Representing Structural Requirements in Software Architecture

Vincenzo Ambriola and Vincenzo Gervasi
Università di Pisa – Dipartimento di Informatica
Corso Italia 40, I-56125 Pisa, Italy
e-mail: {ambriola,gervasi}@di.unipi.it

Abstract
Software architecture is often intended as a synonym of functional decomposition. Recently, the growing interest in quality factors is pushing software architects to explicitly address issues such as reliability, robustness, and efficiency in the early phases of the software process. Also, an effective software process emphasizes the need for requirement traceability in all phases of development. This paper is a first step in the direction of representing and tracking structural requirements (i.e., non functional requirements) in the software architecture. A graphical language, based on the Dean and Cordy proposal, is extended to cope with this new issue, and an example of its use is given.

Keywords
Software architecture, requirements, design language.

1 INTRODUCTION

The growing complexity of software systems, the agreement that a critical factor in their design is the high level organization of computational elements and interactions, coupled with the increasing production costs of software and the decreasing costs of hardware, are formidable arguments for those researchers in quest of new system abstractions. One of the effects of this concrete push is the birth of a new software engineering discipline, *software architecture*, that, in a historical perspective has noble, solid, and old roots (Parnas 1972).

Perry and Wolf (1992) identify the need of a timely approach to structural system aspects. In their perspective, the software architecture is a framework with multiple purposes: it must satisfy the functional requirements, offer a basic design technique, allow effective reuse, and act as foundation for consistency and dependencies analysis.

The increasing costs of software are also imputable to its architecture. In particular, under the pressure of new requirements, software systems are subject to a natural evolution that results in long and expensive rework. These activities often have as a side effect a perceivable reduction of the robustness of the system. We interpret this effect as the lack of architectural maturity. Software production is still dominated by the process of component re-implementation for each new architecture. We miss a set of standard architectural styles associated with the design and the formal description of their components. A software system should be modeled through its architecture, including system structures and topology. This approach makes possible the analysis of properties related to the system structure, and allows to understand how they influenced the structural choices and their consequences. We expect that this analysis can suggest how to improve the qualitative attributes of a software system so that its life span can be prolonged.

The paper is organized as follows: Section 2 compares the classical notion of architecture with the new definition used in the software context; Section 3 lists the software requirements that we consider structural; Section 4 presents a production process that takes into account the distinction between structural and functional requirements. The formalism we propose to represent the architectural model is defined in Section 5, a discussion and a complete example follows; Section 8 concludes the paper with some remark.

2 CIVIL ARCHITECTURE AND SOFTWARE ARCHITECTURE

There are many and different important analogies between civil architecture (i.e., architecture related to buildings) and software architecture. Both derive from a common approach that leads the architect or the software designer to reconcile his creativity with respect to the requirements that a building or a software system must satisfy, as stated by the customer. These analogies concern the use of models, notations, architectural styles, and standards.

Models of classical architecture (usually physical scale models or perspective drawings) privilege the aesthetic impact of a building and say little about its functionality, that is often described separately. On the other hand, models of software architecture are usually diagrams showing structural, behavioural, or information-oriented views of the system, completely ignoring the aesthetic issues implied by the creative process.

As for buildings, architectural styles exist for software, too. For instance, there is a clear distinction between Romanesque, Gothic, and Baroque architectures for the same class of buildings as there is between distributed, client-server, and layered architecture inside the same class of systems. These different structural topologies are, at a high level of abstraction, the aesthetic facet of every software system.

The construction constraints of a building can concern, for instance, acoustic insulation or resistance against earthquakes; analogously, software systems can be required to provide real-time performance, fault tolerance, or computational efficiency. In general, these constraints are not explicit in the system design, partly because they are considered "quality attributes" that play a secondary role with respect to functional requirements, and partly because the models exploited for design do not take into account such factors, and so cannot express these constraints.

There is another point of contact between the two kinds of architecture. Civil architecture has a focus in human living, since ever assumed as a reference point for the architect's actions. Human living has an evolutionary character, and thus the reference point changes with respect to the historical context where the design experience originally took form. This happens also for software, with the important difference that the software architect is confronted with a much faster evolution.

Civil architecture tries to solve this problem by making –qualitative– choices in the design phase. Software engineering is partially following the same direction, moving its focus towards quality attributes that strongly depend upon system structure, even when these attributes are not explicitly present in the requirement list.

3 STRUCTURAL REQUIREMENTS

Structural requirements have significant effects upon the software product; they determine the final quality and justify the design decisions that constrain the implementation of functional requirements. Structural requirements, differently from functional ones, can be perceived as instances of generic categories that play the role of quality factors to be achieved in the development of the software product (Tervonen 1996). These categories are: efficiency, portability, maintainability, expandability, robustness, safety.

- *Efficiency* is how the system exploits its resources, in terms of performance under given conditions (ISO9126). Efficiency can be described in the specification phase and successively prescribed in the design phase. It can be measured, however, only after the implementation phase.
- *Portability* refers to the ability of the system to be moved between (hardware or software) environments (ISO9126). It has attributes that depend upon the ability to be adapted, installed, conformed, and replaced.
- *Maintainability* refers to the ability of the system to react to changes performed to adapt software to environment or requirement modifications (ISO9126). The attributes of this category are: simplicity of analysis, conciseness, stability, and system modularity.
- *Expandability* is the system ability of reacting to the insertion of new components, caused by the request of new functionality that the system must possess. Expandability and maintainability are quality factors that strongly depend upon architecture. At the same time, they are the goals of architectural specification. Both aspects can be better captured through a timely choice of the right architectural style.
- *Robustness* is the system ability of reacting to incorrect inputs or to component faults. It is a quality desired for every system, although only rarely expressed in the requirement list.

- *Safety* is the aspect related to the protection of the system and of the data that it manages. This quality has a great relevance in specific application domains, even if it cannot be formalized by traditional modeling languages.

Other structural requirements are the so-called environment or architecture requirements. These are the true architectural requirements determined by precise hardware or software needs, for instance those of a real-time system or of a system that must be implemented on pre-existing hardware. In general, these requirements cannot be expressed, and therefore validated, by the most common modeling languages; however, they are commonly found in real projects, where they assume a crucial role. They differ from the quality-based structural requirements mentioned above in that the latter determine how desirable (or, how "good") a given implementation is, while architectural requirements determine if the implementation is acceptable at all. Although this kind of requirements are usually understood, and their satisfaction assumed without further mentions, the lack of a notational support for their expression can adversely affect the lower phases of the development process and, above all, subsequent maintenance activities.

4 STRUCTURAL REQUIREMENTS IN THE PRODUCTION PROCESS

As implicitly said by Parnas (1972), the identification of modules and of their responsibilities belongs to the pre-design phase, when assumptions are made about the roles to assign to modules in the context of the whole system. This activity requires a significant *creative effort*, justified by the fact that design is based on the documents produced by the previous phases of the software process, that often do not leave any indication about system decomposition (and indeed, especially at the higher levels, they should not).

The problem is further aggravated when the design phase has the goal of producing a decomposition based on qualitative constraints, either directly determined by the requirements or induced by the expected benefits of a particular decomposition.

To better understand the goals of a system decomposition in modules we need to better understand which are the factors that drive the decomposition. Parnas believes that these factors have an exclusive functional characteristic. The individuation of software functionality and the consequent splitting of the responsibilities for their implementation imply, through the *design for change* motto, the qualitative goals of modifiability and reuse. Reliability and comprehensibility are instead achieved by the complexity decrease generally induced by module decomposition.

The work made by Parnas on system modularization does not discriminate upon the module nature, the characteristics of their interactions, or the different topologies that can be obtained in the system construction. Modularization is, in fact, the first step towards the concept of software architecture,

Factor	Efficiency	Portab.	Maintain.	Expand.	Robust.	Safety
Separation	$^+/_-$	+	+	+		
Abstraction	−	+	+	+		
Composition	+	−	−	−		
Replication	$^+/_-$	−	−	−	+	
Res. sharing	−	+	+		−	−

Table 1 Influence of operative units on structural requirements.

although an architecture satisfying given non functional requirements cannot be achieved only by means of functional decomposition.

The following architectural factors (or *operative units*, according to Kazman and Bass (1994)) influence the fulfillment of these requirements:

- *Separation* is the distribution of system functionalities among several components, usually in a top-down fashion;
- *Abstraction* is the creation of a virtual machine, implemented by software modules that hide their implementation from the other modules;
- *Composition* is the operation opposite to Separation; it consists of merging together several layers of the design (usually in a bottom-up fashion);
- *Replication* is the duplication of the same functionality in several modules;
- *Resource sharing* is the operation of encapsulating data or services in a handler so that they can be used by several other modules (users of the resource) in a regulated fashion.

As shown in Table 1, there is a distinct relationship between these factors and the categories of non functional requirements we discussed above: + entries denote factors positively influencing the quality, − entries denote negative influences and $^+/_-$ entries denote positive or negative influence, depending on other conditions.

Since architectures can be described by their adoption of the various units (Rosestolato 1996), we can track several pieces of common savvy about the properties of architectures via Table 1. For instance, a client/server architecture makes use of separation and resource sharing, and this positively affects portability, while a blackboard architecture uses resource sharing (where the resource shared is the blackboard itself) and thus is less robust and safe — having a critical point — but better maintenable than other architectures. Given a certain non functional requirement, a software architect can identify the category to which the requirement belongs and determine a set of compatible architectures among those based on the appropriate operative unit. It is also interesting to note that Table 1 makes clear that certain structural requirements are conflicting, and thus cannot be obtained without compromising the soundness of the architecture:

- portability, maintainability, and expandability conflict with efficiency if using abstraction and composition;

- portability and maintainability conflict with efficiency, robustness, and safety if using resource sharing;
- robustness conflicts with portability, maintainability, and expandability if using replication.

In the construction of a software system, modeling and design play a fundamental role. Traditional modeling consists of the functional specification only (using, for instance, Petri nets, Lotos or Z), and describes properties of the system stemming exclusively from its functional requirements. Then, design has to satisfy this functional specification and some additional constraints, i.e. structural requirements, even if they are not explicit in the model.

As a consequence, the same (functional) model corresponds to many different designs, that differ in their choice of non-functional requirements. This is unpleasant, since the "best" implementation of a given system has then to be selected on the basis of unstated choices. We maintain thus that an explicit structural model should be developed alongside the functional one. The architectural specification document describing this model serves both as a descriptive model for analysis, and as a prescriptive model for design. A critical factor in the achievement of this double function is the use of an appropriate notation.

5 THE ARCHITECTURAL MODEL

To synthetically represent a large system we need a high abstraction level. The goal of the language proposed in this paper is to provide a notation to describe the architecture and the structural requirements of a software system. The model is an evolution of the proposal of Dean (1993), refined in Dean and Cordy (1995).

The language aims at providing a comfortable notation, i.e., an easily understandable one (see Figure 1). Beside the graphical notation, there exists also a textual, set-based notation, whose main purpose is to allow a formal definition of the set of well-formed architectures (Rosestolato 1996).

The language allows the designer to express the architecture of a software system, modeled as a set of interacting computational components, without worrying about their content, i.e., about the functionality that they must provide. A model is a graph, whose nodes (boxes) represent the elements of type component and whose arcs (lines) represent the elements of type connector.

The architectural elements of type task are the only active components of the system. They can represent processes, modules, objects, or procedures; more generally, they are computational elements that can manage control flow. The elements of type data are generic data structures. They can be permanent or temporary, structured or not, simple files, databases or archives; they can allow read-only access, write-only access, or both. They can have a variable or fixed dimension. In general, they can be considered as random access memory. Expected connections are denoted by small circles; they are used to indicate expected system interfaces or to represent partially-instantiated

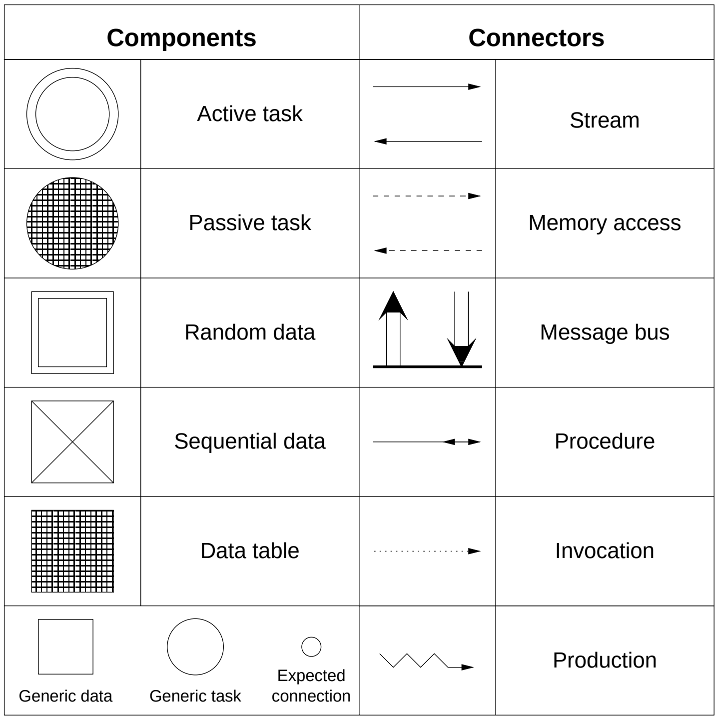


Figure 1 Notation for the formalism.

systems (e.g., when using a component-based architecture). The architectural elements of type connector represent the interactions between elements of type components. They are explicitly defined as architectural entities, that characterize the identity and the role of the participants to an interaction and that, together with the topology of the elements that make up the system architecture, characterize an architectural style.

The language allows the designer to associate to the architectural elements — constants or variables, of type component or connector — a set of attributes that indicate which type or category of non functional requirements they must satisfy. This association constitutes a specific view on the non functional requirements of the system. When embedded in the context of an architecture, we call this the NFR-view (where NFR stands for Non Functional Requirements) of the system. In practical use, attributes are annotated with a reference to the original requirement which introduced them, allowing easy traceability of most design decisions.

This approach has two advantages. The first is to establish a documented link between a requirement and the part of the system (a single architectural element, a pair or an entire subsystem) responsible for ensuring its satisfaction. The existence of such a link and its explicit representation in the language greatly simplifies the requirement verification phase, and guarantees traceability of the design choices while evaluating alternatives or modifying

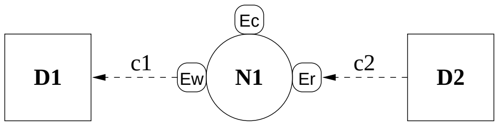


Figure 2 Efficiency constraints.

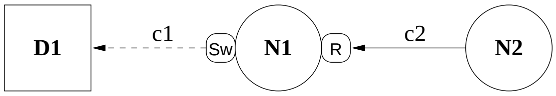


Figure 3 Write security and Robustness constraints.

the design at a later time. Also, in case of evolving requirements, it is much easier to decide which design choices are still valid in a different context and which choices must be reconsidered.

The second advantage derives from the observation that structural requirements are the basis for the software design phase. The refinement of the functional specification, through the choice and the use of particular data structures, is motivated by structural requirements.

The categories of structural requirements that can be expressed in the NFR-view are efficiency, robustness, safety, and many architectural and environmental requirements. The NFR-view collects three kinds of non functional requirements of the efficiency category: computational, write, and read. These requirements are abbreviated as Ec, Ew, and Er respectively. The computational efficiency is a qualitative attribute exclusively related to a single architectural element of type task. The write or the read efficiency are requirements that involve a pair of architectural elements, the first of type task and the second of type connector, which are responsible of the write or of the read. For instance, in Figure 2 computational efficiency is associated to the element N1. The qualitative attributes of read and write efficiency involve, respectively, the pairs of architectural elements $<$N1,c2$>$ and $<$N1,c1$>$.

Robustness (R) and write security (Sw) involve a pair of architectural elements. The first is of type task and the second of type connector, respectively read and write. For instance, Figure 3 shows that robustness is associated to the pair $<$N1,c2$>$, while the responsibility of write security in the data structure represented by D1 is attributed to the pair $<$N1,c1$>$.

The language offers four different types of environmental or architectural requirements: time, synchronous, asynchronous, event. The NFR-view describes them with an association to a single architectural element of type connector. For instance, Figure 4 shows how connector c1, of type memory access, is involved in a time requirement and how the interaction between N1 and D1 happens in time intervals. Architectural elements c2 and c3, of type stream, allow the interaction between elements of type task, respectively in synchronous or asynchronous modality. The write operation made by N3 in D1 is performed each time "event" happens. This requirement is expressed by the graphical element, denoted by Event, associated to c4.

The architectural specification language allows the attribution of multiple

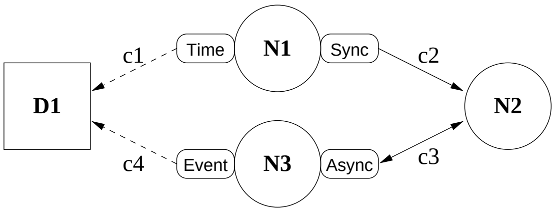


Figure 4 Timing constraints.

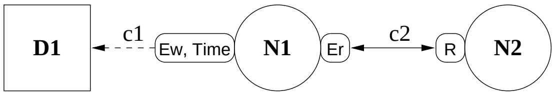


Figure 5 An example of multiple constraints.

structural requirements to the same portion of an architecture. For instance, Figure 5 shows that N1 writes in D1 at given intervals and, at the same time, that this operation must be write efficient. An architectural element of type connector can be responsible of many non functional requirements. Connector c2, together with N1, is responsible for the satisfaction of a read efficiency requirement and it must satisfy with N2 a robustness requirement.

In few words, the architectural specification language allows the (complete or partial) description of a system static structure in terms of components and connectors. The NFR-view associates non functional requirements, and thus qualitative attributes, to portions of the architecture. The architectural specification expresses the potentiality of components to perform a computation and, by means of connectors, the interaction between them.

6 DISCUSSION

Systems are very sensible to changes in the qualitative requirements. When embedded in the system architecture, here defined as the set of active and passive components and of their relations, qualitative requirements can be seen at, and therefore associated to, three different levels: system, modules, calls. At the system level, the qualitative requirements are those that involve the system in its entirety; they are not imputable to part of it as they do not involve single entities. Typical categories of requirements at this level are, for instance, maintainability, expandability, and portability. In all these cases, the focus of interest is the whole system (Boasson 1995).

Among the categories of structural requirements, maintainability, expandability, and partially also portability are directly imputable to the architectural style, as they are related to the system structure and, hence, to its global architecture. The involved items are never single modules or interactions between them and they cannot be associated to single functional parts (those that often drive the system decomposition).

Efficiency, robustness, safety, and environmental or architectural requirements are categories with precise relationships with the architectural elements of a system. They can be listed or described in strict relation with these elements, becoming integral part of the specific system architecture.

Following this approach, the categories of structural requirements can be described in terms of components, connectors, and of their different topologies, i.e., through an architectural model. The idea behind this characterization is that a language can express these requirements in a structured way, without forcing this phase to be too much detailed or to deal with implementation issues. The software architecture is responsible of the structural requirements description with the purpose of supporting the validation process. The examples of architectural specification given in this paper give emphasis to two aspects. Firstly, even if it is possible to associate the achievement of requirements such as efficiency, robustness, and architectural requirements to parts of the architecture, we will never be able to gather all the structural requirements that depend upon the whole system. This is true not only for maintainability or expandability but also for safety, as exemplified by the architecture in Figure 7(b). Secondly, if we start from an architectural specification based on a functional decomposition, and integrate it with an NFR-view, we can gain some benefits already from the design phase, only if we introduce some topological restriction on its architecture or on the nature of its components.

The idea that stems from these two aspects is that we must determine *a priori* the topological restrictions on the architecture and on the nature of the architectural elements involved. We must determine in advance, for each of them, the properties that will be exploited in the achievement of non functional requirements. Different architectural styles, in fact, do not simply determine different designs but endow them with different properties. The choice of a style upon the other has effects on the system description, on its decomposition in parts, on its functionality and performance (Shaw *et al.* 1995). Unfortunately, software designers in practice do not follow this approach, partly because a formalism to document architectural styles, including all benefits and consequences of their adoption, does not exist yet. The absence of an architectural specification, together with the lack of reference architectures, often causes the failure of architectural decisions that can only be definitely recorded in the source code.

7 A COMPLETE EXAMPLE

In this section we show an example of use of the architectural specification language, taken from (Rosestolato 1996): the Weather Data Extractor (WDE) of a radar system. We present some architectural specifications that have significant impact with respect to the treatment of non functional requirements.

Each architectural specification emphasizes, with a distribution of responsibility among modules, specific functional aspects that the designers believe to be the most relevant. At the same time, architectural parts of the system have been made responsible of the validation of the structural requirements that the NFR-view puts in evidence. The exact function of the various modules

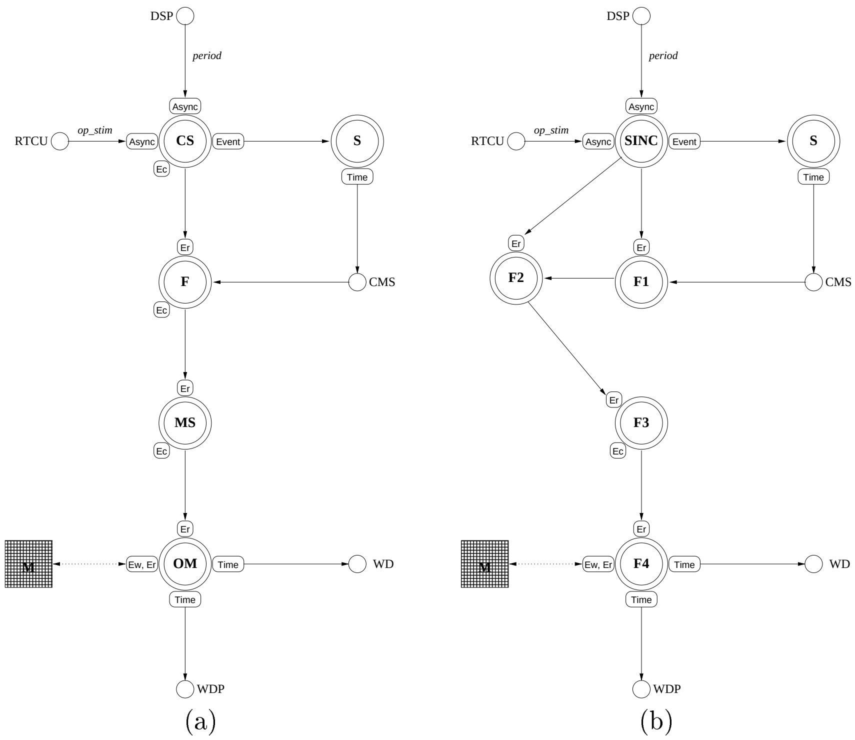


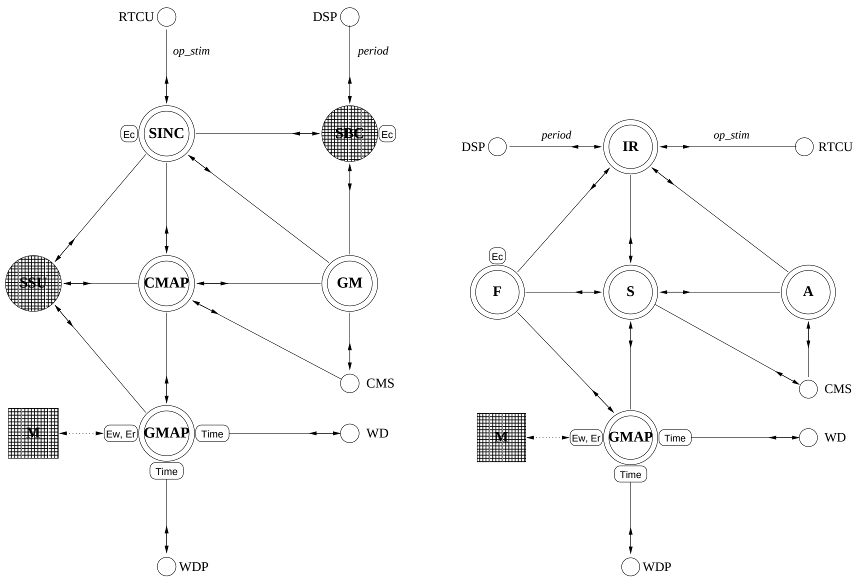
Figure 6 Two pipe & filter architectures.

is not relevant for our discussion, since we concentrate mainly in considering their relations (that is, we look at the architecture rather than at the components).

The specifications shown in Figure 6 are similar. They identify tasks that communicate through streams. Both emphasize the sequence of operations that WDE must perform through a linear sequence of tasks (CS, F, MS, and OM in Figure 6(a); SINC, F2, F3, and F4 in Figure 6(b)) where the last architectural elements (OM and F4) are responsible for the same functionality. These architectures, in fact, depend upon the sequence of events performed by WDE; they follow a strict functional decomposition. For instance, in the specification presented in Figure 6(a), the component CS verifies the correspondence between stim operations and periodgrams, F removes noise from the periodgrams, MS computes the spectral analysis for each periodgram, and OM manages the map organization by sending them to WD and to WDP at given intervals (connectors are labeled Time).

The specification in Figure 6(b) is a little more complex than the previous one, due to the splitting of the F task. This is due to a higher resolution level of the specification.

The specifications in Figure 7 can be defined in object oriented style. In architecture (a), tasks interact via remote procedure calls. This architecture



(a) (b)

Figure 7 Two object-oriented architectures.

has a task (GMAP) that manages the maps produced by the radar and sends them to WD and WDP at given time intervals. This architecture is less comprehensible than the previous one: we cannot identify a linear sequence among the tasks that performs the WDE operations. This specification, still strongly driven by a functional decomposition, is negatively influenced by the lack of topological restrictions, because of its object oriented nature: compare this situation with the architectures in Figure 6, where the strong topological constraints (both architectures are actually a directed acyclic graph) and the information on the type of connectors (all of type stream, and most with Er attributes) would allow us to prove strong properties like deadlock-freedom and preservation of read efficiency along the pipeline.

Figure 7(b) shows another architectural specification in object oriented style, this time emphasizing the issue of WDE state control. This can be recognized by the presence of a central active task (S) that interacts with all the other tasks and with CMS. This architectural specification shows how the need of giving evidence to a non functional aspect has, as a side effect, the introduction of topological constraints. Furthermore, these constraints result to be highly relevant for the achievement of other qualitative requirements.

Finally, Figure 8 shows an example of client-server architecture, where a passive task (LM) acts as server and a set of active tasks act as clients. Clients do not communicate each other and make access to the services provided by the server by means of remote procedure calls. The topological constraint determines a classical star architecture, with the server at the center and

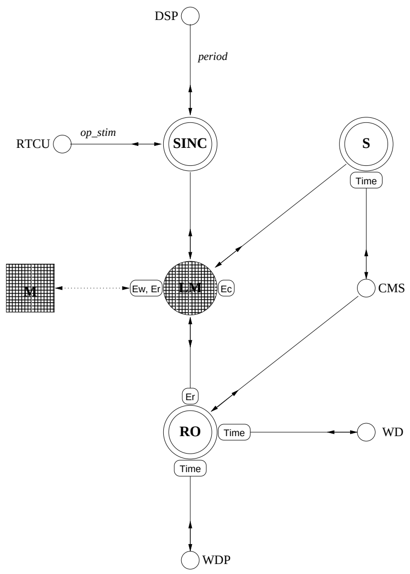


Figure 8 A client-server architecture.

resources only accessible through it. Notice how the load imposed on the server is explicitly pointed out by the Ec requirement on LM.

8 CONCLUSIONS

In this paper we discussed how the traditional approach to system architecture, based on functional decomposition, can satisfy some NFRs (modifiability, reuse, reliability, comprehensibility) in limited ways induced by modularization. However, entire classes of NFRs cannot be obtained by function-based modularization alone. We showed how a common kind of requirements, structural requirements, are directly related to the use of certain decomposition techniques, and thus how the choice of a certain architectural style over another can make easier (or more difficult) the fulfillment of these requirements.

The visual language proposed in this paper allows practical denotation of software architectures, exposing the constraints imposed on parts of the system. More work is needed to classify the topological and structural constraints that characterize properties of the whole system, but we believe that by explicitly tracking the influence of structural requirements in the design, our notation moves a first step toward better integration of non functional requirements in the software process.

Acknowledgements. We wish to thank Massimo Rosestolato for his early work on the subject and for allowing us to use his example in this paper. A long discussion with Adriano Ambriola, a civil architect, inspired Section 2.

REFERENCES

M. Boasson. *The artistry of software architecture.* IEEE Software, 12(6):13-16, November 1995.

T.R. Dean. *Software characterization using connectivity.* Ph.D. dissertation, Dept. of Computing and Information Science, Queen's University, Kingston, Canada, 1993.

T.R. Dean, J.R. Cordy. *A syntactic theory of software architecture.* IEEE Transactions on Software Engineering, 21(4):302-313, April 1995.

D.L. Parnas. *On the criteria to be used in decomposing system into modules.* Communications of the ACM, 15(12):1053-1058, December 1972.

D.E. Perry, A.L. Wolf. *Foundations for the study of software architecture.* ACM Sigsoft Software Engineering Notes, 17(4):40-52, October 1992.

R. Kazman, L. Bass. *Toward deriving software architectures from quality attributes.* Technical report CMU/SEI-94-TR-10, August 1994.

M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, G. Zelesnik. *Abstractions for software architecture and tools to support them.* IEEE Transactions on Software Engineering, 21(4):314-335, April 1995.

M. Rosestolato. *Non functional requirements in the software architecture.* Master's thesis, Dipartimento di Informatica, Pisa, 1996. In Italian.

I. Tervonen. *Support quality-based design and inspection.* IEEE Software, 13(1):44-54, January 1996.

9 BIOGRAPHIES

Vincenzo Ambriola received the Laurea degree in Computer Science from the University of Pisa, in 1976. Currently he is Associate Professor of Software Engineering at the Department of Computer Science of the University of Pisa. His research activity is focused on Requirement Engineering and Software Quality.

Vincenzo Gervasi received the Laurea degree in Computer Science from the University of Pisa, in 1993. Currently he is a Ph.D. student at the Department of Computer Science of the University of Pisa. His research interests include Requirement Engineering, Specification Languages and Active/Deductive Databases.