patient admission

This process is initiated by the registration unit (admission desk) and involves the following actions:

1. The system generates a new emergency record where patient identification is entered.
2. According to the specifics of the emergency, appropriate emergency resources are allocated and patient sent in.
3. The database manager looks up the archive to retrieve the appropriate record.
   - If a match is found, the record is uncompressed and the patient profile along with relevant medical history is retrieved, following the retrieval strategy specified by the admission desk. The retrieved data is combined with the (emergency) admission information to form the current patient record.
   - If no match is found, the system assumes this is a new patient and a new current medical record is generated by means of a special questionnaire

[72] The retrieval strategies make a separate topic.

to determine the profile of the patient and its medical history. Due to the emergency situation this could be collected later[73].

4. The current medical record is transferred to the current database and at the same time the record is sent to the emergency unit.

One could think of using the booking event response in order to allocate emergency resources. There is also potential to exploit concurrency in order to make the former patient record (in case there is one) readily available: in the above response, step 3 (searching the database) could be done at the same time with step 2 (resource allocation), as soon as patient identification is entered.

Patient transfer

The initial assessment of the event responses was used as a base for detailing this particular response; however, some changes occurred. Prior to detailing the response we noticed that some of the responsibilities could be shared with the units. As outlined below, the initiation and termination of the transfer could be deferred to the initiating and receiving units respectively, at times the central database manager acting as a mediator between the two. Sharing these actions between the units slightly modified the initial layout of the response[74].

A ward or emergency unit initiates the process by sending a transfer request to the registry (that is the central database manager). Consequently, the following actions take place:

1. The database manager checks the validity of the request; if not valid, the request is returned to the originator with an error message.
2. Once the transfer accepted the database manager signals the originating unit to start the transfer procedure and the target unit to prepare for acceptance (preliminary patient information is sent to the receiving unit). The originating unit is expected to have updated the current patient record and now ask for archival of irrelevant information if necessary. The receiving unit is expected to ask for resources to be booked and retrieval of additional information, if necessary; if this unit needs additional information, it is extracted from the archive.
3. The database manager receives the current record, archives the information from the origination unit and updates the current record with information asked for by the destination unit.

[73] Eventual checks to ensure the information entered is complete before submission should be disabled.

[74] In this case, one has to go back and modify the initial event response.

4. The record is sent to the targeted unit as an acknowledgement of the transfer. The originating unit is asked to release associated resources.

Let us note that the patient record along with the archival request can be transmitted from the first time as a transfer request (as opposed to step 3). It is worth to mention that the patient record is not deleted from the database of the originating unit[75].

Patient release

The releasing unit initiates this procedure, which involves the following steps:

1. The database manager checks the validity of the request; if not valid, the request is returned to the originator with an error message.
2. Once the request accepted, the database manager verifies that all outstanding service requests are completed; if not valid, the request is returned to the originator with an error message.
3. The database manager asks for the patient record in order to archive it; once the record received it signals all other units to delete this patient from their databases and the originating unit release associated resources.
4. Once the archival completed, the current medical record of the patient is deleted from the current database.

Just like in the case of a patient transfer, the patient record along with the archival request could be transmitted from the first time as a transfer request. Before concluding with the central database manager and move on to some of the individual units, let us mention that there are more event responses to be analyzed, such as service or update requests (initiated by the units and serviced by the database manager). We have only expanded on a limited number of responses, in an attempt to illustrate the design process rather than implement the system.

From here on, the detailed responses can be implemented in PAD/PAL and fed into the PAL simulator for a first running model of the system. The users are then consulted on this model and the requirements could be revisited if necessary based on the user input. The event responses are modified accordingly.

---

[75] The record is only deleted upon patient release; this way, the patient has a well-defined record track while still under observation within the hospital.

The ECG unit

The basic function of the ECG unit within the hospital is to record, analyze and store ECG tracings. Additional functionality includes on-spot identification of faulty tracings followed by alarming the medical staff for an immediate retake, digital storage and analysis of ECG records as well as handling of external ECG requests. All of the above is based on the steps involved in a standard (paper-based) ECG unit. These steps have been briefly outlined at the system level, when the system architect and the senior medical staff discussed the specific activities in each unit. Following more detailed discussions with an ECG unit representative (i.e. the cardiologist), a recursive step has been added for case where the ECG recording is faulty. The revised and more detailed activity flow is listed below:

Unit type	Unit	Activity	Notes
Checkup clinic	Cardiac unit	• Patient record arrives ahead of time • Patient checks in and an ECG request is filled • An ECG is taken and given to the patient • Cardiologist sees the patient and examines the ECG tracing; if the ECG is faulty, a new ECG is taken • Patient record is updated (diagnosis, medication, ECG trace)	In a regular system this would require another appointment. At this point a prescription could be issued as well.

If the ECG-related functionality could only be performed by specialized medical equipment, while the manipulation, digital storage and analysis of ECG records could be assisted by the hospital information system. As background information, we assume ten cardiologists associated with the clinic, working in shifts of five. The clinic operates for two hours in the morning and another two hours in the afternoon; a consultation takes ~20 min., out of which five are spent on examining/updating the records. The ECG, which is taken in advance,

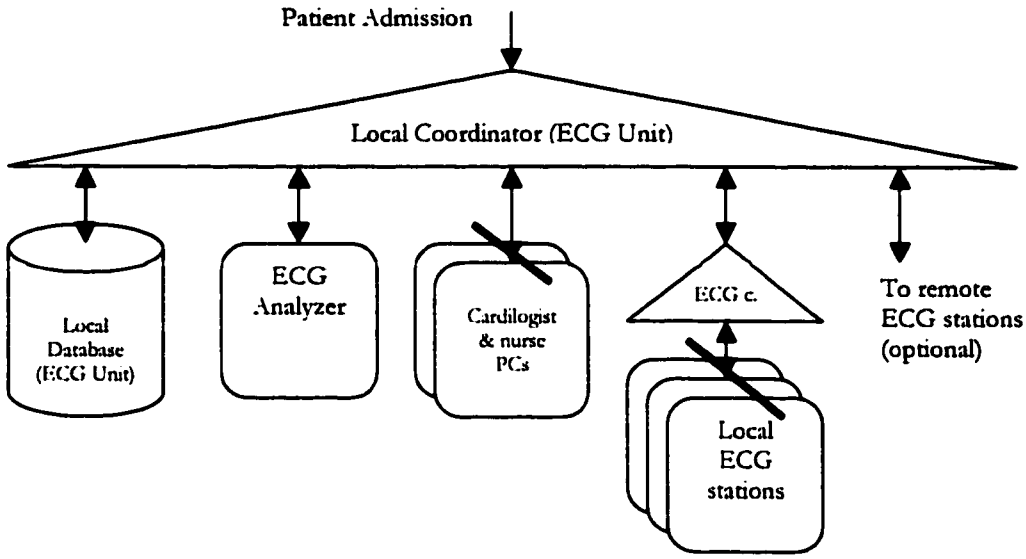
takes ~15 min. One can assume four ECG stations working in parallel. This data will be useful to further detail the design of the unit.

The architecture of the unit is also based on separating coordination from execution and consists of a coordinator with the actual ECG stations as working units, tools to analyze the tracings, the storage facilities, as well as the user interfaces (PCs and printing devices). The ECG stations themselves would have to be under the supervision of a separate workstation, which has the task of centralizing and presenting the tracings to the ECG coordinator, in an orderly fashion. Responses for performing this task will not be detailed here.

The possibility to service remote ECG stations has already been mentioned; this case is somehow analogous to the emergency admission at the central coordinator level and is required in a case where the requesting unit has no facilities for ECG analysis. A special procedure is needed in order to avoid any delays in servicing this type of request. The cardiologist has access to PC-based tools to magnify, compare and analyze the tracings, as well as access to the hospital network via the coordinator. Nurses have access to both the local and the central database also through the local coordinator. The coordinator can also communicate to the ECG stations and perform basic checks on the data that is being exchanged.

A brief look at the activity flow in the unit reveals the time spent by the patients in the ECG unit as being the critical resource. This consideration is implied by the fact that all processes relate to minimizing this time; as a consequence, the entire information flow is being synchronized with the patients[76]. The SW for this unit should be built around optimizing the number of patients that could be diagnosed within a given time interval. Most of the time in the unit is being spent analyzing the ECG tracings as well as accessing patient information. From a system architecture perspective, this translates into the patient database, ECG analyzer and associated computing equipment being controlled directly by the unit coordinator.

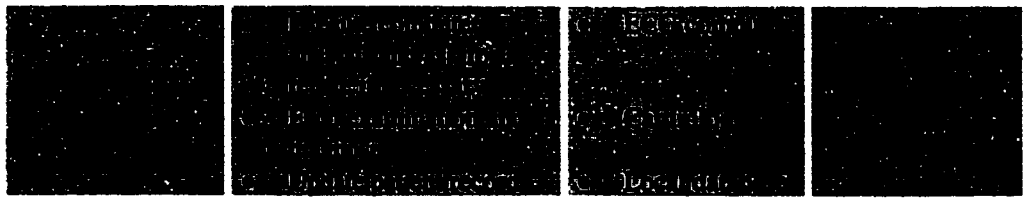
[76] One could argue in favor of the ECG tracings as a critical resource, but these tracings in their turn are tightly related to the patients they belong to (they are the patient's representation within the ECG unit). Another argument that could be carried on the critical resource topic is to look at the patient record as a critical resource.



Based on the critical resource, the primary event is considered to be the patient check in[77]; this event triggers the main event response in the ECG unit.

Receive patient record	• Receive appointment request and patient record • Store patient record • Update appointment list	• Receive buffer • Local database • Local database	Error checks are done

[77] More room for debate: considering the patient record as a critical resource, the primary event would be the arrival of a new record in the database. However, we consider this as being part of the patient admission event response, and not specific to the ECG unit.



[...] [...] **Other event responses**

There are more responses to be mentioned in the above table; one particular example is the case where ECG requests need to be accepted directly from external hospital units without going through the entire registry process (analog to the emergency registry procedure). In this case, three concurrent sets of activities could be instantiated, corresponding to:

- Straight ECG analysis, with no comparison against older records
- Query for available cardiologists
- Check for patient history

Although this is a more complex response (most likely combining some of the smaller ones, already defined), its analysis would be analogous to the one we discuss here, therefore will not be detailed.

We will instead focus on the actual patient consultation, which is a good example of a case where third-party existing equipment is involved. The ECG station has to be connected to the local coordinator in order to allow for manipulation of ECG records. On the other hand, for the computer analysis, one could either use an expert system or provide some basic tools through the information system to assist with the analysis[80]. The patients are identified by a patient key, containing not only the name of the patient, but also the name of the cardiologist and the date of the appointment, as well as the patient status (former or new). This key identifies all information transfers. A current visit status can be stored on the local coordinator (similar to the pending request storage on the central coordinator). This status can be verified throughout the day to ensure that all requests and received data are consistent and patients finally get their treatment completed.

[80] Simple tools would include basic storage and retrieval facilities for ECG records, as well as diagram magnifiers.

From now on, the system architect and the ECG representative proceed by detailing the event responses above, step by step. For this case study, we will only consider the primary event.

Patient check-in

This process is initiated by a call from the ECG station, along with the patient key as the input data. It is important to note that the patient record arrives ahead of time, along with the appointment date, and it is stored away in the local database until the day of the appointment. This action is being performed as part of the regular data exchange between the central and local manager, by a separate response (which will not be detailed here) in the local database manager.

1. The patient shows up; the coordinator retrieves the record and checks the appointment validity/status. If the check fails, an error message is sent to the ECG station and the record is purged from that machine; the patient is declined the visit.
2. Once the validity check passed, the patient is sent to the ECG station and takes an ECG examination; basic checks are performed here to see if the trace was correctly recorded.
3. The trace is distributed to the cardiologist PC as well as to the ECG computer for analysis[81].
4. The ECG computer executes an ECG analysis; if it turns out inconclusive, a retake is ordered and the data is invalidated. In a case where the trace turns out satisfactory, the ECG station is signaled to release the patient.
5. The cardiologist sees the patient for additional examination and final diagnosis. The patient file is updated and stored accordingly; a decision is made here on whether the ECG record would be stored along with the patient record.

One would now detail only the processes associated to the local coordinator through PAD/PAL and simulate them. The processing requirements of the specialized systems involved are not detailed since they are considered 'external'

[81] There are actually two options here:

- first is to transmit the trace right away for analysis by both the cardiologist and the ECG computer, then acknowledge it. This ties up the system for a long time.
- the second option is to store the data locally, acknowledge it then analyze and review the acknowledgement if necessary.

We will choose the second option, since it is considered more effective.

activities; once started, they are delegated to third-party medical systems which do not need any intervention from the coordinator.

The ICU unit

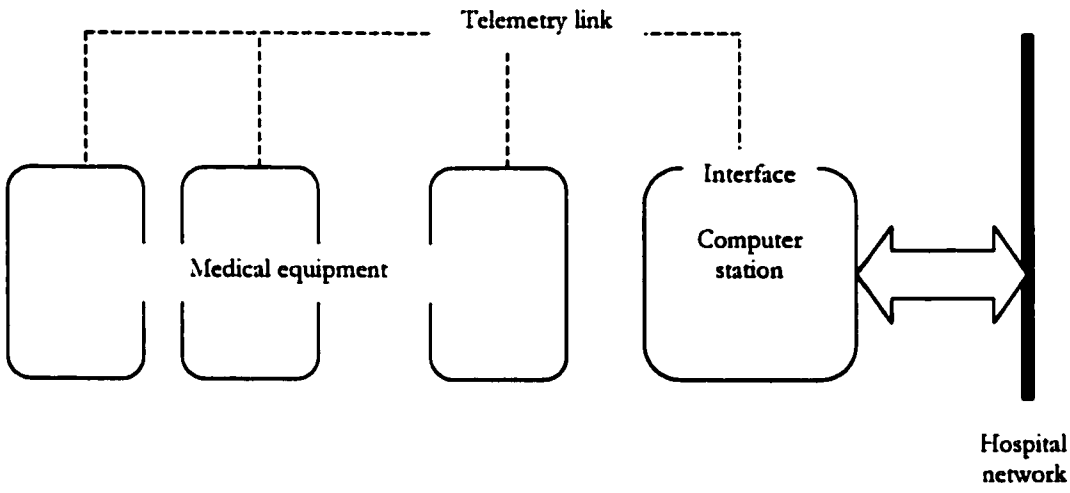
The function of the Intensive Care Unit (ICU) is to provide patients with assistance for a fast recovery, usually after operation. From the HIS perspective, it needs to host a patient monitoring and advisory system, which integrates data from the patients in the unit and displays the information needed to the attending staff, on a single screen. This monitoring system is considered a valuable assistant in identifying immediate and preventing potentially critical situations. Once again, existing medical equipment has to be interfaced and integrated within the information system.

Just like before, the system architect starts by reviewing with an ICU representative the main activities taking place in a typical Intensive Care Unit (these activities had been previously listed when discussing the overall system).

Unit type	Unit	Activity	Notes
Ward	Intensive Care Unit	• Patient checks in • Patient record is retrieved and updated accordingly	If the case, ICU staff orders specific medication
		[redacted]	[redacted]
		• Patient evolution is monitored/recorded daily • The prescribed treatment is applied	This activity could be divided into regular monitoring/treatment and emergency response.
		[redacted]	[redacted]

One could already identify a lot of commonality between the overall process in this ward and the process in units we previously looked at. Activities like patient check-in, patient release or retrieval of patient record are essentially the same; as expected, differences come into play when it comes to activities related to the specific of each unit.

Given the fact that we basically want to centralize data collected from existing medical equipment, the HIS for the ICU unit suggests a data acquisition system organization; this system would monitor signals and data from medical instruments and display them on the screen, alarming the medical staff if necessary. To some extent, the design of the ICU unit shares the same characteristics with the one of the ECG unit. Patient monitoring is actually performed by (existing) third party equipment, while the task of the HIS is to centralize and display information and, if the case, alarm medical staff. Consequently, at this stage the basic organization of the ICU information system looks as depicted in Figure 12.



Specific system features would include:

- Pre-programmed configuration (therefore no need for special setup); at the same time the possibility to be customized for different types of hospital wards.
- Permanent monitoring of medical equipment combined with a comprehensive alarm system to alert medical staff if the prescribed thresholds are reached.
- User interface which is easy to use and understand, even by lower trained medical personnel.
- Automatic instrument verification and activation of appropriate alarms upon fault detection.
- Trend prediction capabilities are a desirable feature.

Being integrated with the rest of the hospital information system, the ICU unit should also have local capabilities to store patient records and other specific data, providing as much as possible for a paperless environment. Connectivity to other hospital services (like pharmacy, laboratory or kitchen) is also a need.

Since the ICU unit is primarily a monitoring unit, a lot of design emphasis will be put on interfacing the system to the user that is, on the GUI and its functionality. Without going into a lot of detail at this level, it is intended that information for each patient be displayed by request, on a single screen/window, whereas another screen/window would carry overall information about the patients' condition for the entire unit.

No storage or dependencies are forecasted at this time; given the fact that a PC could easily accommodate the database for the entire hospital, so much the better for a single unit. In terms of computing, the system should be able to cope with the extreme situation where there each patient in the ward experiences an emergency; based on this fact, a true multitask/multiprocessing operating system is indicated for the computer in this unit. Nevertheless, rather than storage or computing facilities, the platform constraints are mostly related to interfacing to the medical equipment; just like in the case of the ECG unit, standard medical bus interfaces and drivers would be used. Implementation of the data acquisition part is considered to be straightforward, given the fact that there are a lot of off-the-shelf products available, capable to easily perform this task. Basically all this functionality could be easily accommodated on a single PC, connected to the medical equipment, as well as the hospital network. It is true this PC could become a single point of failure in the unit, however, in

absence of a spare station, the essential task of monitoring each patient would continue to be carried individually by medical equipment.

To summarize the overall system organization suggested above, the role of the ICU computer is to

- Monitor the medical instruments in a prescribed way; also implement various levels of alarms based on crossing specific thresholds that are set
- Keep track of all the real-time and mean values of recorded data; during an emergency situation the system should be capable to automatically switch to instantaneous values.
- Communicate with other hospital units and services
- Locally store patient records and other relevant patient data

Before detailing the system implementation in terms of event responses, the system architect highlights a few improvements in the process, made possible by the use of the HIS system:

- the patient record arrives ahead of time, forwarded to the ICU unit by the unit/doctor last visited by the patient. This would allow for early ordering and shipment of any necessary medication from the pharmacy
- the system could provide for multiple error checks with respect to
  - the medication that was shipped; the invoice could be independently verified against the original prescription
  - the treatment being applied; each time before application, the tratment could be double-checked against the original recommendations

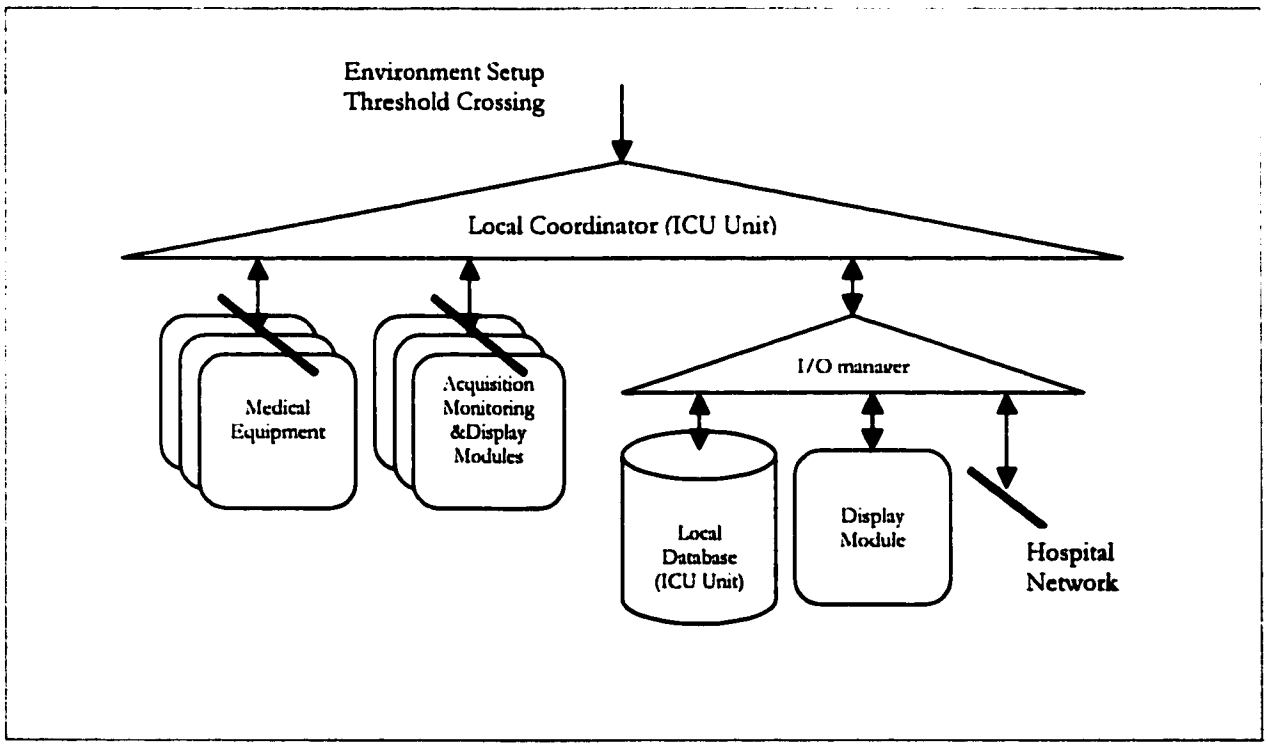
When it comes to this type of hospital unit, one could be see that the patient state has to be dynamically managed especially in a case where a serious condition occurs. The data on the patient state is collected by the acquisition system and presented to the medical staff through the monitoring stations. According to doctor prescription and patient evolution, the appropriate medical treatment is applied. In a worst case scenario, one should provide for handling (eventually multiple) emergency situations in the unit. All these considerations are pointing towards the patient state as being the critical resource to be managed in the ICU unit. Based on the choice made for the critical resource, the threshold set up as well as the crossing of a previously set threshold are considered primary events.

According to the list of activities outlined before, it is possible now to sketch out the ICU unit events (highlighting the primary ones) and associated event responses:

Receive patient record	• **Receive appointment request and patient record** • **Update appointment list and book resources** • **Order medication if required** • **Store patient record**	• **Receive buffer** • **Local database** • **Pharmacy** • **Local database**	**Patient record is received ahead of time.** **Error checks could be in place here.**
[illegible]	[illegible]	[illegible]	[illegible]
Environment set up	• **Validate setup access** • **Set up monitoring environment** • **Validate and apply settings**	• **Monitoring station; local database** • **Medical equipment** • **Local database; monitoring equipment**	**Security checks performed.**
[illegible]	[illegible]	[illegible]	[illegible]

Threshold crossing	• Alarm medical staff according to the preset severity levels • Continue monitoring according to the preset severity levels • Update patient record	• Monitoring station • Medical equipment • Local database	
[...]	[...]		Other event responses

The SWA described below emphasizes patient monitoring and data display as priorities compared to the administrative work related to patient check-in, consultation or release.



For practical reasons, it is recommended that the display units (monitors included) assigned to these two types of activities be also separated. This way, any interference between an emergency situation and the other activities in the ICU is being reduced to a minimum.

The system architect and the ICU representative would now proceed by going into more detail with the event responses mentioned above. We will only consider two of them: the 'Environment set up' and the 'Threshold crossing'.

Environment set up

This response is initiated upon completion of the patient check-in, as soon as resources booked for a particular patient have been allocated. A proper setup of the monitoring and alarm environment is instrumental for the recovery process of the patients. Because this step is so important in the overall ICU process, one could think of various security and error checks to be provided for at this stage. Security checks insure that only authorized personnel (i.e. the ward doctor) has access to the setups, while error checks verify (to various extents, depending on the implementation) the setup correctness.

1. Once the patient admission completed, the medical staff is asked to set up the monitoring environment. The authorized personnel enter the patient key as well as their own security ID; upon validation of both, the system is granting access to the environment setup.
2. The setup of the monitoring and alarm equipment is performed in the following way:
   - the monitoring equipment is set according to the results of the patient consultation performed on admission.
   - the alarm thresholds are set according to the initial recommendations as well, on priority levels function of the severity of the situation.
   - emergency handling scenarios are selected or customized from a set of pre-defined ones. At this stage, it is also decided the personnel to be alerted in case of particularly serious conditions.
3. The entire setup is double-checked for errors, then validated and applied[82].

[82] There could be two levels of error checking here:
- a coarse grained one (which could already be covered by the medical equipment itself) ensuring the setups would not exceed the medical standards
- a fine grained one double-checking the settings against the ones prescribed at the initial consultation, according to patient condition

It is important to that the security and error checking features of the HIS provide for the lower trained medical staff to have access to the environment setups. This way, once the doctor recommends the settings, he could authorize specific personnel to enter them and once this done, the settings are again double-checked against the initial recommendations.

Threshold crossing

The monitoring equipment, upon crossing of any of the preset measurement limits, triggers the threshold crossing event response. Regardless of any particulars, the monitoring station raises an alarm; depending on the severity, the alarm could be escalated. Also depending of the severity, the acquisition rate of the monitoring equipment, as well as other critical parameters are being modified to handle the emergency. All these actions are part of preset emergency handling scenarios, which could also be customized at setup time.

1. As soon as a threshold crossing is detected, the monitoring station raises an alarm, along with a display of relevant data on the patient condition. The medical staff, upon providing proper authorization, could eventually acknowledge the alarm.
2. A continuous severity evaluation starts being performed. The evaluation has two dimensions
   - a qualitative dimension, in terms of the nature of the condition
   - a quantitative dimension, in terms of how much and for how long the thresholds have been exceeded
3. According to the severity evaluation, the preset emergency scenarios are being activated. These could include escalating the alarms, modification of the acquisition parameters or interactive capabilities of the system.
4. The response terminates either after the patient returns to a stable condition or through manual override by authorized personnel.
5. All the events, along with relevant data and the actions taken are being automatically logged to file.

In order to improve system response, there is a lot of room for concurrency to be exploited. For instance the qualitative and quantitative aspects of the condition could be computed concurrently, while the severity evaluation itself is being performed concurrently with all the other activities.

We would like to stop here with the analysis of the Hospital Information System, as an illustration of the architectural phase in the CODE design process.

From this stage on, further refinements are possible, one is getting closer to the design phase. As we have already pointed out, there is an iterative process where the event responses are simulated through the PAL simulator, the user having now a first exposure to the running model of the product. Eventual modifications are agreed upon between the user and the architect and as a consequence, the requirements are modified accordingly. The event responses are reviewed and a new model is simulated. Event responses are then refined until the smallest work units of interest to the user (activities) are reached; at this level, event responses end being made up of a number of activities. Depending on the size of the system, two layers of responses could be considered: a first level of elementary responses (made up of activities) and a second level of combined responses, made up of a number of elementary responses. It makes sense to keep a hierarchical view of the response structure, in line with the general principles of CODE-based design.

Concluding remarks

This chapter introduced CODE, a coordination oriented design environment primarily based on a behavioral[83] specification of the system to be built. The important benefit of specifying the system in terms of responses is a clear separation of coordination from execution; this separation comes as natural consequence of the response oriented approach, rather than another design constraint imposed on the system. Strictly from a coordination perspective, the CODE architecture comes close to the Task Assignment coordination pattern presented in Chapter 3. In line with the concepts there, CODE achieves a hierarchical structure through a top-down goal decomposition in which work assignment is done in terms of event responses. From a broader perspective, although structurally the same (involving coordination centered on critical resources and execution), the CODE architecture offers the benefit of always suggesting how a particular implementation has been achieved. This way, unlike other architectural models, it comes closer to what SWA and architectural styles really mean.

One cannot consider CODE just a methodology or an architectural style. It is an integrated development environment that includes:

- a hierarchical coordination-oriented architecture

[83] The motivation for such an approach has been already discussed when we emphasized the rather interactive character of today's SW systems.

- a matching development process spanning the entire life cycle of the product (requirement specification, architecture, design, implementation, delivery, maintenance and retirement)
- a set of conceptual models and tools including the architecture model as well as the PAD/PAL representations.

With CODE, the system to be built can be closely wrapped around the requirements of the application. Many refer to the 'school mentality'[84] to define something that attempts to fit all. In reality, one cannot expect that the same set of tools be needed for a wide range of applications; skills and tools must be related to the particular application area at hand. We believe CODE fits a large set of widely different (mostly event-driven) system applications. It represents a general strategy that can be adapted to fit given application areas, which is the main role of the architect in the development process; one always has to decide on the most critical element or resource to be managed. CODE fits very well with modern concepts in Computer Engineering that is, engineering of the end-to-end computer system as opposed to piece-by-piece, bringing non-functional requirements to the forefront and not at least, working closely with the user in the development process[85].

We pointed out in the introduction that creativity remains one of the main ingredients for good engineering in general (SWA in particular). CODE has two levels of creativity:

1. At a high level, the architect has to understand users and designers at the same time; this way he is able to create systems that meet both user needs and developer skills.
2. At a lower level, creativity is strictly related to engineering, and is applied mostly on the design and implementation level.

In addition, the architect should always consider the possibility of generalization, in the sense of creating a framework from which one could later develop/customize versions of the system capable to satisfy different needs. Basic components, that could be reused in other systems need to be defined

---

[84] The business school mentality is that everything can be managed the same way; the equivalent engineering mentality is that everything can be engineered the same way. There is a commonality indeed, but only in the approach to the matter – in our case this is the coordination-based perspective. We use specific ways to design the units.

[85] CODE involves the user at all development stages except coding; this is in contrast to the usual development process where the user is only involved in the requirement acquisition phase.

from an architectural perspective (with respect to our case study, basic hospitals may have different needs – fit different clinics).

Many of the above concepts have been illustrated by a case study in which we set out to architect a hospital information system. Without going into design or implementation details, the intent of this example was to illustrate that CODE concepts could be applied in a systematic way. We focused on the main steps in the requirement specification and architectural phases, as well as their iterative character.

It was shown that one could start with a relatively undefined idea of how the system would be implemented (which is usually the case with a complex system) and then successively refine it with users, which bring in different expertise. There have been two levels of user involvement in developing the Hospital Information System:

- A first level of discussions was between the architect and the hospital managers. At this level it was decided that the resource to be managed is the patient database; the overall system partitioning has also been defined.
- On a second level, the architect along with development team leaders discussed each hospital unit with specialized users of these units. Users could be matched with developers who are more knowledgeable in areas related to a particular hospital unit (for instance somebody with database and GUI development expertise be assigned to the registry unit, while somebody with real-time systems or data acquisition expertise be assigned to the ICU).

The case study hopefully indicated that if different people come up with the same critical resources within the suggested architecture, the overall module partitioning turns out alike.. In light of our previous discussions on empiricism vs. formalism, we consider the case study as being the empirical[86] part of the thesis. It suggests that CODE allows for multidisciplinary design, where one could closely work with the user before the implementation stage and at the same time fit a well-defined development process.

[86] The PAD/PAL representations could be considered the formal part of CODE. However, we are not using them to formalize the requirements or the architecture. Instead, they are used to validate decisions and later, design verification.

Chapter 5

A BEHAVIORAL VIEW OF SWA STYLES

As outlined in the introduction, the 'science' of SW architecture is still a very active field of research. We think the word 'science' is appropriated since most of the activities going on in the SWA field, focus on the study, classification, experimentation or evaluation of existing architectures. The architectural styles mentioned so far are a direct result of these efforts. More work is needed though in order to define and fully use the architectural perspective on SW. It all starts with a paradigm shift, from the algorithmic approach used on the lower levels of the SW design process, to an interaction (and as we suggested, coordination) based approach to SW on the architectural level. This comes along the technology shift experienced this decade, which took us from local to distributed computing by and large[87]. It becomes more and more obvious that interactive systems could become much more powerful problem-solving engines than the algorithmic ones; as a consequence, computing technology paradigms centered around interaction, make their way from embedded systems (once their designated area) to more general-purpose computing.

Let us also note that, just like in the case of SW engineering, current approaches to SWA emphasize the engineering aspect, the solution, rather than the problem it solves. There are authors [33][34] who consider understanding and classifying SW development problems as important as investigating their solutions. Without going into details about problem architectures, we try to take a similar stand in this chapter, as a first step to approaching styles in SWA. With enough incentives to reconsider the approach to SWA in general and styles in particular, we will base our view on SWA styles on the behavioral aspect.

[87] This shift is seen in terms of both computing platforms (mainframes vs. workstations) and programming techniques (procedural to object oriented and then distributed programming).

Behavior as a base for SWA styles

Most of the styles listed so far were actually samples of architectures resulted from partitioning particular applications into modules and connecting them in a suitable way. In spite of the relatively large number of styles, mentioned by as many authors, we believe there are basically two categories of SWA styles, based on the way the work is done:

1. *Flow oriented*, where the execution steps follow in sequence, usually with no return of data as soon as it has been processed once[88].
2. *Call oriented*, where components call another to execute work, data being passed around as needed. By the way the calls are made, call oriented architectures can be:
   - *Unstructured*, in which case, any component can virtually call any other component (examples include instances of Object Oriented or Object Based, and even Client-Server organizations)
   - *Structured*, where calls are made only on designated channels, most often on a hierarchical basis (examples here include layered systems, main program and subroutines, coordinated systems by and large)[89]

The above classification is only a first approach and still does not take into account the type of application that is being targeted, especially with respect to the environment in which the system will have to perform. We will go one step further by actually identifying the needs that lead to a particular architectural style and then build our taxonomy mainly based on this criterion.

From the application perspective, we think SW architectures are geared towards either *static* or *dynamic* environments. We consider these two types of applications with respect to the degree to which changes in the environment are relevant to the system behavior (in other words, they are a measure of the 'awareness' of the system with respect to evolution of the environment)[90]. To better define this perspective we generally consider that any system to be designed has to meet two sets of requirements:

---

[88] We will see later that feedback loops within flow oriented architectures could be looked at as an exception from this rule.

[89] The structure is considered implicit in the case of flow oriented styles.

[90] Changes in a *'static'* (although, by any other criteria, maybe highly dynamic) environment barely exert any constraints on the system. By making the same system more dependent on changes of exactly the same environment, one could now see this environment as *'dynamic'* from the perspective of our new application.

- *design requirements* which support the desired functional properties of the system and which, to a large extent, are under the designer's control
- *system constraints* which conflict with the design requirements, restricting the design space; most often, these are associated to the environment around the system and are barely controllable by the designer

From a design perspective, system constraints are fixed (or at least predictable) if running in a static environment; once they have been taken into consideration at design time, they become part of the expected operation of the system (one can say operation is deterministic). By contrast, systems running in dynamic environments are characterized by constraints changing at run time; the changes in the environment are not entirely predictable, therefore the system should be able to handle exception situations, out of the initial design. Many times, the systems performing in a dynamic environment have to cope with incomplete information or are required to handle (and often prioritize) concurrent events, therefore yield partial solutions.

To a great extent, most of the SWA styles so far have been developed almost making abstraction of the environment where they were supposed to be instantiated, therefore developed intrinsically to handle events in static environments. In this case, the complexity of the development task revolves around data processing, the state of the system being decided to a great extent by the state of the data. As a direct consequence, most of the conceptual work around SW development followed the same guidelines (the only important exception being the case of embedded systems); from a real-time standpoint, one could call these systems 'single event response systems'. For systems placed in a dynamic (unpredictable) environments on the other hand, the complexity and focus reside in the decision-making, since we have here multiple event responses to handle; both the state of the data and the state of the environment therefore determine the state of the system. What is really important to note about this type of systems is that they make special provisions for exception handling (that is, handle events out of our design). This kind of systems could stand very well for 'multiple event response systems'. It is worth to mention that lately, design requirements and techniques belonging a few years ago exclusively to real-time systems, are making their way into the 'general purpose' SW. This trend is a consequence of the increasing demands (and consequently complexity) put on today's general purpose SW, where decision making plays an increasingly important role.

Given the reasons above, we grouped SWA styles into two main categories:

- for *static/semistatic environments* (with two subgroups: flow oriented and call oriented architectures)
- for *dynamic environments* (with either central or distributed coordination)

To the best of our knowledge, none of the authors approaching the SWA field have taken a similar standpoint, although Alexander's way to describe patterns for building architecture may somehow suggest it [6][91]. In our opinion, just listing a number of SWA styles is not enough. In line with the multiple view approach on SWA (promoted in the first chapter), styles should be looked at in the same manner. For instance, it is worth taking an application-oriented perspective of SWA styles, even only by comparison with other established engineering disciplines (for instance building architecture[92]). On the other hand, when talking of SWA styles, one cannot consider strictly one particular application, since that application would be too narrow to be captured in a style on its own. It is useful to come up with families of applications first, then estimate what kind of architecture would suit them better. Only further from here, specialization becomes important; looking at established branches of engineering, one can easily see that they are all specialized (take for example bridge as opposed to house construction in civil engineering or steel structures which are different from engine construction in mechanical engineering). Significant steps in this direction have been taken in SWA as well, with DSSA at one end and frameworks at the other. More integration is needed though. Ad-hoc solutions attempted so far in SW development (and considered in the literature most of the time) may not prove always useful, for lack of a long enough history to validate them.

Our initial classification (see appendix) will be further refined in the coming pages, where we will proceed with a more detailed description of each architectural style considered separately. More coverage was reserved for some of the styles, deemed to be more popular than the others (take the case of the layered architecture, which has been detailed more than event systems for instance).

---

[91] We will discuss later in this chapter the relation between SW patterns SWA styles. Let us mention for now that SWA styles are different from patterns at least for their relatively high level of generality, which is specific to architectures but does not apply to patterns.

[92] When people think of building a house, they take into consideration factors like the climate where the house would be located (for instance, tropical or arctic), intended use (private or public) and many others.

Architectures for static environments

We consider an environment to be *'static'* if, from the system's perspective, changes in the environment are either transparent (that is, do not affect or are of no interest to the system) or are known ahead of time and could be taken into account by design. It is our belief that in a static/predictable environment, the complexity and focus reside in data processing. Each application can be architected alone, on a single event response basis and generally the state of the system is determined by the state of the data. In light of the arguments brought forward when we discussed about coordination, these architectures have a pronounced computational character, coordination being either implicit or of a very low emphasis.

Both flow and call based architectures are considered to be suitable to such an environment. With respect to the latter, and in strong contrast with dynamic environments, static environments could, to some extent, accommodate call-based architectures that are not structured.

Flow oriented architectures

We grouped under this heading the *'control flow'* and the *'data flow'* architectures. Let's try and establish some common characteristics of the two types. First of all, flow oriented architectures are about processing data streams in a case where processing has to be done in a specific order. Different input sources may be available, as well as various ways to store the data (considering intermediate results as well). Given this context, the solution is to divide the task of the system into several sequential steps providing a structure that serially processes a stream of data. Usually (with the exception of interpreters), the output of a processing step serves as input for the next one (by analogy with an assembly line); in this case, the input supplying the system is called *data source*, while the output is flowing into a *data sink*. A processing element is called to be *passive* in a case where the subsequent element pulls output or the previous element pushes input; if the processing element is tied in a loop, the structure is called *active*.

Flow architectures are able to perform mostly in static environments, given their 'hard-coded' coordination and limited capability for decision making. In the

following paragraphs we will discuss only the basic flow architectures (that is, single stream), under the observation that more complicated topologies (like concurrent flows or control loops) are also possible.

Data Flow

The 'data flow' computing model is based on incremental transformation of data (stream to stream) so that output begins before input is consumed. This leads to an execution control, which follows functional data relations, as opposed to a programmed sequence of operations. [41].

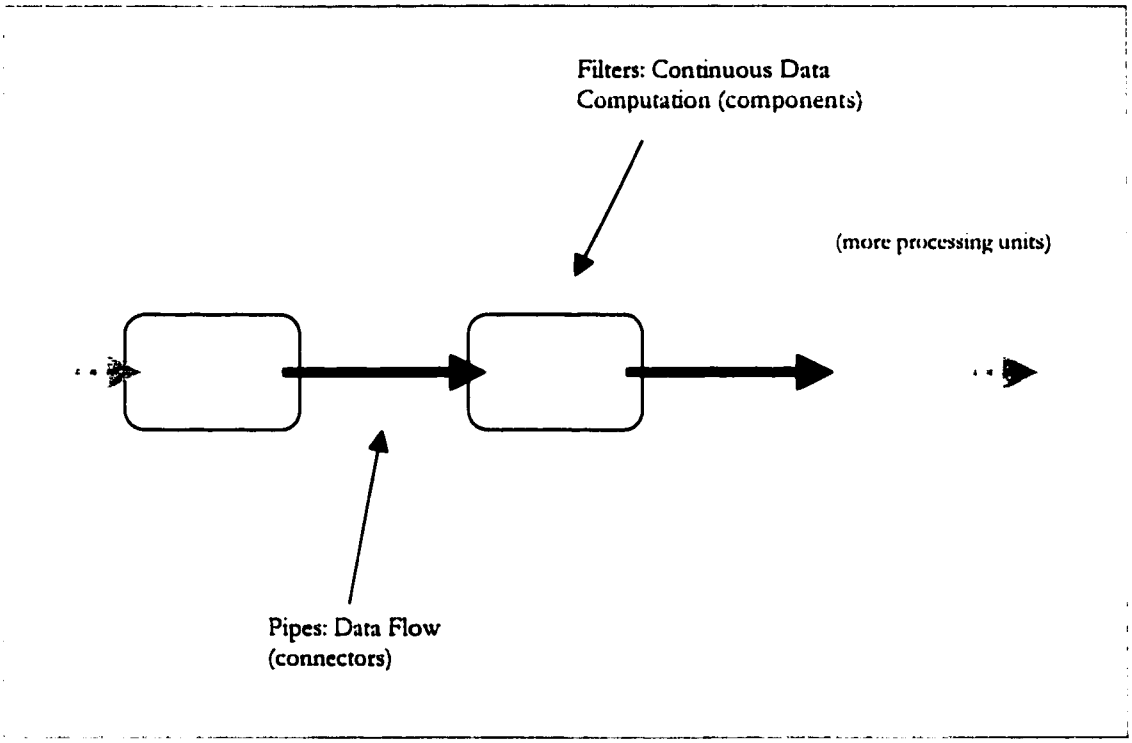
Pipes and Filters

The most popular architectural style here, derived from the data flow model, is called 'pipes and filters' itself. *Filters* are modules responsible for the local transformation of data, whereas the *pipes* connect the output of one filter with the input of the other. The sequence of filters combined with pipes is called *pipeline*. There is little or no context information utilized by the filters, hence no state information is preserved between data instantiations; filters do not know the identity (or even existence) of other filters upstream or downstream. Along the same lines, filters could be considered as independent entities; they do not share state with any of the other filters. As a general rule, it is highly desirable that the correctness of the data at the output of a filter network not be dependent on the order in which filters process the data.

[14] mentions a number of advantages and drawbacks of using this architectural style, based on standard aspects like system decomposition and analysis, performance, maintenance, etc. Amongst advantages we consider the following:

- in terms of system decomposition, the overall system function is cumulating the functions of its filters; there are instances where problems may be even hierarchically decomposed.
- in terms of maintenance and reuse, often a black box approach can be taken, this leading to ease of extension, modification and reuse. There is room for a great deal of flexibility given that the filters usually have a simple interface, allowing for their easy exchange.
- the performance of such systems can be enhanced by the use of concurrent execution or parallel processing

- specialized analysis is possible, such as for deadlock and throughput. What is even more important, rapid prototyping from existent filters is also possible, followed by incremental optimization



There are also disadvantages, which could be summarized within the same framework as above:

- problem decomposition for such systems encourages batch mentality. Once this achieved, filter ordering may be a tough decision. Building interactive applications is even harder, due to the directional flow of data.
- maintenance and reuse is hampered by the fact that this type of structure usually forces a common denominator of data, which may not be suitable for future applications; later implementation of data translation may lead to important performance hits

- given the stream of data between filters, performance is hard to model; often queuing theory is involved. Performance may also be degraded by parsing/unparsing overheads.
- sharing state information between processes may become expensive; if it is necessary to share large amounts of global data, this kind of architecture becomes inefficient.
- performance gains through parallel processing may prove to be an illusion: the cost of data transport could be high compared to doing the same computation in a single filter. One could add on top of this cost the complexity of synchronization as well as the fact that a filter could consume all the input before generating any output, which makes buffering a necessity in case of concurrency.
- error handling is extremely difficult; a common strategy for error recovery should be in place and restarting the whole pipeline may prove to be a dramatic decision in many of the applications.

Control Flow

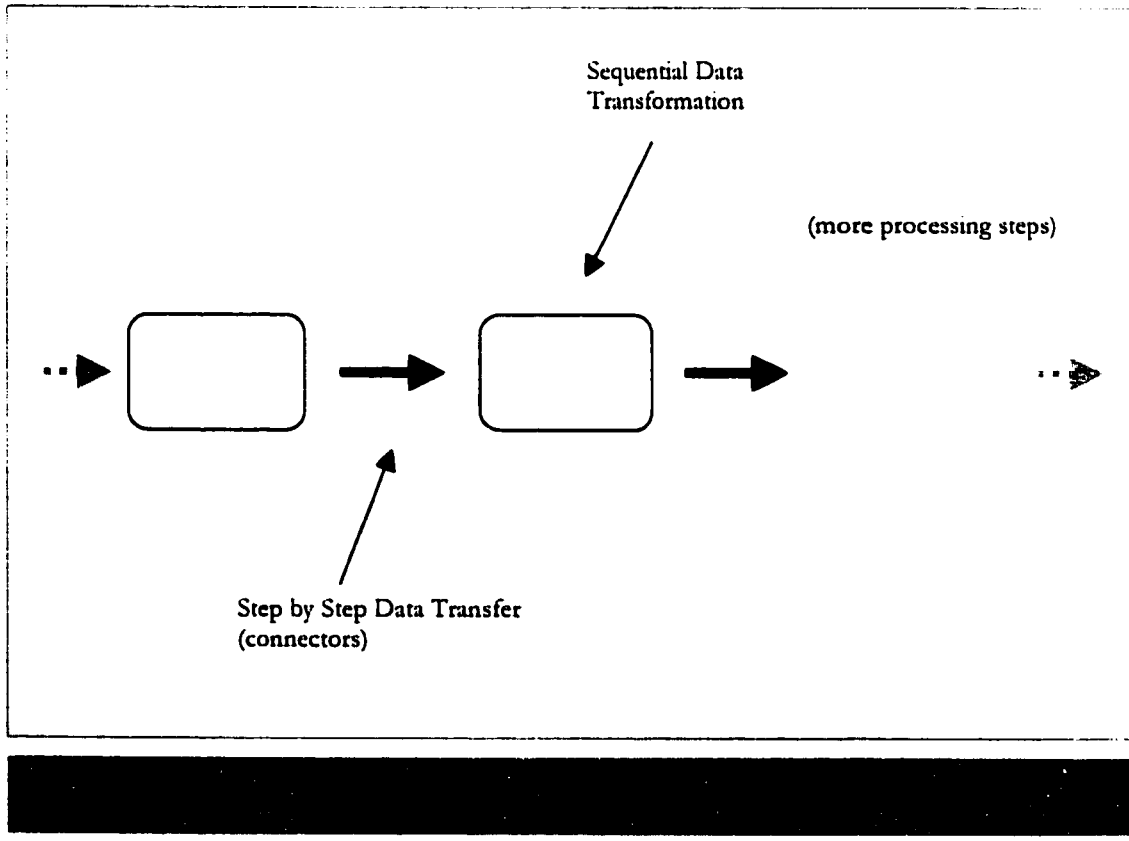
As mentioned in [41], the 'control flow' constructions can be divided into three categories:

- Sequential Control Flow
- Parallel Execution
- Multiflow Execution

Out of the three, the Sequential Control Flow is by far the most popular, partly due to the fact that it is a direct derivation of the von Neumann computing principle. The other two, making use of parallel execution, rely heavily on coordination mechanisms rarely available on the architectural level (usually performed by operating systems); we have already discussed aspects of parallel and multiflow execution along with the 'pipes and filters' model. By and large, one could think of two models of control flow: one where control flows from one component to the next in a serial fashion (lockstep operation) and another where control is exerted from an external component, on top of the serial processing. The control flow structures considered here include the batch sequential and interpreter architectures, both belonging to the sequential class.

Batch Sequential Architectures

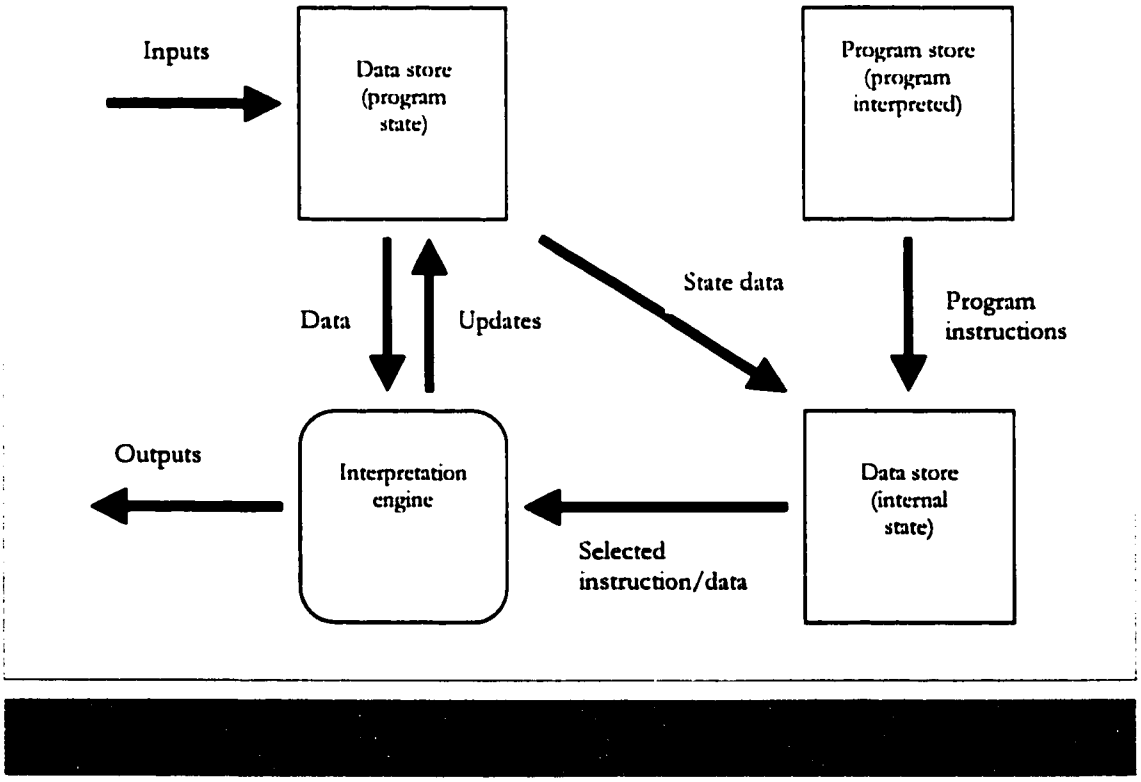
Batch sequential architectures are deemed to be a degenerated case of the pipes and filters, where all the input data is processed by a component before being delivered to the next one; in other words, each step runs to completion before the next one begins. Data transformation steps are incorporated by independent programs, therefore data is transmitted between steps only in large chunks and not necessarily through permanent communication channels. We consider this particular architecture as being of a control flow type since the execution control is decided by data readiness for the next step; in our view, the (interrupted) flow of data is not a characteristic of this style. Typical applications here include program development (compilers) and classical data processing.



It is worth to note that this type of sequential processing later evolved to the repository model. We postpone discussing this transition until we had talked about repositories as well.

Interpreters

Interpreters gained a lot of popularity being used as execution engines in SW, with the intent to implement a virtual machine. They perform a sequenced processing based on the state of the data, rather than a pre-determined scheme (as it is the case with the batch-sequential style). To some extent, we see interpreters as building on top of batch-sequential architectures by adding a control loop containing the state of the computation.



Basically an interpreter consists of an interpretation engine and the entity (program) to be interpreted. Data is three-fold in this case: the entity being interpreted, the state data of this entity and the state data of the interpreter itself. The interpretation engine is the one to carry all the control. The components of

the interpreter could be further refined in order to fit particular applications and degrees of complexity. Rule-based systems are one of the most common developments of this style.

Call oriented architectures

Call oriented are by far the most popular styles used in SW design; we grouped here the hierarchical layers organization as well as main programs with subroutines. As the name might suggest, the underlying principle of this type of architectures is the use of hierarchical based calls for interaction within the system. As opposed to flow-oriented architectures, which are all coordinated, the call oriented may be coordinated or not.

Layers

The layered style structures applications that can be decomposed into groups of related subtasks, each group on a particular level of functionality. In the case of large systems that require decomposition, this is a tremendously popular solution. There are a number of factors, which indicate that a layered architecture would better suit a particular application; based again on [14] estimates, some of these factors are:

- two orthogonal hierarchy structures can be identified: a vertical hierarchy independent of the horizontal structure. Computations on the same level are still highly independent one from each other.
- a mix of high and low level requirements, where the higher requirements rely on the low level ones;
- code changes are not allowed to ripple through the system
- parts of the system should be replaceable, with well defined interfaces
- some of the responsibilities are to be grouped
- no standard component granularity

Layered systems have basically a hierarchical organization where each layer provides various facilities (usually hiding the lower layer). We could actually look to both a vertical and a horizontal decomposition: vertical organization is hierarchical, horizontal organization is peer-to-peer or independent.

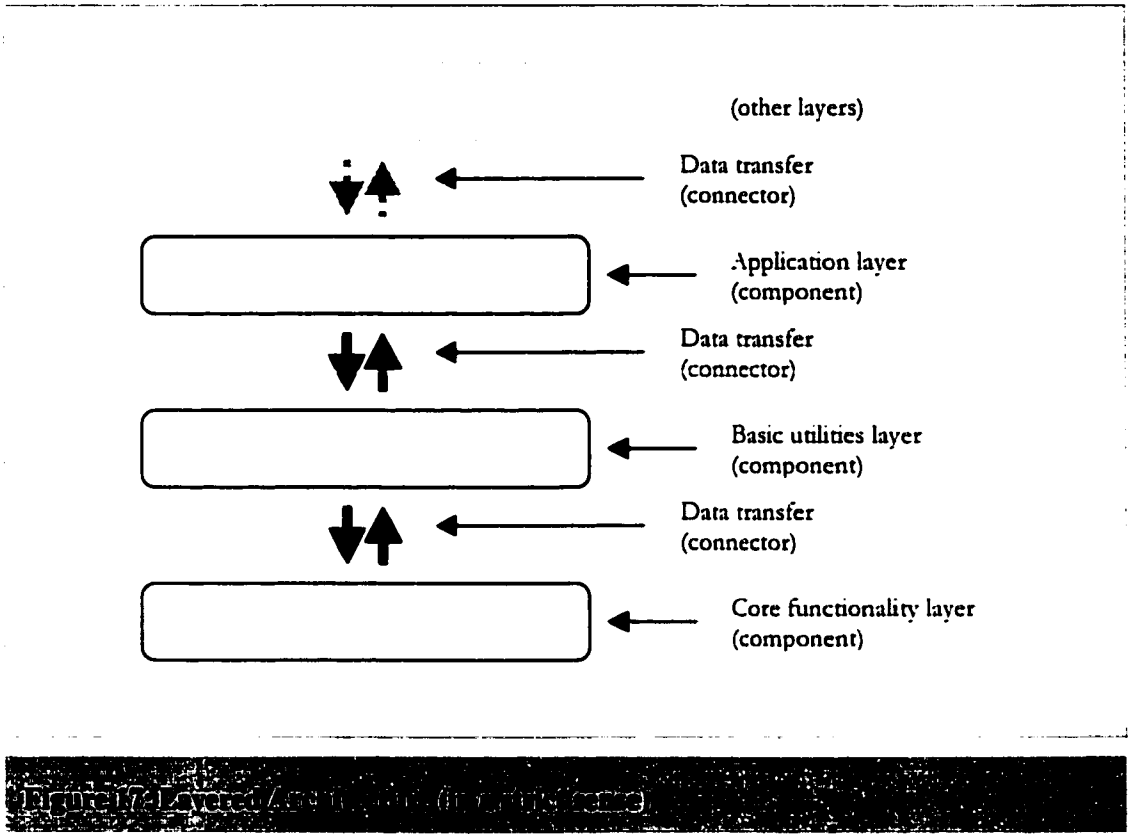


Figure 7.1 Layered Architecture (Interface Focus)

Characteristic for this type of structure is the fact that a particular layer relies on services of the layers below; a specific layer transmits requests to the layers underneath and provides services to the layer(s) above. One could talk here about an interleaved client-server organization, a particular layer being a client to the layer below and at the same time a server to the layer above. Connectors are implemented by protocols defining how the layers would interact.

Various scoping regimes can be implemented; that is, layers could be either opaque or transparent (or in-between). Traditional layered architectures consist of *opaque* layers, which limit the client-server relationship only to adjacent layers; in other words a particular layer cannot pipe through services from the layer above to the layer below. At the other end are the *transparent* layers, which can provide communication services between non-adjacent layers. The level of transparency could be modulated in the case of *semi-transparent* layers where only specific services are piped through. For practical reasons though, the extent of the scoping is usually limited to adjacent layers; in order to preserve a minimum coupling level, interaction is only restricted to the next layer above or below.

With the main benefit of supporting design based on increasing levels of abstraction, there are other significant advantages connected to the layered structure:

- with a proper design, dependencies are kept local (to the upper/bottom layer); this allows implementers to partition a complex problem into a sequence of incremental steps.
- ease of testability since layers can be tested separately, especially if highly de-coupled
- ease of reuse, provided an individual layer is properly abstracted and has well defined interfaces
- support for standardization in a case where a proper level of abstraction has been achieved and commonly accepted (standardized tasks and interfaces would be used)
- layers could be easily exchangeable, as a consequence of local dependencies and standardization of interfaces; this allows different implementors to build on standard interfaces hiding different implementations of the same layer.

Counting the drawbacks, one could mention:

- not all the systems could be easily implemented as layers
- it is difficult to establish the right level of abstraction and their granularity; finding the abstractions themselves may prove challenging – once the abstractions are wrong, layers have to be bridged, which ruins the model
- performance considerations may often require a closer coupling between high level functions and their low level collaborators; efficiency is affected if data (from lower layers) has to travel all the way up (and maybe also be transformed along the way)
- as soon as the behavior of a layer changes, behavior changes may be cascaded to related layers, if they are not sufficiently de-coupled
- duplication of work if layers executing similar tasks are far apart

Without the intent of a detailed comparison between CODE and the layered architecture, virtually none of the drawbacks above applies to CODE. Our architecture and its associated development steps allow for easy module implementation. Performance and efficiency issues are alleviated by communication only at the coordinator level, while duplication of work is essentially avoided by the event response oriented structure. It is true that a layered architecture of coordinators could inherit some of the drawbacks above, however in the case of CODE the layers are always properly decoupled.

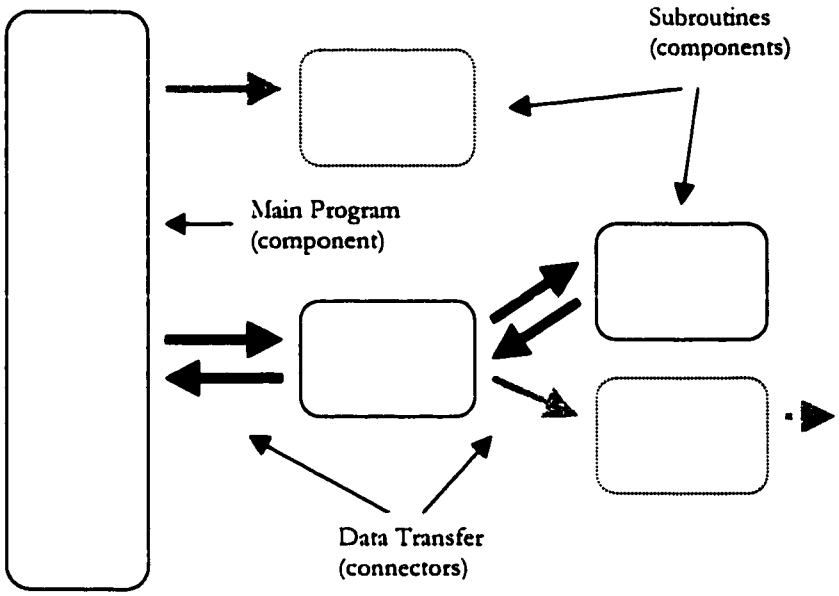
Some of the best examples of layered architectures are found in Telecommunication applications where often enough the SW is structured following the OSI layers. For general-purpose systems, a three-layer architecture is commonly used, where the bottom layer is hardware/operating system, on top of which there is a layer providing support for portability, the upper layer(s) being a virtual machine. Virtual machines considered alone, if implemented as a layered structure, often have their inner layers hidden from all but the adjacent outer layer (other than certain functions selected to be exported).

Main Program and Subroutines

The 'main program and subroutines' is a classical (maybe the oldest) programming paradigm, based on functional decomposition. Beyond historical reasons though, the fact is that hierarchical decomposition (composition) and reasoning go along well with this style:

- from a *decomposition* perspective, routines can all be seen as subordinated to the main program (directly or indirectly)
- *composition* comes into play in the case of aggregating more routines into modules, in which way the system structure becomes implicit. On a more abstract level, hierarchical composition leaves room to hierarchical reasoning: correctness of the main program relies on the correctness of the subordinated routines.

This style provides very well for central coordination, the control thread (coordinator) being the main program while the routine modules could shape the working units. A main program and subroutines structure is unlikely to be achieved on the architectural level since it is relying very much on the modularization support offered by the language itself. Given this fact, we come too close to the actual implementation and lose the architectural perspective; this is the reason for the limited space allotted by us to this style.



Main Program and Subroutine Architecture

Architectures for dynamic environments

As already mentioned before, there are architectures meant to perform in an unpredictable environment, that is capable to handle situations out of the initial design. In such context the focus shifts towards the decision making rather than the computational aspect; consequently, coordination plays a paramount role and is probably the best criteria to classify such systems. In our opinion there are two main groups of architectural styles in this category: with *central coordination* and *distributed coordination.*

Central Coordination

Characteristic for centrally coordinated systems is the existence of a (central) coordinator to delegate work and eventually handle events. A piece of program or simply shared data could play the role of the coordinator. One could even use

a hierarchical structure of coordinators; in this case, the hierarchy is being established by the way the coordinators communicate, communication being made between known recipients, of course. We grouped under central coordination the blackboard and the so-called 'communicating processes' architectures.

Blackboard

The blackboard architecture is actually a particular instance of the larger class called 'data centered systems'[93]. The main component in such an architecture is a central data structure (which eventually is the only one to decide the state of the system) and a collection of independent components that operate on the data structure. In a case where the state of the central data structure is the one to continuously decide which processes to execute, the data centered structure is actually called a 'blackboard'. Blackboard architectures are geared especially towards AI systems.

According to [14], a blackboard architecture consists of several specialized subsystems assembling their knowledge in order to build (even) a partial or approximate solution of a particular problem. The aim is to find an optimal solution in most cases and at least a sub-optimal (rather than no solution) for the rest. This approach proves useful where no deterministic solution strategies are known or a complete search for the solution is not feasible. Such an approach is called *opportunistic* problem solving, and is in strong contrast to the *deterministic* approach where the solution steps are known and their arrangement/activation is hard-coded.

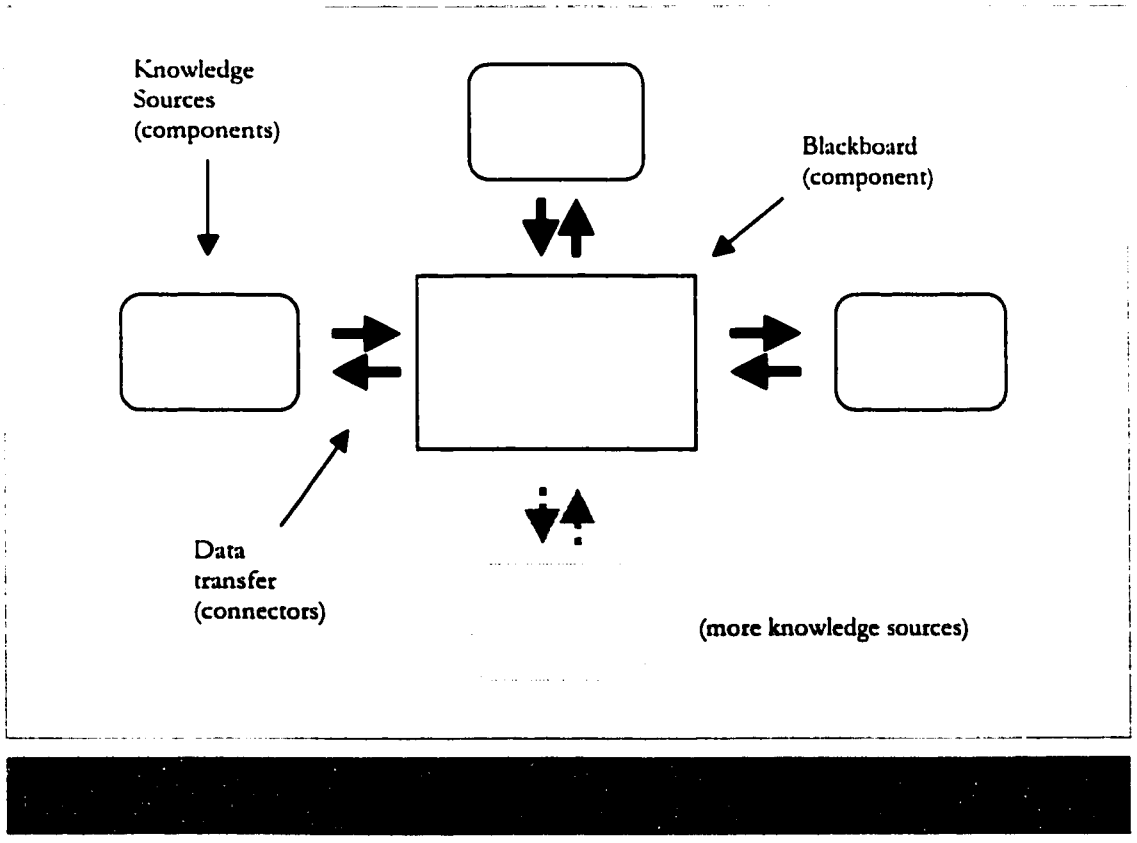
The independent, highly specialized programs are named knowledge sources; they jointly work on the common blackboard data structure. Although working independently, the knowledge sources do cooperate in putting the results together. The main reason to placing the blackboard model under central coordination is the existence of a control component to evaluate the progress on the blackboard, based on the state of the central data. D. Garlan summarizes the components and functionality of the blackboard as follows [28]:

---

[93] The common data centered structure is the classical database where the selection of processes to execute is made by the type of processing required to the incoming data (and usually indicated in the data stream itself). After processing, data is stored away. By and large, this type of organization is called a 'repository'.

- the *knowledge sources* are independent pieces of application-dependent knowledge; what is significant for the blackboard structure is that the only way of communication amongst the knowledge sources is through the blackboard itself.
- the *blackboard* data structure is actually state data hierarchically organized depending on the application. The knowledge sources make changes to the data so that, incrementally, a solution is reached (eventually).

Coordination is entirely driven by the state of the blackboard. The control could be either implemented as a separate component, or distributed within the knowledge sources, or something in between. Regardless of the implementation, what is really important here is that the only reference for control is the state of data on the blackboard.



Usually no closed approach to a solution is known or feasible, and the knowledge sources need to experiment with different algorithms for the same subtask. These various algorithms could solve partial problems, therefore uncertain or approximate solutions are involved. The fact that the input, intermediate, and final data may each have their own representation could set the grounds to exploit parallelism, if possible. Within their restricted domain of use, blackboard architectures exhibit the following benefits:

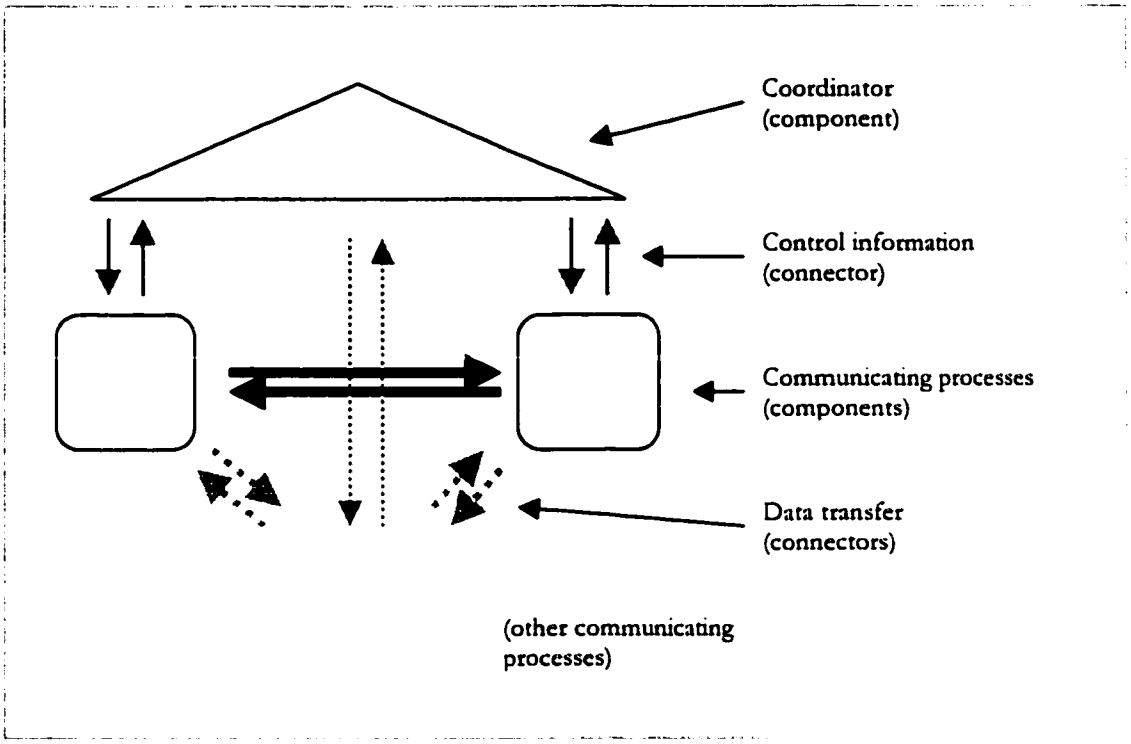
- in place of a complete search of the solution space, experimentation with different algorithms is possible, even for partial solutions. This makes blackboards a useful tool for modelling..
- support for maintainability, interoperability and reuse since the knowledge sources are independent (amongst themselves and from the control component)
- support for fault tolerance and robustness - only strongly supported hypothesis are accepted (therefore, immunity to noisy or corrupted data for instance)

The drawbacks are mostly related to issues like

- difficulty in finding an optimal control strategy, associated to the difficulty of testing the behavior of the system; these result in a high development effort and high costs, as a consequence
- no optimal solution is guaranteed
- low efficiency (overhead with finding sub-optimal solutions and proving them)
- little support for parallelism (concurrent access to central data)

Communicating Processes

In the case of communicating processes, independent processes take the place of components. Processes are centrally coordinated but still able to cooperate on a peer-to-peer basis, in order to complete a certain task. The fact that the processes are allowed to directly communicate amongst themselves makes a major difference from Multiactivity, where all the inter-process communication relies on the coordinator. Communication is mainly performed via message passing (synchronous or asynchronous).



Messaging is mainly point-to-point, the most common example being the remote procedure calls (RPC). Because of the unrestricted communication between components, we consider this particular style as being an undisciplined way to implement the Multiactivity paradigm; nevertheless, it seems to be common use in practice.

Distributed coordination

Distributed coordination systems make use of local coordinators to handle events and delegate work. Overall communication is being performed only amongst local coordinators on a peer-to-peer basis. Distributed systems are systems designed to use resources available on a network, therefore needing radically different SW than centralized systems. The development of SW for distributed systems was driven by two major trends in the HW technology:

- multiprocessing introduced especially by multi-CPU computer systems running multiprocessor operating systems

- LAN/WAN proliferation, connecting multiple heterogeneous machines

Distributed systems are backed up by compelling economic reasons, the most important being:

- better price/performance ratios than centralized computing systems (mainframes)
- higher performance and scalability
- higher reliability

In the general context of distributed computing, and from a historical perspective, one could define three development stages in terms of the communication models being used:

- traditional Inter-Process Communication (IPC) mechanisms: shared memory, pipes, queues. Being quite deeply involved at the OS level, these provided little or no room for interoperability
- Remote Procedure Call (RPC) type libraries with their associated environment; this way more freedom was provided for design at the application level but interoperability was an issue (from various vendors, each implementation having its own clues)
- higher level abstractions brought by standardized platforms like CORBA or OLE. Interoperability starts to come along with concepts like distributed management of resources, object oriented encapsulation/inheritance and finally distributed management of resources

The two architectural styles that have been grouped under this heading are 'event systems' and repositories.

Event Systems

In the case of event systems, the components are still independent processes, communicating on a peer-to-peer basis. Communication though is based on implicit invocation rather than explicit (that is, recipients are not necessarily known). Processes have interfaces to define allowable incoming/outgoing events; these interfaces provide for both a collection of procedures and the set of corresponding events. The idea behind implicit communication is based on event-procedure bindings: procedures are registered with specific events and the components interact by 'announcing' events. Instead of directly invoking a procedure, a component can broadcast one or more events, while other

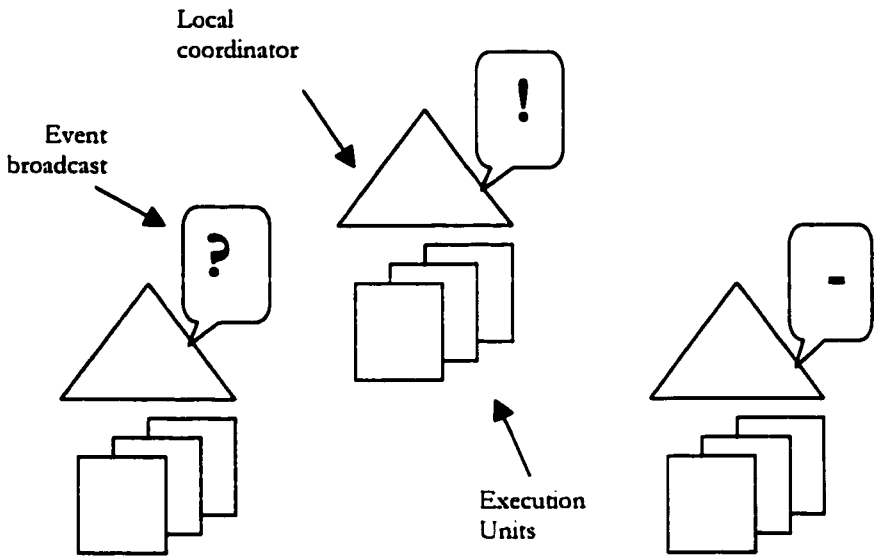
components on the system can register their interest with certain events and associate procedures with them. Upon receiving one of the events of interest, its associated procedures are implicitly invoked; thus an event announcement causes the implicit invocation of procedures in other modules. Since a central coordination is missing, the order of invocation is non-deterministic; more sophisticated event systems could make use of bidding techniques in order to control procedure invocation. In addition, there is still room for procedures to be called directly.

What should really be highlighted here is that, in general, event announcers have no idea which components will be affected by the events; there is no indication on what is the processing that will occur at run time, more important, what the order of processing will be. In any case, there are a number of advantages associated to this type of architecture:

- intrinsic support for separating coordination from execution, in spite of not having a central coordinator. At least conceptually, we have on one hand the event generation and on the other hand the event handling. What has to be done can easily be partitioned in event responses as a foundation to further development.
- system maintenance and reuse benefit from integration of new components by simply registering them. Due to the lack of static name dependencies, ease of component replacement can also be mentioned.
- performance can be enhanced by parallel invocations

Disadvantages of event systems are mostly related to the following aspects:

- problem decomposition is made difficult by the lack of control over the order of invocation (that is, order of processing); therefore correctness is difficult to ensure, as well as the data exchange (if data is not passed with the event, a common repository is needed in which case resource management becomes an important issue).
- maintenance may become problematic if a centralized register of events, registration and dispatch policies is not available, so that activity can be tracked
- performance is penalized by the indirect communication/data transfer



Repositories

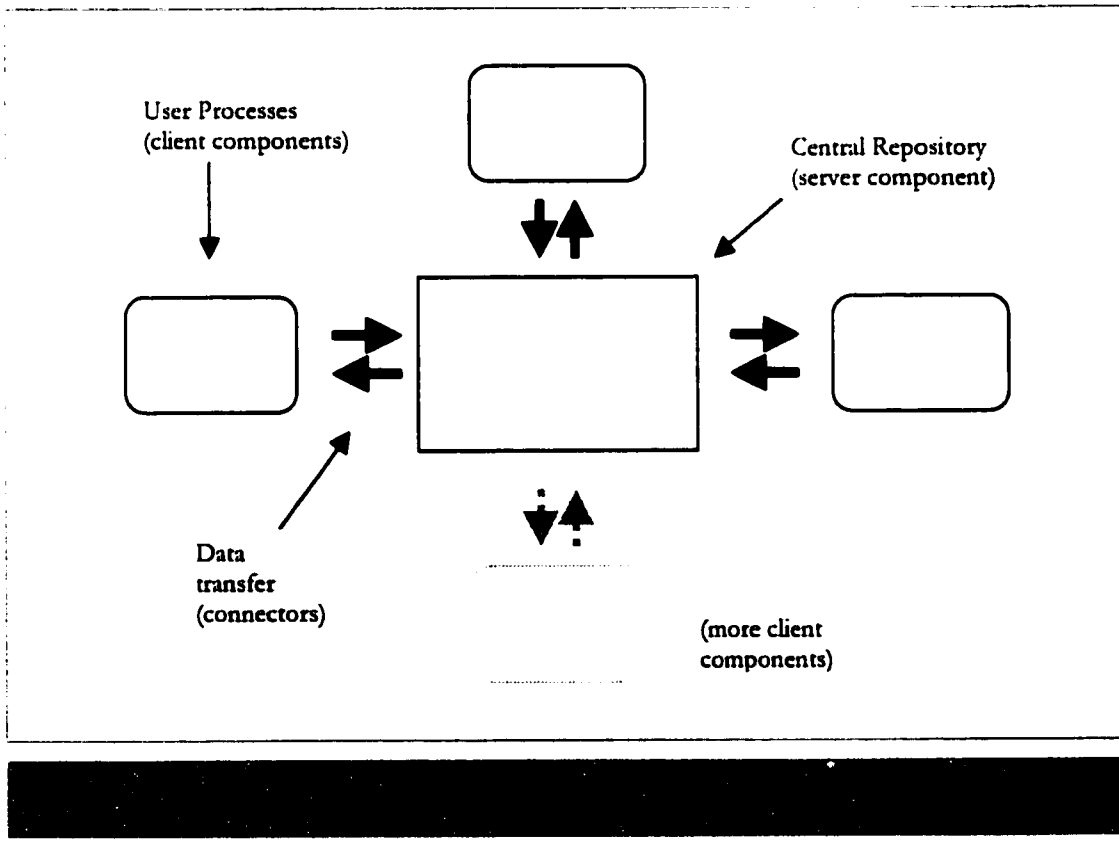
The repository style was already introduced in this thesis as a 'data centered' structure, along with blackboards. Its components are the central data structure along with a number of independent entities that work on the data (a client-server concept, by and large). In contrast with blackboards though, in this case, the incoming transactions determine which processes to execute and not the state of the data structure. An oversimplified view of a repository looks essentially the same with the one of a blackboard, therefore more detail is needed so as to differentiate them. We will consider the most common example of a repository, which is the transactional database. Databases have been generally shaped around three major components:

- an *internal* component which contains the physical storage (and being concerned with the way data is stored) along with mechanisms to handle concurrent access.

- *external* components which are close to the users and are concerned with the way data is viewed by its users
- *intermediate* components which act as levels of indirection between the two above

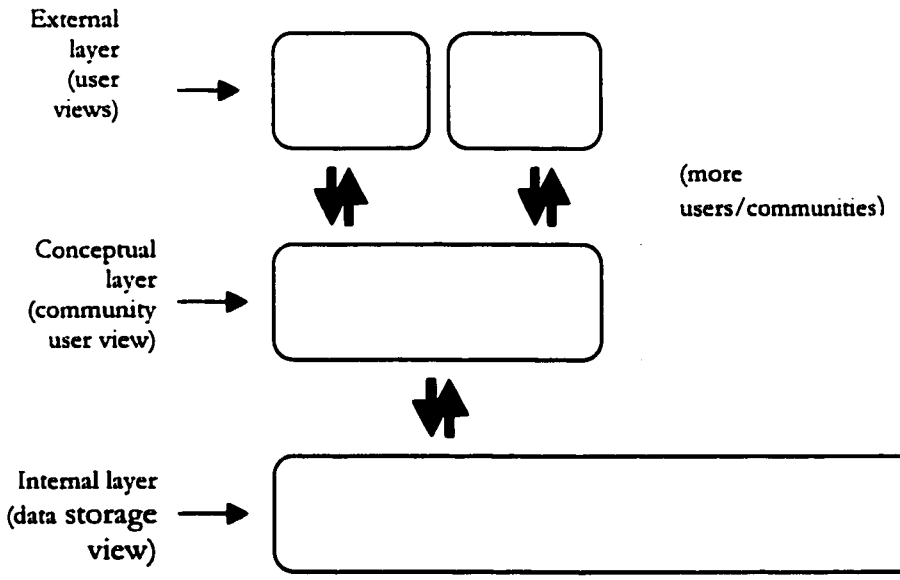
The last two sets of components are grouped as clients around the data storage.

If the external components are exclusively concerned with individual user views of the data, the intermediate level is meant to provide a conceptual view for communities of users. In other words, the intermediate components provide similarly abstract views of the data, similarities going from representation to data localization (most users are not interested in all the data in the storage, but in parts of it). The internal component though keeps precisely one internal view of data, representing the entire database, as it is physically stored. Requests to query or modify the database are generally handled by the intermediate components and then submitted to the actual database. These components can be associated actually with the processes running around the central data storage.



Coordination of the database as a whole is therefore distributed around these processes, in contrast to the blackboard model where coordination was embedded within the data stored. The distribution of coordination is even more eloquent in the case of distributed databases. Given the large extent and variety of this field, it is hard to abstract a more detailed model for this type of repositories.

Without going into further details, we prefer to now bring to attention another interesting aspect of repositories, and that is their complimentary view as both distributed coordination systems (already outlined above) and layered or sequential systems. The layered view might be readily suggested by the initial partition into external, intermediate and internal view components. Starting from the bottom, one can consider the data store itself as a continuous layer, on top of which resides an intermediate layer horizontally segmented (with segments being the intermediate view components).



Above each intermediate component resides another layer of external view components, which is also horizontally segmented according to the number of users. This layered model is consistent with the restriction that layer interactions be limited only to adjacent layers.

At last, the sequential view has to do in its own with the evolution of SW systems. Databases are some of the earliest SW applications; these applications consisted of a small number of stand-alone programs executing batch operations on flat files. The steps were independent from each other and had to run in a predefined sequence, which makes for a typical batch-sequential architecture. Of course the querying capabilities were limited. The requirement for interactivity forced the evolution of this model to one that would allow online queries (eventually multi-user) as well as data processing. This trend finally lead to the current model where one could say data is still processed sequentially but it is also buffered to allow for batch transactions or make it available for queries. Essentially the batch sequential model is still in place at the intermediate level.

Other distributed systems

With the evolution of network computing, distributed applications became of high profile amongst other SW systems. At this time, we think this area is still too dynamic in order to draw any architectural styles out of it. Nevertheless, there are – at least – some structures that emerged as widely accepted based on their topology (star or ring for instance), or communication mechanisms (for example broadcast or heartbeat). The expansion of distributed applications also redefined the old concept of client-server in the sense of solely the clients knowing the identity of the server; the latter does not know the identity of its clients ahead of time.

SWA styles and patterns

The basic idea behind patterns is that common idioms are found repeatedly in SW designs and that these patterns should be made explicit and applied appropriately to similar problems. There are three fundamental requirements for specifying and then reusing patterns:

- the design domain should be well understood
- some support for encapsulation of design elements should be in place
- a collection of well known and proven design idioms available

Architectural styles relate closely to design patterns in two ways:

- styles can be viewed as collections of related patterns (or pattern languages providing a vocabulary and a framework to build specific applications)
- for a given style there might be a set of idiomatic uses (as micro-architectures or architectural design patterns)

Patterns and architectural styles are complimentary mechanisms to encapsulate design expertise[94]. SWA styles provide the building block design elements along with rules and constraints for composition (eventually tools for analysis); styles usually provide guidance for building a broad class of architectures in a specific domain. Patterns on the other side focus on solving smaller and more specific problems usually within a given style (sometime across more styles). Patterns could rarely make it to the architectural level; their architectural presence is mostly restricted to DSSA in the form of frameworks. There are authors [14] who use the term 'architectural patterns' interchangeably with 'SWA styles'. We insist on the fact that the only expression of patterns on the architectural level is in frameworks which, being so close to a finite product, are substantially different from styles. So far, most of the documented patterns deal with solutions closer to the implementation level rather than system structuring issues. Only a few of these patterns though could have applicability on the architectural level as well[95], having little to do though with an architectural style. Thus frameworks and design patterns are just instances of the more general class of 'design' patterns. As a commonality though, patterns allow a designer to reason and make decisions on a set of designs (just like SWA) in order to pick the one that best suits the application.

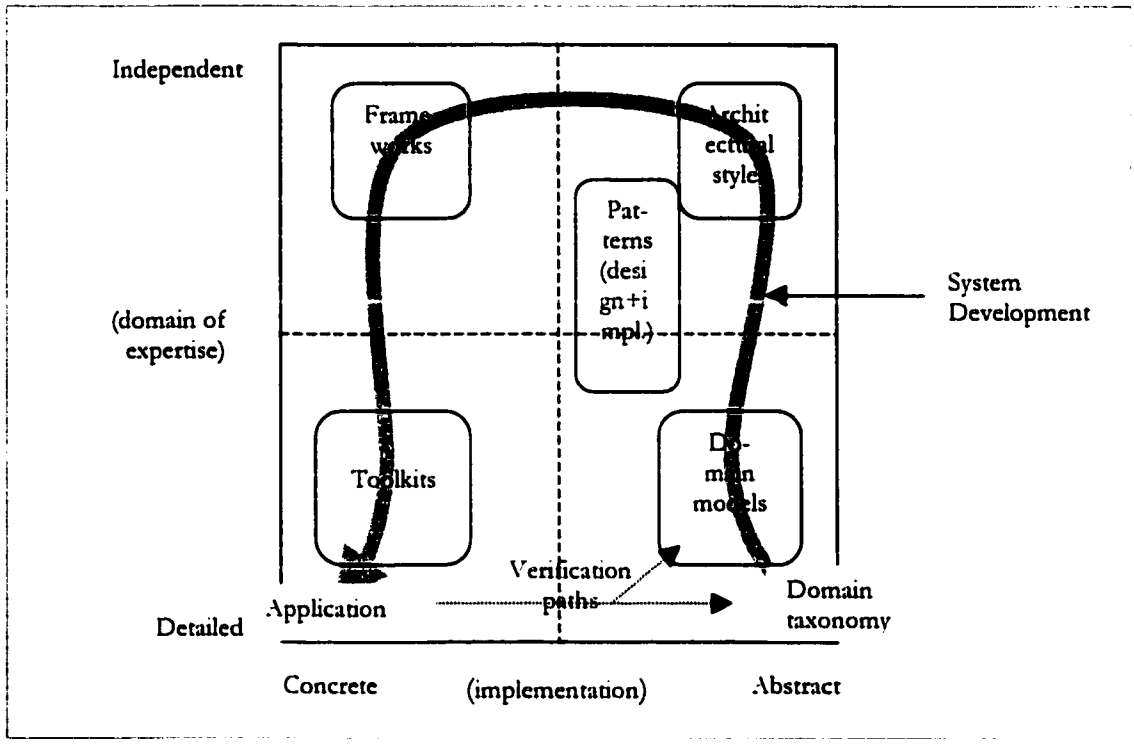
Figure 24 (based on [59]) illustrates very well the place of design patterns compared to SWA styles and amongst other popular technologies in SW development. SWA styles are basically domain independent, describing a system in highly abstract terms. Patterns are limited to a relatively narrow but abstract dimension of implementation, stretching though from relatively detailed to independent in a specific domain[96].

---

[94] From the very beginning, let us actually mention one common aspect of patterns on one hand and SWA styles on the other. In both cases, the approach taken by the SW community was only to look back and categorize whatever was done in the area, without necessarily coming up with something new.

[95] From the Go4 book, the Facade, Observer or Strategy; nevertheless most patterns in the same book fail to address architectural issues.

[96] Due to their three levels of abstraction [14]: architectural (frameworks), design and implementation.



The development trace of a large-scale system is also figured, starting from the problem domain, going through architecture, eventually framework implementation and ending with instantiating the application using available kits.

An architectural style provides a language and a framework for describing families of well-defined SW architectures. The role of a style is to provide a way to express both architectural instances and patterns of common architectural design. Therefore the concepts of a particular architectural style could be compared to the ones underlying a specific methodology (as we attempted in CODE). An architectural style is better thought of as a language for building patterns rather than an instance of a pattern itself[97]. On the other hand, a pattern language is a system of patterns organized in a structure that guide the application of the patterns, and from this perspective SWA styles could be

[97] Having already stated that frameworks are the typical example of patterns on the architectural level, SWA styles could very well offer models for building frameworks. With a certain set of applications in mind, a framework could be built fitting that particular set of applications but following a specific architectural style.

looked at as being close to a pattern language. Although a SW architectural philosophy is a very hard to explain concept, a pattern language can embody this philosophy in a form that could be written, discussed and evaluated.

Both styles and patterns try to capture key ideas behind successful designs, each on different abstraction levels. These ideas are just instances of a more complex concept defined by Alexander as 'quality-without-a-name' and which is an intimate blend of customer requirements and deep convictions about how a design should be made. W. Cunningham makes a comparison between SWA and building architecture where he places above the SW architectural style an 'architectural philosophy'. Just as applied to buildings, the style acts as a filter and does the mapping between the 'architectural philosophy', which is somehow biased, and the final design decisions. It is not an architectural style that leads great building architects to successful constructions but their internal philosophy. This type of creativity remains a basic ingredient for good engineering, as we have pointed out already. We don't know yet about many of these aspects since SW architecture has been rarely discussed, described or written about; the emerging of the SWA discipline first, and later the focus on patterns, attempt to fill this need.

Concluding remarks

The goal of this chapter was to look at the SWA styles (introduced earlier in the thesis) from a coordination perspective. Our review of the SWA literature turned out that most of the work in this area has focused on analyzing and categorizing popular architectural organizations into SWA styles. So far, architectural styles (as well as patterns) have been concerned with how to leverage past experience in order to produce better designs[98]. Following this approach we could split existing SWA taxonomies between call and flow architectures; none of these necessarily fits the interactive (event-driven) character of today's computer systems. In terms of SWA, we think that over the past few years the interest shifted dramatically from what we called architectures for static environments (where the call/flow models could be applied), to the ones for dynamic environments (leading to coordination oriented organizations). This shift is a direct consequence of higher and higher interaction demands put on SW systems today, which calls for more emphasis on coordination in the architectural design. CBD itself is a bold step in this direction. Consequently we introduced in this chapter a behavior-based taxonomy of the SWA styles identified so far (see Appendix 2), as a step towards

[98] Nevertheless even rather informal architectural or design characterizations are able to convey a lot about a system's structure and the underlying computational model.

integrating SWA with today's major developments in computing technology (that is, the shift from algorithmic to interactive/behavioral models). With the tremendous increase in demands and complexity put on today's SW systems, programming in the large becomes highly interactive and cannot be expressed anymore by (or reduced to) the same abstractions used with programming in the small.

A lot more discussions and arguments are still being carried about SWA styles, starting from taxonomy-related issues and ending with the utility of styles as opposed to patterns. For example some authors are tempted to see OO as a style on its own[99], given that both OO and architectural styles are able to capture a broad range of design families. We tend to disagree with this point of view, seeing OO actually as a rather versatile way of hiding information, with its own principles, languages and tools. It is true that OO is an important step towards a behavioral approach to SW by bringing interaction at the forefront (compared to the procedural one), nevertheless virtually any of the popular SWA styles could be implemented with either procedural or OO techniques.

SWA concepts (captured in 'styles') allow an architect to describe multiple and versatile interfaces to a component and to describe and encapsulate protocols of component interaction, otherwise difficult to express using conventional design techniques (OO for instance). On the other hand, situations that require cooperation of multiple design entities (like objects) or specifying and packaging related collections of those entities (for future reuse) are captured very well by patterns. We have therefore discussed in detail how styles and patterns are related.

[99] **Where all components are objects and all connections are simple associations of objects.**

SW becomes more and more of a commodity these days. From simple to complex, the SW component is the one that is predominant in most of today's systems. Starting by being qualified as a 'science' in its early days (almost irrespective to its complexity) SW is now looked at as a product. The overall design and specification of software systems emerged as a central concern, with the increasing size and complexity of today's systems. These concerns are now being addressed at a higher abstraction level including the gross organization of the system, assignment of functionality to constituent elements and selection of design alternatives. In this context, Software Architecture was of particular interest to us, in an attempt to characterize SW systems at a higher level of abstraction and make the 'complete system' intellectually tractable. But we also believe that the ultimate goal of SWA is to promote quality attributes of the design product (like usability, reliability, maintainability, efficiency, etc) on abstraction levels higher than the design phase. We maintain that next to the need to handle complexity, the need for architecture mostly comes from satisfying non-functional product requirements on a large scale. Therefore, we consider SWA as a mandatory step in the SW development process, in particular for large SW systems. SWA is believed to have quite a few positive impacts on system development [30][50], specifically in terms of

- *understanding* of the system: SWA describes large systems at a level of abstraction that makes them easy to comprehend and also highlights the principles behind the most important design choices
- *analysis*: SWA provides opportunities for analysis such as high level system consistency, conformance to a particular architectural style or domain specific architecture
- *reuse*: SWA supports the reuse of large components as well as control structures into which components can be integrated. The current interest in frameworks and architectural patterns are a direct consequence of this.

- *evolution*: SWA highlights the dimensions a system is expected to evolve along, by exposing the 'load-bearing walls', what could be done and what couldn't. This way, maintainers can make an informed call on their design decisions and ramifications of their changes. Architecture can also flesh out functionality from the way components are connected and interact, allowing for a stepwise evolution from prototype to a fully developed system and further on to enhancement, modifications or reuse.
- *management*: SWA proves to be a useful tool to project management in industrial SW development. It serves to specify initial functional requirements, gross functional partition – and related to it, work division among developers – performance and capabilities, as well as dimensions of anticipated growth. There is ongoing work on whether to map the SWA on development teams or the other way around.

Besides advocating the need for SWA, we want to emphasize once again the value of making coordination a center point when it comes to SW development. Today's SW development practice confirms the fact that, as we get closer to the implementation stage in the design process, current design and programming tools tend to focus increasingly on components, leaving the coordination and communication aspects out of the picture. Support for complex module interactions is buried in the semantics of programming languages or operating systems or, even worse, is fragmented in order to be embedded into the computational modules. Our review of the SWA literature turned out that most of the work in this area has focused on analyzing and categorizing popular architectural organizations into SWA styles. Following this approach, such classification could further split between call and flow architectures, none of which necessarily matching the interactive (event-driven) character of today's computer systems. In contrast, we think that coordination deserves at least as much attention as the execution part starting as early as the architectural level. If decisions regarding the computational aspects could be deferred to the design level, it is the coordination structure that needs to be decided along with the architecture. In order to facilitate this, components should be separated from their interdependencies from the very beginning. If in the regular SW architecture components aggregate the core functionality, interdependencies amongst components should be considered a concept *orthogonal*[100] to the problem domain. Once this separation in place, we believe that a coordination-

[100] In order to achieve such a separation, it is essential that no interconnection or coordination assumptions be built into the computational modules.

oriented architecture is the only one to fit the complex interactive[101] character of modern computer-based systems.

CODE[102], the integrated development environment we introduce in this thesis genuinely achieves a lot of these desiderates. In contrast to other present-day development methods (which are specific to particular classes of applications and not necessarily consistent from one design level to another in terms of methods and tools), CODE is believed to make a difference. It extends and integrates the phases of the development process (from requirement acquisition to implementation and further to product retirement) with matching methodologies, system architecture and modeling tools. The CODE methodology uses a behavioral view that characterizes the system as being event-driven; the system executes 'work' as a response to events that drive it. Furthermore, the work associated with a system response is partitioned into relatively independent units of work called 'activities'. In other words, each response is viewed as a set of activities which have to be executed in a given precedence relation (represented by a precedence graph) to provide the required outcome to an event. As a consequence, in specifying the event responses, *execution* (what has to be done, reflected in the units of work) is separated from *coordination* (when to do it). The resulting system architecture is not static; through its hierarchy and selection of critical resources it strongly suggests how the architecture has been achieved. Furthermore, both methodology and architecture view the computer system to be implemented as a set of *assignable resources* (software objects/modules, hardware components and interfaces), used in an orderly manner to execute the activities (the units of work) that configure the responses of the system. This approach in particular is the one that leads to maintaining a strict separation of coordination from execution throughout the development process.

Many of the CODE concepts have been illustrated through a case study in which we set out to architect a hospital information system. Without going into design or implementation details, the intent of this example was to illustrate that CODE concepts could be applied in a systematic way. It was shown that one could start with a relatively undefined idea of how the system would be implemented (which is usually the case with a complex system) and then successively refine it with users, which bring in different expertise. The case

---

[101] From an architectural standpoint and given their interactive character, present-day computer systems look more along the lines of urbanism rather than civil architecture: one would have to provide not only for complex interactions between the parts, but also for controlled system evolution.

[102] CODE (Coordination Oriented Development Environment) is an extension of the work on Coordination Based Design and its associated methodology [2].

study hopefully indicated that if different people come up with the same critical resources within the suggested architecture, the overall module partitioning turns out alike. We consider the case study as being the empirical part of the thesis. It suggests that CODE allows for multidisciplinary design, where one could closely work with the user before the implementation stage and at the same time fit a well-defined development process. CODE is in fact a user-controlled design environment; the methodology attempts to understand user needs beyond the rough functionality of the system. It seeks to translate the user understanding of the system into design, ask the user why they want specific things, and eventually suggest new features. This way CODE fits very well the present-day concept in Computer Engineering where the computer specialist works closely with the user throughout the development process. CODE involves the user at all the development stages except coding; this is in contrast to the usual development process where the user is only involved in the requirement acquisition phase. Being such a user-centered development process is only one of the reasons we believe CODE is "better".

The CODE architecture fulfills many of the desiderates identified when we discussed the need for SWA, specifically:

- an architecture that provides the user with an easy to understand cognitive/mental model of the system, thus facilitating immediate user feedback. These models also provide the required communication (and informal coordination) amongst developer groups, both in time and space. So far, the common way to communicate important system design ideas was through block diagrams, flow charts, etc. CODE does this in two ways:
  1. various coordinator/executor *diagrams* that are presented at different levels of detail and could be easily annotated.
  2. a set of *event responses* (in the form of PAL expressions) associated to each coordinator/executor diagram.

  This leads to the design of better products by making available the proper information when and where it is needed.
- an architecture that is matched to the behavioral specification needed in most modern systems. This means that the system specification is made in terms of event responses. Each event response represents the work to be done in terms of 'units of work' executed in a given sequence. As such, coordination is separated from execution via coordinators and executors, with the following major consequences:
  1. *information hiding*, where the 'what' is encapsulated in coordinators and the 'how' in executors.

2. a *structured view* of the system in terms of assignable resources, which are associated to event responses. This system perspective helps reuse and fits well with component-based design.
3. good *traceability* of requirements onto the implemented product. Traceability is provided by the fact that functional requirements and constraints associated to a particular event response are taken into account at all development stages.

- an architecture that partitions the system in a top-down fashion, thus providing for ease of refinement and overall system perspective. Such an approach allows for partial initial specification followed by stepwise refinements (new responses, interactions, etc.) as one uncovers them during design. One could easily zoom-in/zoom-out by refining or hiding the details of the executor modules in the various coordinator/executor diagrams. Both hierarchical decomposition and separation of concerns also facilitate system maintenance (replacement or addition of functionality), allow for simple reuse as well as for component-based design.
- an architecture that is not stand-alone but part of a complete development environment, capable of investigation and decision making. Our architecture comes not only with a matching design environment (PAL-based tools) but also with a well defined development methodology and process. The following benefits could be mentioned with respect to the architecture and its associated development environment:
  1. CODE fits the reality of group design and allows for investigation at all development stages; it considers not only what but also why particular features are needed. CODE uses the same models and tools throughout the development process, which provides a common language for users, architects, developers, implementers and maintainers.
  2. CODE considers the entire product lifecycle, from requirements specification to retirement. It provides for easier development and maintenance through instrumentation at the coordinator level, as well as incremental delivery and retirement through the response-oriented structure. Such attributes are needed in the case of large-scale complex systems, which have to go through a number of upgrades during their lifecycle.
- an architecture that considers complete computer-based systems – HW and SW – to which one can apply system constraints. The traditional approach to SW development considers the HW platform as a constraint on the SW architecture. In contrast, CODE evolves SW architecture towards a computer-based architecture that considers HW and SW together, as resources assigned to a particular event response.

Both the development and refinement of CODE are still underway. More work is needed in the areas of system implementation, delivery and retirement, while the architectural and design phases are likely to still be refined. While more analysis is necessary and definitely useful, it is not enough. Practice in using the models is at least as important and ultimately is the only way to verify the strengths and weaknesses of a concept. Therefore the entire CODE concept needs to be proven within the framework of a large SW project spanning the entire lifecycle of a product.

The word *'engineering'* originates from the Latin *'gignere'* – to create, to produce – and its associated *'ingenium'* – talent, natural capacity, invention. We have mentioned creativity throughout the thesis, as one of the important ingredients in present day engineering. Nevertheless, creativity has been referred to not in the artistic sense, but as the capacity to make products that are pragmatic and fulfill specific needs. Without the hope of a silver bullet, we believe CODE is a bold step in this direction.

Appendix 1

SWA STYLES SURVEY

Here is the original layout of the literature survey made by us with respect to SWA styles. We abstracted many of the definitions below, in absence of any from their original promoters (but hopefully close to their views).

	Perry and Wolf	Garlan and Shaw	Shaw	Katzman	Go5
Multi Phase	(early analog of Batch-Sequential)				
Sequential	(early analog of Pipes and Filters)				
Parallel process, Shared Data	(early analog of Blackboard)				
Pipes and Filters		Connected components reading and producing streams of data	Mapping of data streams (pipes) performed by filters	Incremental transformation of data (stream to stream)	Systems that process streams of data
Batch sequential		Degenerated case of 'pipes and filters'.		Independent programs performing data transformation in steps.	
Object Oriented		Particular case of data abstraction	Instance of data abstraction	Encapsulation + Inheritance + polimorphism	
Data Abstraction		Data and its operations encapsulated	Local state storage; de-centralized		

		in an abstract data type	components interact by procedure call		
Implicit invocation		Components invoking registered operations via broadcast	Independent reactive processes making invocations without knowing the recipients		
Event Systems		Used interchangeably with implicit invocation	Included under implicit invocation	Objects or processes registering procedures with events non-deterministically announced.	
Layered		Hierarchical organization where a layer acts as a client to the one below.	Collections of procedures in a hierarchy of opaque layers.	Layer: hiding the ones below and serving the ones above.	Groups of subtasks, each group on a particular level of abstraction.
Repository		Central data structure operated by independent components.	Structured, centralized data directly accessed by more computational processes.		
Interpreters		Virtual machine produced in SW.	Virtual state machine.	Execution engine in SW.	
Distributed Processes		(mentioned as other category)			
Main Program and Subroutines		(mentioned as other category)	Single thread of control and definitions hierarchy between	Hierarchical decomposition based on definition/use relationship	

			procedures		
DSSA		(mentioned as other category)			
State Transition		(mentioned as other category)			
Process control		(mentioned as other category)			
Database			Included under Repository	Large central data store controlled by incoming transactions	
Blackboard			Included under Repository	Central shared data representation controlled by the state of data.	Specialized subsystems assembling knowledge to build together an approximate solution.
Communicating Processes				Independent processes communicating via message passing	
Rule-based Systems				Specialization of interpreters.	
Broker					Systems with de-coupled components interacting by remote service invocation
Model View Controller					Division of interactive applications into functionality and data, display, user input.

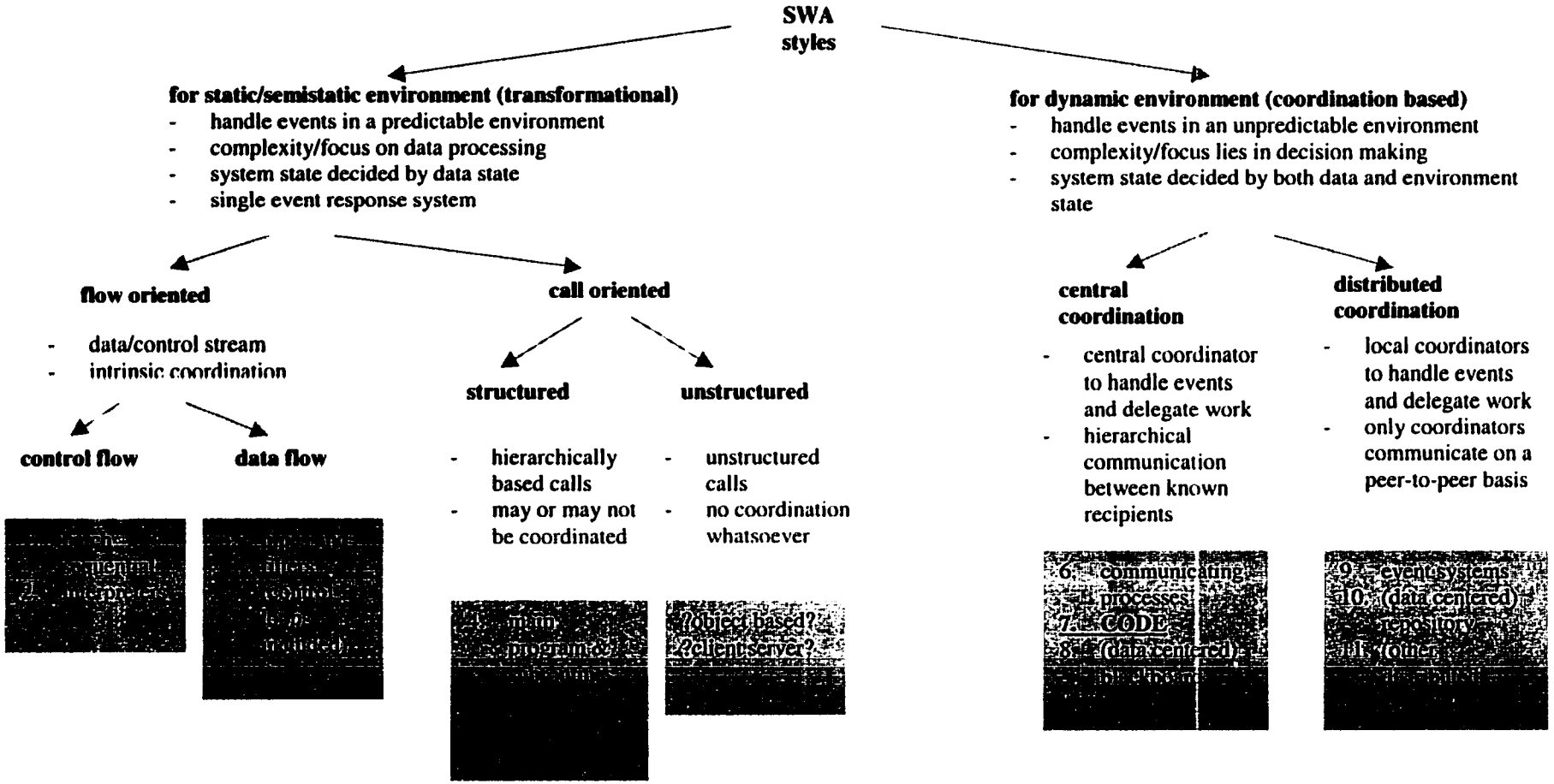
Presentation Abstraction Control					Hierarchy of cooperating agents, each level responsible of a specific aspect in an interactive application.
Microkernel					Separation of a core of minimal functionality from extended functionality and customizations.
Reflection					Abstraction of selected system properties from the application logic.
	Only examples, no attempt of a taxonomy.	First attempt of a style classification	Further refinement of Garlan & Shaw	Significant contribution based on the Carneggie-Mellon school.	Taxonomy oriented very much towards DSSA.

From this survey seems that:

1. Rather than developing a taxonomy, Perry&Wolf were more interested in the conceptual aspects involving SWA and architectural styles. They only use some examples to illustrate their ideas.
2. Garlan&Shaw; Shaw; Katzman were interested in taxonomies that fit most applications known so far.
3. Go5 are also interested in providing a taxonomy of styles, this time very pattern-minded, extracted from very specialized application areas (GUIs, OS, etc).

Appendix 2

SWA STYLES DIAGRAM



- *architected to only handle events anticipated at design time*
- each application considered alone; applications often self-contained, do not interact much
- if there is a number of applications, they do not interact except when competing for the same resources
- all of what has to be done, is done by well defined entities of the same application

- *architected to handle events out of the initial design (exceptions)*
- client-server: any application assumes there are other (specialized) units that can do work for it; these units do work for other applications as well
- often work with imprecise computations (Blackboard)

Appendix 3

COORDINATION BASED DESIGN

The Multiactivity Paradigm: An Approach for the Design of Embedded Systems by Application Specialists[1]

Moshe Krieger[2]

Department of Electrical Engineering
University of Ottawa
Ottawa, Ontario, Canada, K1N 6N5

Abstract — The proliferation of cheap high-performance components has lead to the introduction of many new computer-based embedded systems. However, the development of these systems is hindered by: 1) the lack of experienced computer engineers, 2) the cost of maintaining development teams that include both application and computer engineers, and 3) the design and implementation errors that result from the inability of computer engineers to understand the complicated embedded applications. For these reasons, a new design environment that removes the computer engineer from the design cycle is proposed here. This design environment is based on the *multiactivity paradigm* that separates activity executions from their coordination by viewing a system as a set of assignable executors controlled by a coordinator. This paper introduces the *Process Activity Language, PAL*, a high level executable specification language that matches the multiactivity view and allows the specification of event-driven embedded systems in a structured way. We have implemented a number of PAL-based tools that aid the designer to check for specification errors and to configure the embedded system to meet various constraints. These tools demonstrate the feasibility of a design environment that enables an application specialist to design a complete computer control system.

Key Words — Embedded systems, realtime, multiprocessing, multiactivity paradigm, system specification, system design, prototype based design.

1. Introduction

Rapid technological advances have opened up many new application areas for microprocessor-based systems. One such area is that of embedded computer systems, where the computer is embedded in another system whose operation it monitors and controls in an ongoing fashion. Most of the early embedded systems were relatively simple; they included a basic executive that allowed only for sequential response to the different events from the environment. In such elementary sequential event-response systems, the response to an event had to be completed before another could be serviced, so these executives were limited in their ability to control concurrent realtime responses. Today, the performance requirements of realtime or embedded systems of any complexity require multiple processor implementations [Gen88]. Providing additional processing capability is only one reason that multiple processors are needed in embedded systems. More importantly, because of the asynchronous and parallel nature of event occurrences in the environment, many embedded systems require real concurrency as opposed to apparent concurrency. Additionally, the functional requirements of many embedded systems are such that they require the use of different specialized processors rather than the use of a single high-performance system. Also, the reliability requirements of many of these systems can only be achieved through the use of a reconfigurable system. Finally, multi-microprocessor systems, if properly designed, are attractive in that they are modular systems making them easier to implement, upgrade and maintain [FK83a].

To meet performance requirements, most present day embedded systems of any complexity, have to be implemented as multitasking systems. A major problem in the design of such systems is to properly translate the requirement specification, prepared by the application specialists, into a design specification; that is, a set of cooperating tasks that matches the computing platform (the multiprocessor hardware and the operating system) on which it will execute. For best results it is important to match the application with a specific computing platform. This step is generally beyond the capabilities of most application specialists as it requires a good understanding of operating systems and some knowledge of multiprocessor architectures. On the other hand, the computer specialists do this partitioning (the design specification) based on computer related artifacts (language, operating systems elements, hardware, etc.) rather than upon application characteristics [Law90]. One way to resolve this is to have close cooperation among the two types of specialists and to provide them with the proper specification and design tools [Deu88]. This is not only very hard to achieve in real life, but also because of the high costs involved, few companies can afford the luxury of maintaining computer specialists, distinct from application specialists.

Considering the complexity and the diversity of present day applications, one cannot expect computer engineers to have the problem-relevant insight needed for properly partitioning the requirement specification into a set of cooperating tasks. On the other hand, the multiplicity of computing platforms that are in use today, their complexities, and the differences between platforms make it impossible for application engineers to be specialized in more than one or at most two platforms. As a consequence, there are many errors introduced in the early design stages of embedded systems (regardless of who designs it). These increase the manpower, hardware and time required to make these systems operational. Often, after spending time to find out why the system is not performing properly, designers end up using additional hardware to improve system responsiveness and performance. The result is inefficient, poorly understood systems, running on overspecified hardware. Such systems usually go over budget (time and money) during implementation, and prove difficult if not impossible to update and maintain.

To reduce or to avoid design errors it is highly desirable that during the whole design cycle there should be a common view of the system [Law90]. This view should be problem specific [SGME92, Tak80] and shared by the entire design team. It is preferable that the design should be under the control of the application specialists since they are best able to take into account all possible interactions and responses. The application engineer is also needed to check if the final implementation satisfies all the constraints and fits the needs of the user. An added advantage of having systems designed by the application engineer was noted by Gentleman [Gen88], when he stated that many embedded systems use highly specialized peripherals which have to be handled directly by the application programs because the operating systems generally do not provide the needed drivers.

Today a number of well known specification techniques and design tools are available [Boo91, Har90, HP88, War89]. Most are object oriented, requiring the system to be specified in terms of cooperating tasks. This is quite artificial to many application engineers, who are more control oriented and are used to specifying systems in terms of responses rather than a particular structure (which is what the object oriented specification emphasizes). In embedded computer systems the critical requirements relate to the dynamic behavior, which is abstracted away in an object oriented scheme. The contention of this paper is that the present day needs of highly specialized embedded systems necessitate the development of a design methodology and an associated set of tools that allow application specialists to design and supervise the implementation of these systems without the help of computer engineers. This is possible because many application engineers are used to CAD (computer aided design) tools and because of the availability of high-performance off-the-shelf standard products (computer boards,...) that no longer need be used at the limit of their capabilities.

This paper introduces the multiactivity design methodology and an associated set of tools which are intended for computer literate application engineers to design computer-based controllers. In this scheme, system specification corresponds to formulating system responses as a set of activities (actions or operations) that have to be executed in a given precedence relation. In the proposed design methodology, these activities are the schedulable entities that have to be executed, therefore avoiding the need to partition the system responses into cooperating tasks. System implementation is accomplished by allocating the

[1] Paper submitted to "IEEE Transactions on Systems, Man and Cybernetics".

[2] Part of this research was done at the Institute for Information Technology, National Research Council of Canada, where the author is a guest worker.

activities to the various processors using a system prototype , and by writing the code of the activities. The multiactivity design methodology restricts the solution space, and imposes a particular architecture and control structure, trading some system efficiency for ease of design. Furthermore, the multiactivity view of systems fits the changing role of the designer from that of component designer to system integrator, by supporting system implementation from off-the-shelf components. In the subsequent sections we introduce an overview of embedded systems and present the multiactivity view which forms the basis of our design methodology.

2. Embedded Systems

In the general form of an embedded application, as indicated in Figure 1, there are two distinct entities: the *environment* — generally a complex dynamic system (aspects of which may or may not be alterable by the designer) including a multiplicity of physical devices that have to be controlled; and the *embedded system* — a computer based control system that has to coordinate the operation of the physical components in a prescribed way. In embedded applications, the computer-based system and its environment form a synergistic pair, with the overall behaviour being generally dictated by the environment. The computer has to influence the environment but rarely has complete control over it. Embedded systems are event driven or reactive systems where the computer reacts to events that it monitors via a set of sensors and responds by providing the necessary controls, at the proper times, via actuators.

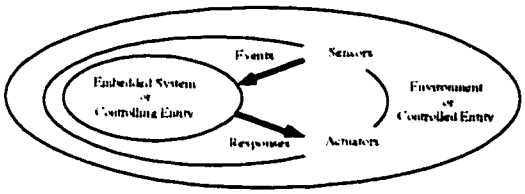


Figure 1. An embedded application.

There are a large variety of embedded systems, from simple industrial controllers to highly complex flexible manufacturing systems, but most of them exhibit to some degree or another the following general characteristics [Gen88, Sta88, KDK89]:

- Interaction with the environment is beyond direct programmed control, it is via sensors and actuators that work to limited accuracy and are prone to fail.
- The environment can be quite complex; that is, it may not be fully modelled nor instrumented at will. In many systems the environment can only be influenced but not controlled completely.
- Generally, the environment is the driver and the computer must be able to handle simultaneous events and coordinate multiple concurrent responses. Event responses can be *sporadic*, initiated by asynchronous events in the environment and/or *periodic*, i.e. executed cyclically.
- In these systems, time becomes a resource that must be managed; things have to happen at the right time — *the validity of a response depends both on the correctness of the computations and the time the results are produced.* There are two types of timing restrictions on the responses: *precedence constraints*, determined by the precedence requirements of the constituent actions; and *timing constraints* for detecting (or accepting) events and/or starting or completing responses.
- Timing constraints can be *hard*, in which case a violation represents an error, or *soft*, in which case some delays may be accepted or some events may be missed. In the literature, timing constraints are most often expressed as *deadlines*, but in many systems *jitter* i.e. deviation from intended time, is the critical requirement.
- These systems generally need stringent safety requirements; system failures may lead to catastrophic results such as loss of life or system destruction. Failures may be caused by software errors, component malfunctions and/or by environmental disturbances.
- Embedded systems generally have very long lifetimes requiring multiple upgrades. These upgrades may not only involve control algorithms but also the interfaces and even the computer system.

Embedded systems can be specified at various levels of abstraction or description. At the highest level, systems are defined as being made up of processes, corresponding to event responses. As shown in Figure 2, these processes can be partitioned into subprocesses, which in turn can be repeatedly partitioned until one reaches the smallest response component of interest to the system designer. The entities at this level are called *activities*, where an activity is defined as *a meaningful sequence of operations (or actions) that can be executed, once started, from beginning to end without having to wait for anything (information from other activities, resources, etc.).*

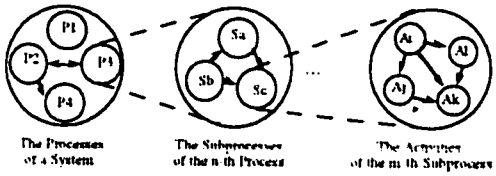


Figure 2. Levels of abstractions in embedded systems.

In the above context, an *activity* is *an independent unit of work* and a *process*[3] is *a set of activities which must be executed in a given precedence relation in order to provide the required response to an external event*. The sequence in which the activities have to be executed is determined by the data dependencies that exist between the activities within a process. There may also be dependencies between the activities of different processes. In fact, seldom can one define the processes so that there is no interaction between them. Interprocess interactions can occur either directly between processes or indirectly via the environment. In any system, the exact dependencies between its constituent activities define the amount of concurrency (parallelism) that is possible within the system.

Quite often embedded systems cannot be specified purely in terms of event responses (fixed control algorithms). One has to monitor the system (both the environment and controller) and determine different *operating modes* and also various *safety procedures* in case of exceptions. Furthermore, to provide for the different operational characteristics, beyond the primary sensors that initiate and control event responses one may need additional sensors, such as *reflex* or *feedback sensors* to determine if a given action was properly executed, *safety sensors* to detect critical conditions and *state sensors* to determine the different operating modes.

From the above overview of embedded systems it can be seen that they may differ significantly from conventional data processing computer systems: they have to interact directly with their environment, the environment can be quite complex and imprecise, time has to be handled explicitly, etc. It is not surprising, therefore, that a different design methodology is needed for many of these systems. In the next section we introduce the multiactivity point of view that is well suited to event-driven embedded systems that have soft time constraints and that require intricate interactions with their environment.

3. Multiactivity View of Embedded Systems

The *multiactivity* paradigm is based on two observations. First, that all *work* has two components: the various *activities* that have to be executed and the *coordination* of these activities. Second, if activity execution is separated from activity coordination in an implementation, the resulting system is more flexible and easier to design.

In terms of embedded systems, the multiactivity view states that system processes are considered at two distinct levels: the *coordination level* — which specifies the sequence of activities that has to be initiated in response to an event, and the *execution level* — which specifies the actions defined by and carried out in an activity. In this sense, the activities represent the "what has to be done" component of the response, while their coordination corresponds to the "when to do" component. Note that there is also a need for interaction between processes, this is done at the boundaries between activities. Functionally, this separation

[3] In our terminology a process may have multiple threads of control as it generally has parts that can execute concurrently. Therefore we consider processes and tasks to be different entities. A *task* is defined to be *a set of activities (generally related) with a single thread of control.*

between the coordination of activities and their execution means that *multiactivity systems* can be realized using one or more *coordinators* and a number of *activity executors*, as shown schematically in Figure 3. The coordinator accepts events, interprets them and initiates the execution of the appropriate sets of activities, at the proper times, by the activity executors. Each coordinator views everything beneath it as activity executors. In this hierarchy each sub-coordinator has decreasing (more localized) knowledge of the total response. Note that the hierarchical organisation of the coordinators is used to simplify their operation. It is possible to design multiactivity systems with only a single coordinator, but then the implementation of the coordinator becomes more complex, and system bottlenecks may occur.

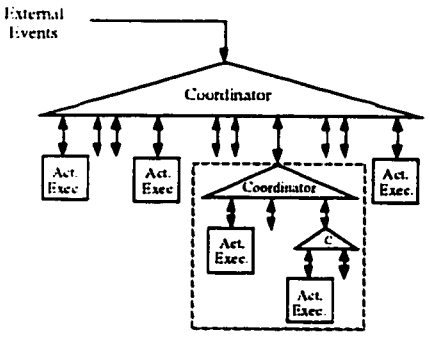


Figure 3. Functional organisation of multiactivity systems

In general, each activity executor can execute a number of activities. The actual assignment of activities to executors is based on a number of factors, such as: specific requirements of the activities, potential concurrencies between activities, and load balancing. As discussed later, the same activity may generally be assigned to more than one executor to provide responsiveness and reliability.

3.1 The Multiactivity Coordinator — The System Software

The *multiactivity coordinator* is responsible for the proper sequencing and scheduling of the activities of the various responses. As indicated in Figure 4, such a coordinator can be implemented as a simple three layered software structure. The top layer of the coordinator consists of the *system manager* which interacts directly with the environment and has a general knowledge of the system. The system manager includes an *event monitor* which could be implemented as an interrupt handler. Upon detecting an external event, the system manager is responsible for initiating (or continuing) the required process. Associated with the system manager is a *time, interaction and error handler* which supervises the various timeouts and the interactions between processes, and initiates the proper responses to the various error (failure) conditions.

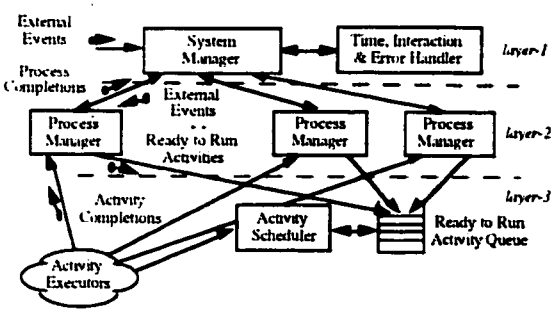


Figure 4. The structure of the multiactivity coordinator

The middle layer of the coordinator consists of *process managers*, one for each active process in the system. It is the process manager that contains the precedence relations of the activities of a given process, and that determines, at each instant, the ready-to-run activities of that process.

The actual scheduling of activity execution is done by the *activity scheduler* which forms the bottom layer of the coordinator. The scheduler assigns activities to the executors that can perform them,based on a predetermined scheduling scheme and the availability of executors capable of performing a specific activity. The scheduling scheme used by the scheduler can be based on activity priorities such as first-come first-served, fixed (multilevel) priorities, shortest execution time, etc., or based on process priorities such as earliest due date, minimal slack time, etc.

The main characteristic of the multiactivity coordinator is that all the relevant system information to manage event responses is contained within it. The activities do not need to include any information related to the coordination of the process. The coordinator also manages the information transfers needed for activity execution and channels all the required interactions between processes and the environment. The multiactivity coordinator may appear complex, but it has been shown that a *general purpose multiactivity executive* can be implemented as a relatively simple graph traversal program [Sha91] that uses linked lists to characterize the specific applications.

3.2 A Hardware Platform

Multiactivity systems are well suited for centrally coordinated multi-microcomputer implementation because of the separation between activity coordination and activity execution. The general organization of a centrally coordinated multiactivity system is given in Figure 5. In such a system there is a central coordinator which monitors the environment and runs the multiactivity executive, and a set of processing units which execute the individual activities. The system may also have some global resources such as a memory used for interprocess communications, and special purpose I/O.

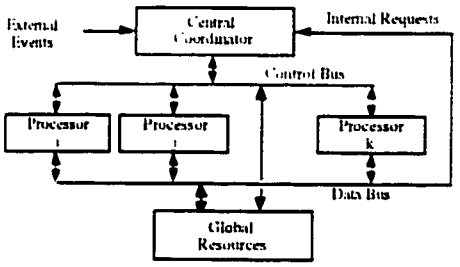


Figure 5. Centrally coordinated system architecture.

The processing elements can be either homogeneous or heterogeneous, with some processors customized for the particular set of activities that they will execute. Often system requirements dictate that only specific processors can execute given activities (for example processors located in the different parts of a robot arm). Each processor has its private program memory, data memory and I/O. Executable code for each of the various activities is permanently stored in the local program memory of the individual processors. Thus when an activity is to be executed, it needs only to be awakened rather than to be downloaded from a central memory. Any data required by the activity is transmitted to the processor on which it will execute either by value or via pointer to the location of the data (in shared memory systems). In order to restrict the size of the program memories and the I/O interface, not all the activity codes are duplicated in each processor. However, to increase system reliability and to obtain the required level of system performance, each activity code is stored in two or more processors, whenever it is possible. To simplify the wake-up call, each activity is assigned the same identifier in each processor capable of executing it. A typical processor organization and a profile of its program memory is indicated in Figure 6.

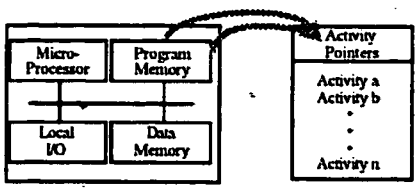


Figure 6. A typical processor organization and a profile of its program memory.

3.4 Characteristics of Central Coordination

When properly implemented, central coordination can provide significant benefits in the areas of system performance and error handling. Because of the centralized nature of the multiactivity system, it is simple to implement various scheduling schemes; the coordinator is aware of the current timing needs of each active process, the current list of ready to run activities and the status of each processor, so it can meet special deadlines or achieve better load balancing.

Because of interaction with the physical world, the requirement for reliable and safe operation is probably the most important feature of embedded systems. In multiactivity systems, the separation of activity coordination from activity execution, as well as the atomic execution of activities, allow for easy fault detection and fault containment. Response coordination can only be affected by a failure of the central coordinator, where a high degree of reliability can be achieved by running two processors in a hot standby configuration. With respect to faults in an activity executor, a catastrophic failure (i.e. a processor does not respond) can be handled by incorporating a timeout mechanism in the coordinator which, when activated, declares the processor faulty, removes it from the list of available processors, and restarts the affected activity on a new processor. Highly critical activities can be run simultaneously on two processors in a master/slave mode to detect discrepancies; in case of error, the activity can be rerun on a third processor to determine the faulty unit. Alternately, one can use triple modular redundancy (TMR) and use the majority output.

Generally the sensors and the actuators are the most vulnerable elements of an embedded system. To guard against sensor errors, the *event monitor* of the coordinator can include a comparator that checks the present sensor input against previous values or against some maximal limits. To guard against actuator errors, whenever applicable an extra sensor or a watchdog activity can be used to check if the activity was executed correctly.

In multiactivity systems, due to the centralized control, it is also quite simple to add *safety features* to guard against actions that may produce catastrophic results. Critical commands can be checked against predetermined limits before being issued. Sensors can be placed at strategic points in the environment and watchdog processes used to abort given responses or to initiate complete system shutdown.

Centrally coordinated systems are also highly flexible in that additional activity executors can be added to overcome bottlenecks and to provide extra performance for additional functionality as requirements change. This flexibility is especially important in embedded systems because of their very long lifetime and the possibility of step-wise system deployment.

Multiactivity systems may exhibit some of the drawbacks generally associated with all centralized systems: controller bottleneck, communications bottleneck and controller reliability. Controller reliability, as stated above, can be improved by using two processors in a standby configuration. Because of the simplicity of the outlined executive [San90, Sha91], the control bottleneck becomes a problem only if the number of processes and activities becomes very large. A good partitioning of the system through activities with large granularities and low data dependencies can significantly reduce the communications bottleneck.

Note that the architecture described above assumes that the number of processes and activities of a given application is not too large, and that the granularity of the activities is such that they have significant processing requirements. If this is not the case, there are two alternative architectures. If the application includes a number of very complex activities, these activities can themselves be implemented as separate multiactivity systems. Alternatively, if the application consists of a large number of low granularity activities, a better choice is to implement the central coordinator as a multiprocessor system that has a number of dedicated process managers, some of which can also execute low granularity activities. This centralized architecture is not applicable to physically distributed systems, unless they can be partitioned into a number of relatively independent centralized systems.

4. Process Activity Language, PAL

The development of any complex system involves a number of distinct steps, requirement analysis, system specification, design, implementation, testing and the development of proper maintenance procedures. Ultimately, the work done at each step must be a reflection of the system specification and it is frequently necessary to refer back to this specification to resolve problems arising at later stages. Because realtime systems can be complex, it is important to develop suitable tools for specifying and modeling the system at various levels of abstraction. Even a brief survey of the literature reveals a variety of ways of modeling discrete-event systems, that is, of representing their activities and precedence constraints. These methods range from the relatively informal graphical notations used in *data flow graphs* [DeM78] to the very rigorous notations used in *Petri nets* [Pet77]. There are also a large number of functional representations from informal system specification languages to highly complex notations based upon *communicating sequential processes* [Hoa85]. However, many of these tools are limited in their scope. Some are not easily applicable to all the system development steps, some are highly application-specific and some require expert computer knowledge.

This section describes PAL, a general purpose high-level language for the specification and design of embedded systems [KLH88]. PAL allows the designer to specify the coordination of event responses independently from the implementation details. For each system response, the precedence relations among the activities and the interaction with other responses are described via well-formed expressions using constructs similar to those found in structured programming languages.

4.1 Elements of PAL

In developing PAL, we have attempted to strike a balance between the need for formalism to allow for verification, and the need for clarity. To satisfy formalism, we have chosen well defined primitives and control constructs, similar to those found in structured programming languages. To satisfy clarity, we have limited the number and complexity of the elements of the language and based them on intuitively understood concepts.

A system in PAL is specified as a set of processes $\{P_1, P_2, \ldots, P_n\}$. A process P_i has a well defined starting event S_i, a response X_i, and an end E. The general form of a PAL process is represented as, $P_i := S_i \bullet X_i \bullet E$.

A given starting event may awaken a number of processes[4]. Depending upon the system, if another starting event S_i occurs while the given response is active, the event may be ignored, queued, or another instantiation of the process may be started. The body of process P_i, the response X_i, is a PAL expression composed of PAL language primitives whose execution ordering is defined by PAL control constructs. There may be an *abort* or *shut-down procedure* associated with each process. Since these procedures are highly application dependent we shall not consider them any further here.

To represent independent processes requires the use of only three primitives: *activity*, *set process condition*, and *delay*. The basic element of a process expression is the *activity*, a schedulable unit of work with a bounded execution time. Decision points in PAL expressions are provided by *process conditions*. The *set process condition* primitive, STP(pc$_i$,expression), is used to modify the i-th process condition pc$_i$, by assigning to it the value resulting from the evaluation of the *expression*. The scope of a process condition is limited to the process in which it is declared. The *delay* primitive is used for timing purposes, where D(n) provides a delay of n time units. We will use the notation p_k to designate any generic primitive in PAL and X_i to denote any well formed expression of p_k's.

Precedence relations between primitives or sub-expressions are defined by two control constructs. The first is the *sequence* construct, $(X_1 \bullet X_2 \bullet \ldots)$, which

[4] Alternately, a system in PAL could be specified as a set of event responses $\{R_1, R_2, \ldots R_n\}$ with each of these responses including one or more processes $\{P_{i1}, P_{i2}, \ldots, P_{ik}\}$ that are all initiated by the starting event S_i.

specifies that each expression can be started only after the previous expression in the sequence has been executed. The construct is completed when the last expression in the sequence is executed. The other is the *concurrent* construct, $(X_1 \parallel X_2 \parallel \ldots)$, which specifies that all the expressions in the construct are independent, and given the availability of resources, any number of them can be executed in parallel. The construct is completed when all the expressions in the construct have been executed.

Conditional execution of primitives or sub-expressions is provided by two additional control constructs; *case* and *repeat*. The *case* construct, $(\text{lexp}(pc_i)_1: X_1 \mid \text{lexp}(pc_i)_2: X_2 \mid \ldots)$, is used to choose a specific execution path, based upon the evaluation of the *logic expression* $\text{lexp}(pc_i)$ containing the specified process condition. To simplify validation, the set of logic expressions must be defined such that at any time one and only one of them is true. The *repeat* construct, $< X : \text{lexp}(pc_i) >$, specifies that the expression X is repeatedly executed until the logic expression $\text{lexp}(pc_i)$ becomes true.

In addition to the algebraic representations introduced above, the PAL control constructs can also be represented in *pseudocode* or in graphical format as shown in Figure 7. The graphical representation, referred to as *process activity net*, provides an easy visual check of the process expression.

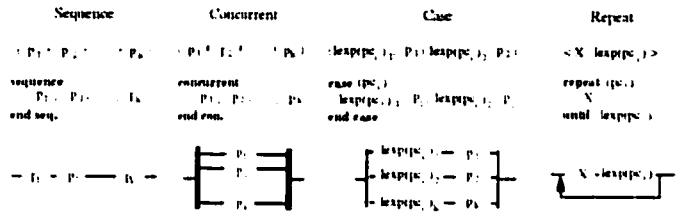


Figure 7. Basic PAL control constructs

Independent PAL processes can be represented using only the above specified primitives and control constructs. A simple example represented in functional, pseudo code and graphical formats is presented in Figure 8. Note that in pseudo code notation, a process is differentiated from the sequence construct by enclosing it in a **begin** ... **end** block. This figure also includes the corresponding *activity timing chart*, a component of PAL that shows the execution times of the various activities of a process. The relative execution times assigned to each activity in the chart are generally estimates of the average or maximum time needed for the execution of the activity. In terms of processes the activity timing chart represents *best case* situation as it assumes that there is an available executor as soon as the activity is ready to execute. The activity timing charts also help to indicate the hierarchical nature of PAL. The shown expression can be viewed, at a higher level of abstraction, as a composite activity with the execution time equal to the combined execution times of its primitive activities.

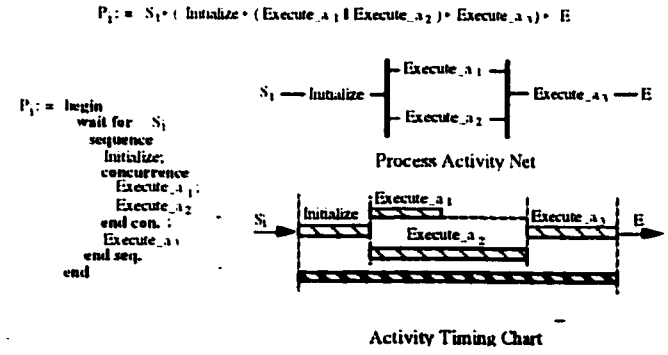


Figure 8. A simple example of an independent process

In PAL, communication between dependent processes is done via two distinct mechanisms: *message passing* for direct communications and the use of *billboard conditions* for posting conditions of general interest.

Messages are passed between PAL processes via named, one-way, finite capacity channels operated on by two messaging primitives: *transmit message*, TXM (channel, parameters) and *wait for message*, WTM (channel, parameters). If a channel is empty, the process executing the WTM will wait until a message is available. If the channel is full, then the process executing the TXM will wait until after a message has been removed from the channel. Thus if the channel is not full, the TXM primitive is non-blocking. For each channel a queue length of 0, 1 or n, has to be specified in the receiving process.

A billboard condition is basically a global variable defined such that it can be globally read, as on a billboard, but can only be modified by one of the processes in the system. The primitives used to set and read system conditions are: *set system condition*, STS (sc_i,expression), and *read system condition*, RDS (sc_i,pc_j). The STS primitive sets the system condition, sc_i, to the value resulting from the evaluation of the expression. To avoid non-determinism, multiple updates of a system condition within a process must be done in a sequential order. The RDS primitive assigns to the process condition, pc_j, the current value of the system variable, sc_i.

One can also access an environment condition, ec_i, by using the *read environment condition*, RDE (ec_i,pc_j), primitive which behaves in the same way as the RDS primitive. To use a system or an environment condition within a process, the value of that condition must first be assigned to a process variable via the proper read primitive. A process can also wait to receive data from the environment via the *wait for event*, WTE (channel, parameters) primitive. Messages and events can represent either pure synchronisation or the transfer of a data item(s).

The basic set of primitives and control constructs defined above is enough to specify most systems. In order to simplify the specification of certain systems, PAL can be expanded to include additional primitives or control constructs, as long as the "single input/single output" condition of structured systems is maintained. For example, the control constructs *select* $(X_1 \mid X_2 \mid \ldots)$, *simultaneous* $\{p_1 \parallel p_2 \parallel \ldots\}$ and *alternation* $\lfloor p_1 \parallel p_2 \parallel \ldots \rfloor$ were found to be useful. The pseudo code and graphical representations of these constructs are shown in Figure 9. The select construct is similar to the one used in Ada. Each path is examined for eligibility to be executed, based on the first primitive in the path and determined as follows: All read (RDS, RDE) and set (STP, STS) primitives are always eligible. The wait (WTM, WTE) primitives are eligible if there is a message in the specified channel. The transmit (TXM) primitive is eligible if the specified channel is not full. An activity A_i is eligible if there is an available processor to execute it. Of the alternatives that are eligible one path is chosen non-deterministically for execution. If none are eligible, then the first path to become eligible is selected. One could also define a *guarded select* construct in which the guards, logical functions of process conditions, are first evaluated and only the paths whose guards are open are checked for eligibility.

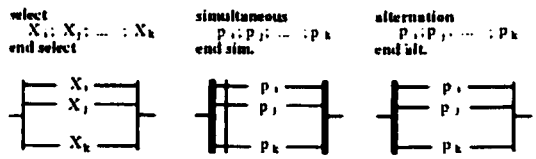


Figure 9. Additional PAL control constructs.

The simultaneous and alternation constructs are basically variations of the concurrency construct and are only specified for primitives (i.e. not expressions). In the simultaneous construct, the start of execution of all the primitives within the construct is delayed until all the necessary resources are available to allow them to start simultaneously. The construct is considered completed when all the primitives are executed. In the alternation construct, execution of each of the primitives is started as resources become available. The construct is considered completed when the first primitive is finished executing, all the others are purged.

4.2 A Specification Example — A Mobile Robot

To show how PAL can be used, here we outline the first step in the requirement specification of the controller of a mobile robot[5]. The robot has to navigate in a partially known environment and has two main types of sensors: A *proximity sensor* that detects close objects and is used for alarm and for docking purposes, and a *range sensor* that can detect more distant objects and is used for recalculation of path and obstacle avoidance. Both sensors provide a *temporary world model* that can be integrated with the *main world model* that contains all the relevant information about the robot's environment.

At the highest level, when the robot receives the command *Go to A*, it retrieves a *rough path* as a set of steps from the world model, it computes the controls needed to execute these steps one step at a time, and when it reaches point A it executes a *docking operation*. Before transmitting the command to execute a step, the controller checks if the step can be executed using information received from the world model. One way to specify the required operation of the controller is to use four processes: *movement control, proximity sensor, range sensor* and *world model* all started by the event *Go to A*. The precedence relations of the constituent subprocesses are shown in the process activity nets of Figure 10.

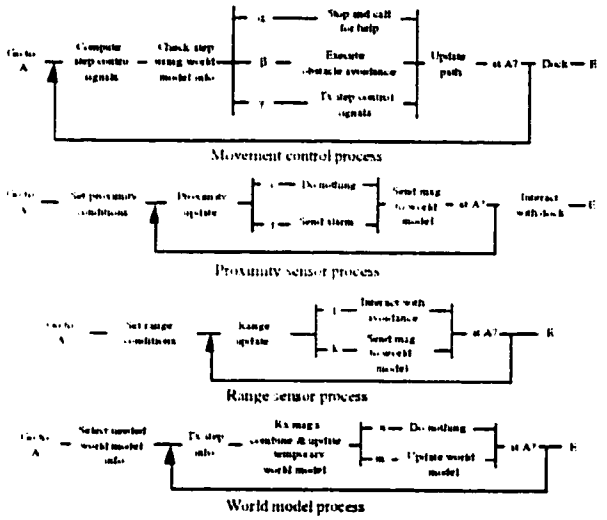


Figure 10. High level specification of a mobile robot

The next step is to define in detail each of the subprocesses. At this step one also has to consider the synchronization between the processes by including the details of needed message exchanges between subprocesses. Note that beyond the direct interactions as represented by the messages, these processes also interact indirectly via the world image. When specifying the system one can start up with a very rudimentary world model that will be refined as the robot moves between different points.

5. Multiactivity Based Design Environment

In this section we shall outline the prototype based design methodology which is highly suited for embedded systems, and indicate the feasibility of developing a set of PAL based tools that can realise such prototypes.

5.1 Prototype Based Design

Because the environments of most embedded systems can not be completely charaterized nor controlled completely, the design of these systems must rely strongly on a *"what-if"* type of thinking. Hence one has to use an interactive development that can only be provided by prototyping. Another major reason for adopting the prototype based design methodology is the fact that most embedded systems have to meet safety requirements that can only be checked out experimentally. As indicated in Figure 11 the design process [Luq89] uses two kinds of prototypes: A *requirement prototype* to check if the requirement specifications are complete and accurate (timing or other performance constraints are ignored at this level), and a *design prototype* used to check if all system constraints can be met for all working conditions. As such, these prototypes are used to check whether the specifications will meet problem requirements in terms of behavior and performance respectively. Note that prototyping can only be used to *detect* errors and not to *prove* their absence. Therefore, to provide the needed confidence to users that the system works as intended, the prototypes have to be easy to use and extendible to allow users to experiment widely and adapt them to specific needs.

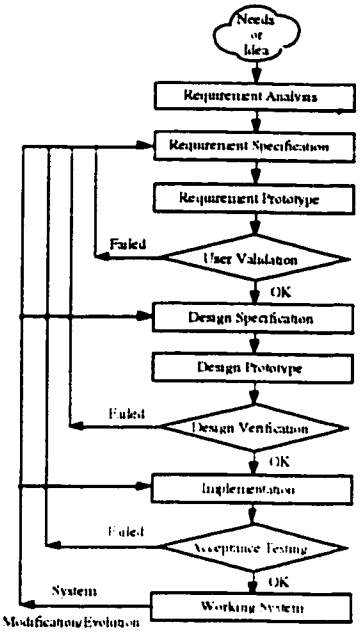


Figure 11. Prototype based system design

5.2 PAL-Based Design Tools

For the proper development of an embedded system, the application engineer must be provided with a set of basic design tools which can be used as prototypes in a well defined design methodology. Such a tool set enables the computer literate application engineer to go from requirement analysis to complete design specification. To demonstrate that one can have a complete multiactivity design environment the following tools (really tool prototypes) have been developed:

Extended PAL Editor [Chu90] — it allows the user to enter, modify and integrate partial responses defined previously either as PAL expressions or in pseudocode. To aid the user, the editor executes comprehensive lexical and syntactical checks on the entered expressions and provides a quick visual check by translating them into properly indented pseudocode. A graphical module to generate process activity nets is under development.

PAL Verifier [Har89] — it is used for checking temporal correctness (deadlock, starvation and other safety and liveness properties) of interacting PAL processes. The verification method is based on the model checking approach; it checks if correctness properties, expressed as branching time temporal logic formulae, are satisfied on a global state graph representation of the system's temporal behaviour.

PAL Simulator [San90] — it interprets PAL processes and runs them assuming that the activities correspond to predetermined time delays. To check whether the PAL processes are correctly specified, the simulator is run assuming infinite resources (very large number of general purpose activity executors). To check if the system can meet the design requirements (constraints), the user can specify the number and type of activity executors, the allocation of activities to executors, a number of activity scheduling schemes and can also simulate various types of executor failures. The simulator also includes an event generator to simulate the different types of external events.

[5] This example represents a highly simplified version of the mobile-robot under development at the Institute for Information Technology at National Research Council Ottawa [LG90].

5.3 Validation of Requirement Specifications

In the PAL-based design environment, validation of requirement specification corresponds to examining whether the PAL expressions correctly specify the required operation of the embedded system. As indicated in Figure 12, this can be done in three steps: 1) a lexical and syntax check of each of the PAL expressions is done using the *Extended PAL Editor*, 2) safety and liveness check of the interacting processes is done using the *PAL Verifier*, 3) functional correctness is checked by exercising the PAL expressions using the *PAL Simulator*.

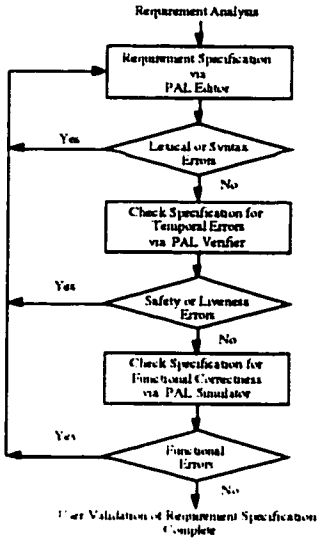


Figure 12. PAL-based requirement prototype

5.4 Design Phase

Design in the PAL-based design environment corresponds to determining the number and types of processors (executors) and an activity to processor allocation that allow the system to meet the specified constraints under all working conditions. Because of the highly iterative and exploratory nature of this phase of the design, the required design prototyping tools must be versatile and easy to use. The general organization of the PAL simulator, called SIMPAL, developed for this purpose is shown in Figure 13. The *controller* part has the same general organization as the *multiactivity coordinator* (shown in Figure 4), but in addition it includes a *statistics compiler*, that generates for each simulation run, data on response times, queueing times, queue sizes, processor utilization, interprocess communications, etc. The *environment* part allows the user to emulate the environment and to introduce processor failures. The *event generator* produces user specified cyclic and probabilistic events to simulate the different working conditions. The *fault generator* introduces user specified, immediate, probabilistic and cyclic failures in the executors. The *man-machine interface* allows for on-line commands and display of various process and system statuses.

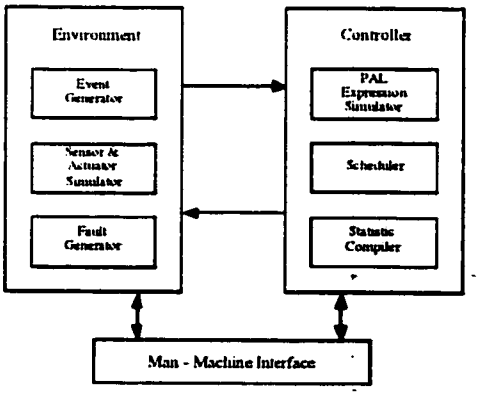


Figure 13. SIMPAL — PAL-based design prototype

Using SIMPAL, the design specification of a multiactivity system can be done via a number of "what if" type experimentations. The nature of the required explorations is highly application dependent; here we shall outline a simple four step scheme that can be applicable to many embedded systems with soft time constraints.

To illustrate the type of results that can be obtained using SIMPAL, we present here some graphs derived from a simple example involving: two external events (E_0, E_1), three processes (P_1, P_2, P_3) and twenty four activities (A_1, A_2, ... A_{24}).

- **Step 1**: Determining the Required Number of Executors

This step represents a first rough estimate of system's responsiveness to events, and can be done in two main ways. The simplest way is to simulate the system assuming that all events start at the same time and check process execution times, as indicated in Figure 14, or to plot response lateness as a function of the number of executors. In this test one generally assumes that each executor can execute all activities and that activity scheduling is done on a first-come first-served basis. Figure 14 indicates that a good first choice is four activity executors, since response execution times are not improved by using more then four. Although this test might look too pessimistic, it does not represent the worst-case conditions as it does not take into consideration dynamic occurrences of events; a new event can occur before the previous response is completed.

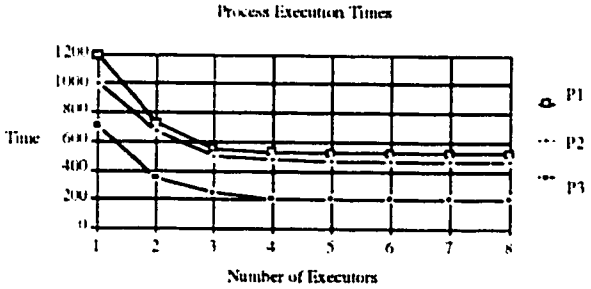


Figure 14. Process execution times as a function of number of executors

An alternate way is to do an individual analysis of the executors needed to meet the timing requirements for each event. An initial estimate of the total number of executors is obtained by combining the individual requirements taking into account the various event concurrencies. In both of the above approaches, specialized executors may have to be included to execute particular activities.

- **Step 2**: Checking System Responsiveness

In this step, using the number of executors determined in the previous step, one checks system responsiveness parameters such as, lateness of responses, percentage of events missed, event queue sizes, etc. (see Figure 15 on the next page) by varying the probability of occurrences of probabilistic events and/or the period of cyclic events. The major difficulty is to cover all the system working conditions properly and to interpret the results correctly.

These graphs indicate that this system, assuming soft time constraints, will perform reasonably well if the probability of occurrence of Event 0 is 15% or less. That is, there will be few late responses and no events will be missed if the queue size of Event 0 is set to 10.

This step, generally involves a large number of simulations, as these systems have many parameters that have to be considered one at a time. Furthermore, as many of the events are probabilistic (as assumed here for Event 0), the simulation runs have to be quite long in order to provide meaningful results. Because of these reasons it becomes quite important that the number of executors determined in step 1 should be realistic.

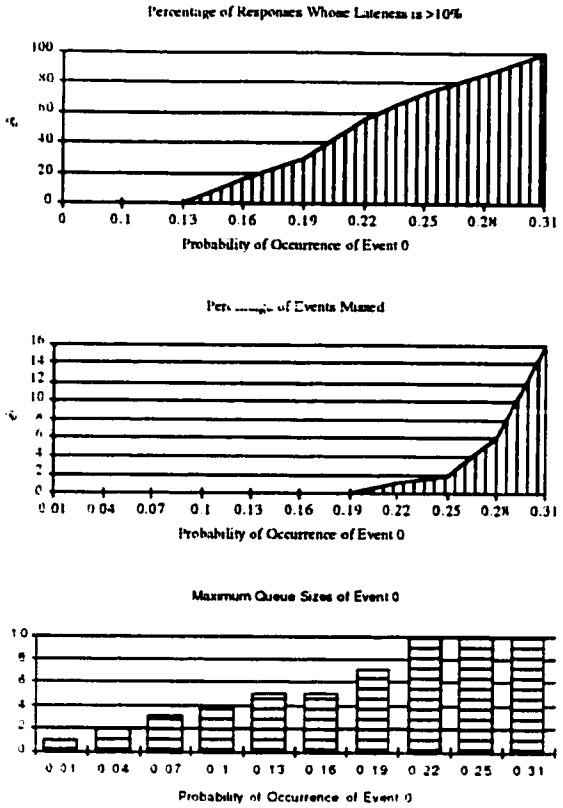


Figure 15. Some system responsiveness parameters

- **Step 3**: Fine Tuning the System

When fine tuning a system there are a number of parameters such as: the capabilities and/or the speed of the executors, executor utilization, allocation of activities to executors, scheduling scheme used, etc., that should be considered. Figure 16 shows two examples: response lateness versus executor power and executor utilization (given that all executors are the same).

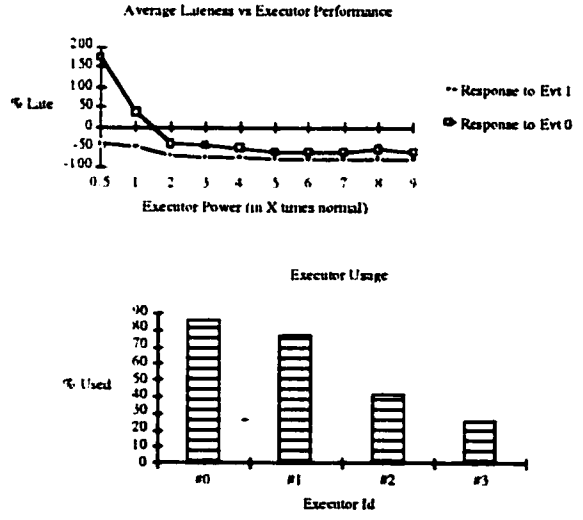


Figure 16. Some fine tuning parameters

These graph indicate that system responsiveness can be improved either by increasing even slightly the capabilities of the of the activity executors or by shifting some of the activities from activity executors 0 and 1.

- **Step 4**: Reliability Considerations

In most embedded systems, there are a number of additional characteristics that should be considered during the design phase. One of the most important characteristics of embedded systems is reliability. SIMPAL has built-in facilities for the generation of executor failures, with variable recovery and detection times, during simulation. Since activities are rescheduled immediately upon the detection of a failure, one can check system degradation for both different failure rates and recovery mechanisms. Intermittent failures are simulated by setting the recovery time to zero. As an example, Figure 17 shows the effect of failure rates of an executor on the lateness of a response at two different event rates (here Event 0 is assumed to be cyclic).

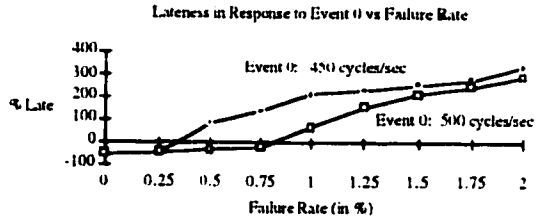


Figure 17. Lateness of response as a function of processor failure and event arrival rates

In SIMPAL it is also possible to specify that critical activities have to be run redundantly and check the corresponding loading on the system resources. For example, Figure 18 indicates that triple modular redundancy (TMR) can be introduced, in the given response, just by adding one more processor. Because the restart of large activities involves longer delays, it is interesting to observe the combined effect of activity granularity and executor failures. Figure 19 shows such an example where some of the larger activities were partitioned into smaller activities. Since the activity partitioning generally means more communication, this figure also includes communication overheads as a parameter. The graphs indicate that the communication overheads can diminish the advantages obtained by partitioning large activities.

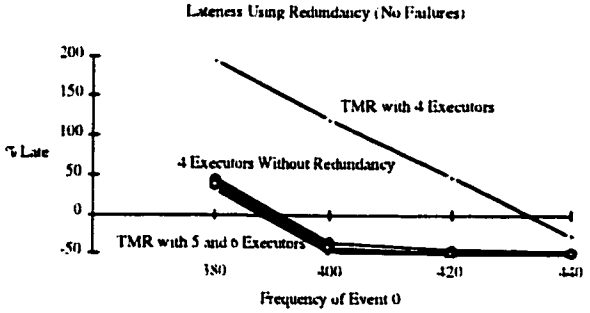


Figure 18. The effect of TMR on response lateness

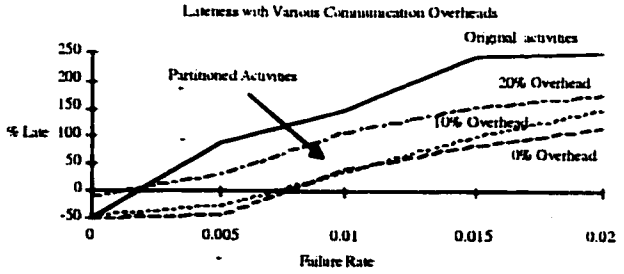


Figure 19. Lateness of response as a function of processor failure rate, activity partitioning and communication overhead

These graphs represent only a sample of the simulation data that can be obtained during this phase of the design. SIMPAL is a fully "instrumented" (one can log all types of information) simulator that allows the designer to experiment widely by analysing the effects of the pertinent design alternatives.

5.5 Review of the Multiactivity Design Methodology

Based on the above, we can now formulate the following multiactivity design methodology of embedded systems:

Requirement Specification — Based on the requirement analysis the application engineer specifies the system as a set of responses using PAL (as a set of PAL expression or in pseudo code).

Requirement Validation — The PAL-based specification is validated using the *Extended PAL Editor, PAL Verifier* and *PAL Simulator* (as outlined in Section 5.3).

Design Specification and Verification — Based upon an estimate of activity execution times, the number and type of activity executors with an activity to executor allocation is obtained using SIMPAL (as outlined in Section 5.4).

Implementation — The system hardware is implemented as a centrally coordinated system (as outlined in Section 3.2), the multiactivity executive (outlined in Section 3.1) and the PAL expressions are loaded into the coordinator, and the activity code is written.

From the above it can be seen that all the above steps, except the last, can be done by application specialists using the PAL-based tools outlined previously. For the last step one may need the help of hardware and software technicians.

6. Conclusions

In this paper we presented the multiactivity paradigm in which the operation of embedded computer systems is viewed as a set of activities whose execution is coordinated in such a way as to produce the required responses to the various events in their environment.

- In terms of system specification, this implies that event responses are to be specified as sets of activities with given precedence relations.
- In terms of system design this implies that in the implementation there is a separation between system resources (the activity executors) and resource management (activity coordination).

To support system specification we have introduced the *process activity language*, a high level executable specification language. We also indicated the feasibility of developing a number of easy to use PAL-based design tools: an editor, a verifier and a simulator. Finally we demonstrated that these tools form a *prototype-based* design environment that can be used by a computer literate application specialist, both for the traditional task of developing the *requirement specification* and for developing the *design specification* for an embedded system; thus eliminating the need for a computer engineer in the design cycle. The prototype based design also allows the application engineer to verify each step of the entire design cycle. What remains is to extend the capabilities of these tools and make them more user friendly.

This paper has two main contributions: First, the basic observation that all work has two main constituents: the "what" part specifying the activities that have to be done, and the "when" part defining the ordering relations in which these activities have to be executed to obtain the proper results. Second, the development of a comprehensive design environment based on a common view; that of the application specialist. This allows the use of problem-specific intuition through the whole design process. Also the proposed design methodology fits present-day technology in that it allows the designer of complex systems not only to do component development but to concentrate on system integration.

The other distinguishing features of the multiactivity paradigm outlined in this paper are the following. First, it does not concentrate solely on software design, like the other CASE tools in this area, but also formulates the required hardware platform, which is critical in embedded systems [LR92]. Second, it permits an extensive analysis of the execution behavior of the total system. Finally, it allows the designer to investigate various fault tolerance characteristics and to include safety features in the embedded system.

The multiactivity-based design methodology, like most modern design schemes, allows for software reusability. For a given application, one can have a library of basic activities and/or templates of standard responses, that can be reused. The multiactivity paradigm does not compromise the designer's ability to specify dynamic behavior so as to permit software reusability. On the contrary, the primary goal is to provide an application specialist with an environment which supports intuitive — response oriented — view of embedded systems. This is in contrast to the object oriented approach, which provides easy software reusability and satisfies other software engineering requirements, but forces an application specialist to accept a view of embedded systems which may be unnatural to him, and worse, makes it hard to specify and test dynamic behavior.

Although the multiactivity paradigm and the associated tool set was developed for the design of embedded systems, we found the basic ideas applicable to other event-driven systems. We used the same principle to define highly modular microprogrammed control units, for both general and special purpose CPU's [VK82, KP84 and MK87] and to design multiple microprocessor systems [Kri79, CK83, FK83a, FK83b and KJ86]. Furthermore, the multiactivity design methodology presented in this paper is applicable with minor modification to flexible manufacturing systems [PKGP86, KHL88, Kri91 and Kri93].

Acknowledgements

The ideas presented in this paper have emerged from long discussions and debates with my graduate students through the years. It is impossible to thank them all. I would like to single out Arto Chubukjian, Pierre Cousineau, Charles Gauthier, Elie Fathi, Randy Harvey, Robert Joannis, Amjad Junaid, Jean Pierre Lachance, Phill Piché, Raul San Martin and Dan Sharron, who have directly contributed to different elements of this work. Special thanks are due to Raul San-Martin and Amjad Junaid who were involved in a previous version of this paper. I also wish to thank the members of the Institute for Information Technology at NRC Canada, whose willingness to listen and freely exchange ideas, have greatly helped me to focus many of my views on system design.

References

[Boo91] Booch G. " Object Oriented Design with Applications" *Benjamin Cummings* 1991.

[Chu90] Chubukjian A. "Extended PAL Editor: An Example of an Interactive Real-Time System Specification Tool" *M.Eng. report University of Ottawa*, April 1990.

[CK82] Cousineau P. and Krieger M., "A Distributed Query Architecture for a Hospital Information System" *Proceedings of the Seventeenth Annual Conference on Information Sciences & Systems*, John Hopkins, March 1983, pp. 44-49.

[DeM78] De Marco T. " Structured Analysis and System Specification" *Yourdon Press 1978.*

[Deu88] Deutsch M.S., "Focusing Real-Time Analysis on User Operations" *IEEE Software*, September 1988, pp. 39-50.

[FK83a] Fathi E. T. and Krieger M. "Multiple Microprocessor Systems: What, Why, and When" *IEEE Computer*, March 1983 pp 23 - 32. Reprinted in *Multi-Microprocessors*- A. Gupta, editor, IEEE Press 1987, pp 4 - 13.

[FK83b] Fathi E. T. and Krieger M. "An Executive for Task-Driven Multimicrocomputer Systems" *IEEE Micro*, October 1983 pp 32 - 41. Reprinted in *Multi-Microprocessors* A. Gupta, editor, IEEE Press 1987, pp 125 - 134.

[Gen88] Gentleman W. M. "Real-Time Applications: Multiprocessors in Harmony" *Proceedings of BUSCON-88 East, The Bus/Board Users' Show and Conference*, New-York, NY, October 1988, pp 269 - 278.

[Har89] Harvey R.A. "Verification of Concurrent System Specifications Using Temporal Logic" *M.A.Sc. thesis University of Ottawa*, July 1989.

[Har90] Harel D., Lachover H., Naamad A., Pnueli A., Politi M., Sherman R., Shtull-Trauring A. and Trakhtenbrot M., "STATEMATE: A Working Environment for the Development of Complex Reactive Systems" *IEEE Transactions on Software Engineering*, April 1990 pp.403-414.

[Hoa85] Hoare C. A. R. "Communicating Sequential Processes" *Prentice Hall 1985*

[HP88] Hatley D. and Pirbhjai I., "Strategies for Real-Time System Specification" *Dorset House* 1988.

[KDK89] Kopetz H., Damm A., Koza C., Mulazzani M., Schwabl W., Senft C. and Zainlinger R. "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach" *IEEE Micro*, February 1989, pp 25 - 40.

[KJ86] Krieger M. and Joannis R. "Process Activity Diagrams: A Tool for the Specification and Design of Multiple Microprocessor Systems" *Proceedings of the ISMM International Symposium on Software and Hardware Applications of Microcomputers*, Beverly-Hills, California, February 1986, pp 157 - 161.

[KHL88] Krieger M., Harvey R.A., Lachance J.P. "Process Activity Language "PAL" for Planning and Coordination of FMS" *MAPL 88 Proceedings of the Symposium on Manufacturing Application Languages*, Winnipeg, Manitoba, June 1988, pp 127-135.

[KP84] Krieger M. and Piche P. R. "Segmented Microprogramming: A Technique for the Design of Special Purpose VLSI Processors" *Proceeding of the IEEE International Conference on Computer Design: VLSI in Computers*, Port Chester, NY, October 1984, pp 396 - 396.

[Kri79] Krieger M. "Task-Driven Multi-Microprocessor Systems" *Proceedings First Canadian Workshop on the Design and Development of Computer Systems*, Ottawa, Ontario, Canada, May 1979, pp 81 - 88.

[Kri91] Kreger M. "Multiactivity Paradigm for the Coordination of Flexible Manufacturing Systems" *Proceeding of International Conference on Manufacturing Systems and Standartization*, Budapest, Hungary, February 1991, pp 10 - 19.

[Kri93] Krieger M. "Multiactivity paradigm for the design and coordination of FMSs" *Computer Integrated Manufacturing Systems*, August 1993, pp 195 - 203.

[Law90] Lawson H.W., "Philosophies for Engineering Computer-Based Systems" *IEEE Computer*, December 1990, pp 52 - 63.

[LG90] Liscano R. & Green D. "READY, An Architecture for a Sensor-Based Mobile Platform" *Proceedings of the Canadian Conference on Electrical and Computer Engineering* Ottawa, Ontario, Canada, September 1990, vol 2 pp. 58.1.1 - 58.1.7.

[LR92] Luqi and Royce W. "Status Report: Computer-Aided Proptyping" *IEEE Software*, November 1992, pp 77- 81.

[Luq89] Luqi "Software Evolution Through Rapid Prototyping" *IEEE Computer*, May 1989, pp 13 - 25.

[MK87] Meng X. and Krieger M. "Segmented Microprogramming in the Design of High Performance Microprocessors" *The Second International Conference on Computers and Applications*, Beijing, People's Republic of China, June 1987, pp 870 - 876.

[Pet77] Peterson J. L. "Petri Nets" *ACM Computing Surveys*, September 1977, pp 223 - 252.

[PKG86] Piché P. R., Krieger M., Gauthier C., Petriu E. "PAD: A New Tool for FMS Design" *Proceedings of The Twelfth Annual Industrial Electronics Conference*, Milwaukee WI. September 1986, pp 794 - 804.

[San90] San Martin R.E. "The Multiactivity Paradigm in Rapid Prototyping of Embedded Systems" *M.A.Sc. thesis University of Ottawa*, July 1990.

[Sha91] Sharon D. "Run Time Kernel for Multiactivity Systems Using Process Activity Language" *Masters thesis Carleton University*, April 1991.

[Sta88] Stankovic J. A. "Misconceptions About Real-Time Computing: A Serious Problem for Next Generation Systems" *IEEE Computer*, October 1988, pp 10 - 19.

[SGME81] Selic B., Gullekson G., McGee, J. and Engelberg I. "ROOM: "An Object-Oriented Methodology for Developing Real-Time Systems" *CASE'92 Fifth International Workshop on Computer-Aided Software Engineering*, Montreal, Quebec, Canada, July 1992, pp 230-240.

[Tak80] Takahashi H., "An Automatic-Controller Description Language" *IEEE Transactions on Software Engineering* . January 1980, pp. 53-64.

[VK82] Vaillancourt S. and Krieger M., "Segmented Microprogramming" *Proc.of 1982 Princeton Conference on Information Sciences & Systems, Princeton, NJ*, March 1982, pp. 231-235.

[War89] Ward P.T., "How to integrate Object Orientation with Structured Analysis and Design" *IEEE Software*, March 1989, pp 74-82.

BIBLIOGRAPHY

1. Krieger, Moshe — *Embedded Systems: Lecture Slides and Article Reprints*; University of Ottawa, 1994
2. Krieger, Moshe — *The Multiactivity Paradigm; An Approach for the Design of Embedded Systems by Application Specialists*; University of Ottawa, 1997
3. Krieger, Moshe — *Coordination Oriented Development Environment (CODE) - An Engineering Approach to Computer Based Systems*; (white paper) University of Ottawa, 1998
4. Krieger, Moshe ea. — *Saturday Prayers Meetings*; round-table discussions, University of Ottawa, 1994 – 1999
5. Alexander, Christopher — *Notes on the Synthesis of Form*; Harvard University Press, 1970
6. Alexander, Christopher — *The Timeless Way of Building*; Oxford University Press, 1979
7. Allen, R; Garlan, D. — *Formalizing Architectural Connection*; 16 International Conference on SW Engineering, Sorrento, Italy, 1991
8. Bergland, G. D. — *A Guided Tour of Software Design Methodologies*; IEEE Computer, 1981
9. Blum, Bruce — *Software Engineering – A Holistic Approach*; Oxford University Press, 1992
10. Blum, Bruce — *A Taxonomy of Software Development Methods*: Communications of the ACM, 1994
11. Brooks, F. P. — *The Mythical Man-Month after 20 Years*; ICSE 1995
12. Brooks, F. P. — *The Mythical Man-Month*; Addison-Wesley Publishing Company Inc., 1979
13. Budgen, David — *Software Design*; Addison-Wesley Publishing Company Inc., 1994
14. Buschmann, Frank ea. — *Pattern-Oriented Software Architecture: A System of patterns*; John Wiley & Sons Inc., 1996
15. Clements, P. C. — *Understanding Architectural Influences and Decisions in large System Projects*; Proceedings of the First International Workshop on Architectures for SW Systems, Seattle, 1995
16. Clements, Paul ea. — *Predicting Software Quality by Architecture Level Evaluation*; Fifth International Conference on SW Quality, Austin, 1995

17. Colorni, Alberto ea. *Distributed Optimization by Ant Colonies*; Proceedings of ECAL Paris 1991
18. Coplien, James O. *Idioms and Patterns as Architectural Literature*; IEEE Software, 1997
19. Cosineau, Pierre *Design of Multi-Layer Information Systems*; University of Ottawa, 1984
20. Dellarocas, C.N. *A Coordination Perspective on Software Architecture: Towards a Design Handbook for Integrating SW Components*; Massachusetts Institute of Technology, 1996
21. DeRemer, Frank ea. *Programming-in-the-Large Versus Programming-in-the-Small*; IEEE Transactions on SW Engineering, 1976
22. Fenton, Norman E. ea. *Software metrics: A Rigorous and Practical Approach*; PWS Publishing, 1997
23. Fernandez, Jose *A Taxonomy of Coordination Mechanisms Used in Real-Time Software Based on Domain Analysis*; Software Engineering Institute, 1993
24. Gacek, Cristina ea. *On the Definition of Software System Architecture*; ICSE – 17 Software Architecture Workshop, 1995
25. Gamma, Erich *Design Patterns: Elements of Reusable Object Oriented Software*; Addison-Wesley Publishing Company, 1995
26. GAO/IMTEC *Patriot Missile Software Problem*; US General Accounting Office, 1992
27. Garlan, D. ea. *Architectural Mismatch: Why Reuse Is So Hard*; IEEE Software, 1995
28. Garlan, D.; Shaw, M. *An Introduction to Software Architecture*; draft for 'Advances in Software Engineering and Knowledge Engineering', 1993
29. Garlan, D.; Shaw, M. *Architectures for Software Systems*; ACM SIGSOFT 1993
30. Garlan, David *Research Directions in Software Architecture*; ACM Computing Surveys,1995
31. Gelernter, D; Carriero, N. *Coordination Languages and their Significance*; Communications of the ACM, 1992
32. Hasselbring, W. *On Defining Computer Science Terminology*; Communications of the ACM, 1999
33. Jackson, Michael *Problem Architectures*; ICSE – 17 Workshop on Architectures for Software Systems, 1995
34. Jackson, Michael *Problems, Methods and Specialization*; Software Engineering Journal, 1994

35. Jackson, Michael — *The World and the Machine*; 17th International Conference on Software Engineering, Seattle, 1995
36. Katzman, Rick ea. — *An Empirical Study of Architectural Design Operations*; University of Waterloo, 1996
37. Katzman, Rick — *Predicting Software Quality Through Architectural Level Evaluation*; 5th International Conference on Software Quality, Austin, 1995
38. Kerth, N. L. — *Using Patterns to Improve Our Architectural Vision*; IEEE Software, 1997
39. Kruchten, Philippe — *The 4+1 View Model of Architecture*; IEEE Software, 1995
40. Kruchten, Philippe — *Mommy, where Do Software Architectures Come From?*; Proceedings of the First International Workshop on Architectures for SW Systems, Seattle, 1995
41. Lent, Bogdan — *Dataflow Architecture for Machine Control*;
42. Malone, Thomas W. ea. — *The Interdisciplinary Study of Coordination*; ACM Computing Surveys, 1994
43. Martin, James — *Computer Data-Base organization*; Prentice-Hall Inc., 1975
44. Mellor, Stephen J. — *Why Explore Object Methods, Patterns and Architectures?*; IEEE Software, 1997
45. Monroe, R. T. — *Architectural Styles, Design patterns and Objects*; IEEE Software, 1997
46. Papadopulos, G. ea. — *Coordination Models and Languages*;
47. Parnas, David L. — *Mathematical Methods for Software Documentation*; slides, SW Engineering Research Group, 1993
48. Parnas, David L. — *On the Criteria to be Used in Decomposing Systems into Modules*; reprinted in 'Great Papers in Computer Science' by Laplante, Phillip, IEEE Press, 1996
49. Parnas, David L. — *Software Aging*; 17th International Conference on Software Engineering, Seattle, 1995
50. Perry, D. E., Wolf, A. L. — *Foundations for the Study of Software Architecture*; ACM SIGSOFT, 1992
51. Reeves, Andrew ea. — *A Software Design Framework or How to Support Real Designers*; Software Engineering Journal, 1995
52. Robichaud, Raymond — *Indust-Ray Medical Systems – Business Plan*; University of Ottawa, 1995
53. Rogers, G. R. — *A Simple Architecture for Consistent Application Program Design*; IBM Systems Journal, 1983
54. Royce, Winston W. — *Managing the Development of Large Software Systems: Concepts and Techniques*; IEEE WESCON, 1970

55. Schmidt, Douglas — *An Introduction to Design Patterns*; http://www.wustl.edu/~schmidt
56. Shaw, Mary — *Architectural issues in Software Reuse: It's Not Just the Functionality, It's the Packaging*; SSR'95 Seattle, Copyright by ACM 1995
57. Shaw, Mary ea. — *Software Architecture: Perspectives on an Emerging Discipline*; Prentice-Hall Inc., 1996
58. Stevens, W. P. ea. — *Structured Design*;
59. Tepfenhart, W. M. — *A Unified Object Topology*; IEEE Software, 1997
60. Wasserman, A. I. — *Toward a Discipline of Software Engineering*; IEEE Software, 1996
61. Wegner, Peter — *Why Interaction is More than Algorithms*; Communications of the ACM, 1997
62. Yourdon, Ed — *The Impact of 'Internet Time' on Software Engineering Principles*; Nortel Design Forum, 1998
63. * * * — *The Software Industry Survey*; The Economist, 1996
64. * * * — *Software Engineering Institute Survey on Architecture* http://www.sei.cmu.edu/architecture