Software Architectures for Real-time Systems

Anders Wall
Department of Computer Engineering
Mälardalen Real-Time research Center
Mälardalen University
Sweden
anders.wall@mdh.se
MRTC Technical Report 00/20, May 2000.

ABSTRACT

The solution to the complex nature of developing software is software engineering. Software engineering provides techniques for structured design, formal- and informal analysis, and software metrics. The part of software engineering concerned with high-level design and analysis is called software architectures. The objective of architectural analysis is to verify quality requirements on software. It can be applied on any level in the design but it focuses on the structure of the software. While the architecture provides a high-level abstraction of the software, divergences between the designed system and the requirements can be detected early in the design phase. However, the structure of the software alone does not always provide enough information in order to analyze all requirements put upon a software system. Additional information about the software construction is provided by different architectural views. The number of views, and their contents varies depending on the system domain and the required quality properties to analyze.

In this report, the state of the art in the field of software architectures is described. The survey is focused on software architectures for real-time systems but many of the described techniques can be applied to general software systems.

Contents

1 Introduction

The number of projects in industry developing software is constantly increasing. Software is not only replacing old and well-established technologies, but also increasing in size and complexity. To manage the complexity, engineering methods for constructing software needed, i.e. *software engineering*. Software engineering has been established as a broad discipline that covers topics ranging from requirements capture, design, implementation, and software metrics, to maintenance, verification and validation. An established engineering practice is taken for granted in many engineering disciplines but not in the software community. In order to be considered an engineering practice, we must be able to construct models that can be analyzed and verified. Moreover, design methods are needed including established techniques that have been proven successful as well as tools supporting the methods. The part of software engineering that focuses on high-level design and analysis is called *software architectures*.

Edsger Dijkstra pointed out in a paper from 1968 the importance of partitioning and structuring software, in contrast to just focusing on programming to produce the correct functionality [dijk68]. This is what software architecture, and software architectural analysis is about as it deals with how to structure a software system and how to evaluate that structure with respect to different quality properties. The interest in the software architecture field has increased lately due to the increased functionality provided by software systems, the increased size and complexity, and the increased cost of developing and maintaining software products. Today, industry is aware of the benefits of being able to analyze and verify software constructions in an early phase of the development process. If a software development project diverges from the functional requirements or the quality requirements, and if those divergences are not detected early, the cost of revising the design in the end of the project will be significant due to redesign. Almost 80 percent of the cost for developing a software product are spent after the initial design and implementation phases [Clem96b]. These 80 percent spent on maintenance, which includes error detection, correction and evolutionary development.

Not only does a structured description of a software system constitute a basis for architectural analysis, it can also improve the productivity of new members in a project. The architecture provides a simple and holistic view of the whole system. This is very important since complex system usually engage a lot of people, all with unique competencies, at different stages of the development process. Since designing real-time systems usually require multi-disciplinary knowledge, it is very important to have an architectural description that can be understood by software engineers as well as control and mechanical engineers. Furthermore, many software projects employ a lot of consultants. Consultants may have little knowledge of a company's product line and need a quick briefing in order to get productive and cost efficient.

The complexity of software systems also causes problems when maintaining and correcting errors in a software product. It is seldom possible to, in advance, be aware of all the side effects that particular a correction may give rise to. If an architectural description is at hand, it could give some guidance on what modules are most likely to be affected by the correction. This is highly related to evolutionary development. If the architecture of the software construction is violated, it ceases to exist in its former shape. The construction still

has an architecture, but as long as the architecture is not explicitly, and correctly described, it is of no use. Consequently, the architectural description may, and should, evolve as the construction that it describes evolves.

1.1 Towards a definition

There are almost as many definitions of software architecture in the literature as there are software architects and designers. We mention a few examples:

The software architecture of a program or computing system is the structure or structures of the system, which compromise software components, the external visible properties of those components, and the relationships among them [BCK98].

In [Paul94] the following definition is given:

Software architecture not only reflects how the functional requirements are met, but addresses:

1. *non-functional requirements*
2. *design rationale*
3. *architecture style*

Yet another definition is provided in [Clem96a]:

A view of a system that includes the system's major components, the behavior of those components as visible to the rest of the system, and the ways in which the components interact and coordinate to achieve the system's mission.

One property that seems to be common among almost every proposed definition is that the software architecture describes a system by a composition of its software components and their interrelationships. In addition, software architectures should provide a high level description, i.e. a more abstract level than the level that algorithms and data structure provides. However, defining a software architecture only as a syntactical representations of components and their interconnections in the software systems is not sufficient. To be useful, additional information must be present in the description, in particular the semantics of components and connections. Different domains of software systems have different semantics of their software architectural description. A domain defines the class of application to which a product belongs, e.g. desktop applications and industrial control applications. As a consequence, there will be variations in the definitions of software architectures depending on the domain. Furthermore, the definition also depends on the aim of the architectural description, e.g. support for architectural analysis, representation or description of the designed system. It is probably impossible to unify software designers in one single definition as it depends on the aim of the architecture and the domain in which it is used. What we can state is that software architecture is a description of the software structure and methods to evaluate and compare design solutions.

1.2 Open research areas

As software architecture is an immature research area, a lot of open questions still exist. Most of the ongoing research in the field of software architectures is focused on description languages and analysis of architectures for non-real-time systems. The analysis methods are

still informal in their nature. As the analysis methods are informal they provide rough metrics and estimations. We believe that formality can be added to architectural models. Thus, the models can provide means for formal verification of some of the quality properties that are listed in this survey.

Most of the material on software architectural analysis found in the literature ignores the temporal aspects. By adding the temporal dimension on software, completely new problem arises. As an example, components developed for real-time systems, i.e. system for which correctness depend on both the functionality and the temporal correctness, can not be reused in new environments unless at least the temporal constraints are still fulfilled. Quality properties such as flexibility, i.e. the ability of a software system to adopt new, or remove old functionality, are also important. As real-time systems are restricted to resources such as processors, communication busses, etc., a lot of additionally parameters must be taken into account in such an analysis.

The tool support for architectural design and analysis is poor. Tools that support the complete process of developing an architecture are needed. Today, architectural tools for real-time systems almost exclusively focus on schedulability analysis. As indicated in this report, there are a lot of other important properties of real-time systems software. However, implementing such tools is non-trivial. One approach is to use existing tools for automatic verification. This can be done if the problem of analyzing a specific quality property can be transformed into a property that can be verified using that tool. Examples of existing tools for formal verification are UPPAAL and KRONOS [LPY97][DaYo95].

1.3 Outline

Chapter 2 discusses architectural description languages and desired properties of such. In Chapter 3, the architectural view necessary for an architectural analysis of real-time software architectures is discussed. Architectural analysis is dealt with in Chapter 4. Finally, Chapter 5 concludes the report. Terminology used in the paper is explained as it is used. Appendix A provides, however, a complete list of the vocabulary together with a short explanation.

2 Architecture description languages

Communication among software engineers is crucial. Without means for communication, important information into- and from the design phase might accidentally get lost, resulting in misinterpretations. Moreover, a system designer must be able to communicate with customers, other project members and management in an unambiguous way. An unambiguous architectural description is also a necessary condition for performing architectural analysis. A parable is the building trade, where building architects transform the customer requirements into a design. This design must be described in a way the building constructor understands in order to do mechanical strength calculus and for building workers to use as a blueprint. When developing software, a software engineer formalizes the customer requirements. Based on the requirements, a high-level design is described in a language that is commonly understood by customers and designers. The common language is a necessity in order to communicate and discuss design solutions. As output from the high-level design phase, one or several candidate architectural solutions are produced.

To verify that the quality requirements of the system are met by the architectural solutions, the architecture has to be analyzed. Hence, the description language used in the high-level design must support the analysis methods. Once a software architecture is constructed that fulfils the requirements, the architectural description is used as a ”blueprint” when implementing the system. In addition, an architectural description makes maintenance easier since it facilitates the understanding how parts of software systems cooperate. Thus, the parts of a software system, i.e. components and sub-systems, affected by a correction are detected in advance.

2.1 Desired properties of an architecture description language

Languages for architectural description are called Architecture Description Languages (ADL). There is an abundance of ADL:s, each of them with its own specific syntax, semantics, expressiveness and purposes[EHLS94][LKAV93][Vest94]. An ideal ADL should however, provide six classes of properties: *composition*, *abstraction*, *reusability*, *configuration*, *heterogeneity* and *analysis* [SHGA96]. By *composition* is meant that a software system should be described as a composition of components and connections. Furthermore, components and connections must also be described in a way that clearly and explicitly describes the exact role of each element, i.e. modeled on an appropriate level of *abstraction*.

As components are *reused* in different applications that are described using different description languages, the architectural description must be able to adopt to reuse. That is**,** it should be possible to reuse components, connectors describing the interconnection between components and architectural patterns in different architectural descriptions. Related to reusability is *heterogeneity*. Heterogeneity is the possibility of combining different heterogeneous architectural descriptions.

Configuration means that the architectural structure among components in the system should be separated from the structure in the components. The language should also support dynamic reconfiguration. As will be discussed in Chapter 3, the structural view describes all

components and connections, whereas the module view unveils the structure of each component.

Finally, as high-level design analysis is one of the primer justifications for using software architectural techniques, the architectural description must support different kinds of *analyses*.

Considering the desired properties of an architectural description above, how can a software architecture be described? One possibility is a plain textural description in a natural language. However, natural languages tend to be ambiguous, making them really hard to interpret in a consistent manner. By using a formal language an unambiguous description is obtained. With formal languages it is possible to use mathematics when modeling and verifying the architecture. The disadvantage of using formal languages as architectural descriptions are that most of them requires a lot of experience and mathematical skill. Consequently, such a description may be sufficient and useful at some stage in the design process but not for communication with partners in a project without a computer science background. By relaxing the formality, a semi-formal, graphical representation may be obtained. Even inexperienced people can get a feeling for how a system is constructed by interpreting a graphical representation. The semi-formal description also permits analyses and quality predictions to be made as described later in this report. The graphical approach has been adopted by many of the available ADL, where the software design is constructed

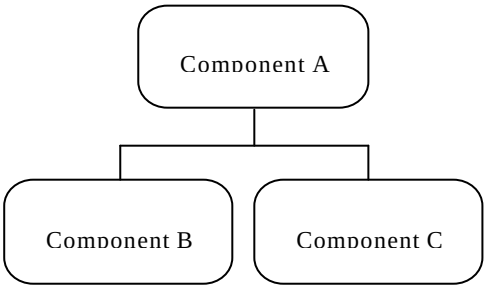

Figure 1. A graphical software architecture description.

using components and their interconnections in a 4th generation language manner as illustrated in Figure 1.

2.2 Semantics of an ADL

The architectural description in Figure 1 provides only the information that there are tree components in the system, which are connected to each other. The connections could indicate a class hierarchy or a network communication link over a distributed hardware architecture. As stressed by Clements and Northrop [Clem96b], it must be known exactly what the components are, what the connections mean and what the position of the components imply, i.e. a well-defined semantics. If the semantics is not clear the architectural description is quite useless.

One single architectural description language can not fit the desired level of abstraction for every different software domain and application. There is for example a big difference between designing a real-time system with hard- and soft temporal requirements compared to designing an administrative application with database management and transactions. Consequently, we need a unique description language for every application domain.

Even though there must be differences in the architectural description depending on the application domain, there might exist a least common denominator. Such a least common denominator could, for instance, consist of components and connections. But the significance of a connection or a component could be domain specific.

If the ADL has an unambiguous semantics, design tools for architectural analyses can be developed [ERGUSA97] [LPY97]. However, analysis of quality properties usually requires more information than just the architectural structure. This additional information is provided by the architectural views and is discussed in Chapter 3.

2.3 Examples of existing architectural description languages

There exist several architectural description languages for real-time systems. Typically they differ in their expressiveness and formality. As an example of a formal modeling language that can be used for describing architectures for real-time systems we use *timed automata* [ALDI92]. Architectures are described in timed automata as a network of finite state machines, where a process or a component is one state machine. Synchronization channels connect processes in timed automata to each other. A synchronization channel defines the name of the signal used for synchronization. Thus, architectural interconnections are described using synchronization. Below is a more rigorous description of timed automata.

A timed automaton is a finite state machine extended with real-valued clocks that increases uniformly. Moreover, transitions in a timed automaton are decorated with guards and actions. Guards are clock constraints that enables or disables a transition, i.e. if the guard is true then the transition can be taken. In Figure 2, the transition from *S1* to *S2* can be taken if the clock *x* has a value greater than 10 time units.

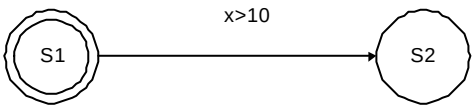

Figure 2. A simple timed automaton

Actions enable synchronization between different timed automata in a rendezvous manner, i.e. processes halts until both participating processes can synchronize. This indicates that a complete system is modeled by a set of timed automata, such a set is called a network and consists of the parallel composition of the included processes. Consider Figure 3 where a small network is displayed consisting of two processes, *A* and *B*.

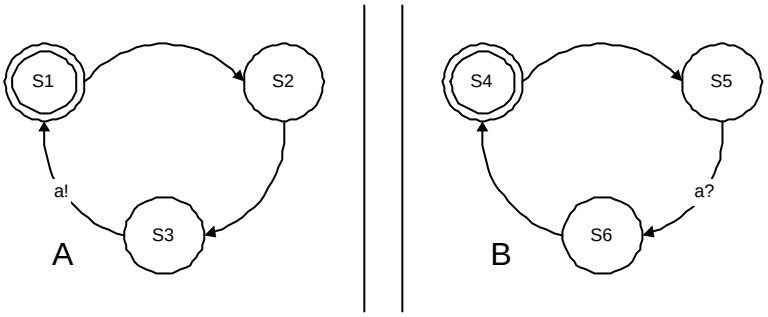

Figure 3. A network of timed automata processes

Whenever process *B* is in state *S5*, it will wait for another process to send a signal on channel *a*. The question mark after the channel name indicates that *B* is waiting for the signal. As long as no signal is being sent on channel *a*, *B* is stuck in state *S5*. As soon as process *A* reaches state *S3*, the processes can synchronize and the processes progress to *S1* and *S4* respectively. During this transition, no clocks are progressing, i.e. it is a discrete transition. Models of timed automata can be constructed and automatically verified using existing model-checking tools, e.g. UPPAAL and KRONOS [LPY97][DaYo95].

Another example of an ADL for real-time systems is *MetaH* [Vest94]. This is a language that models a system on level of abstraction higher than timed automata. MetaH provides means for specifying real-time processes, referred to as tasks, that can be either periodic or aperiodic, communication among tasks, modes and composites of processes and modes that are called macros. Furthermore, the hardware allocation of processes and characteristics of the hardware such as channels that are used for communication among processors can be specified. As the temporal properties of tasks and modes are provided in the models, MetaH support different kinds of real-time analyses such as schedulability analysis. There exist a graphical tool that supports the modeling in MetaH and analysis of real-time software architectures described in MetaH. The schedulability analysis in this tool is based on rate-monotonic [LILA73].

3 Architectural views

Architectural views constitute an important part of a software architectural description as they expose architectural information apart from only the structure. In Figure 4, architectural description languages for different software families (domains), are viewed as an inheritance graph. The top node includes description primitives shared by all domains (compare with a virtual base class in the object orientation community). Two common description primitives could, for example, be syntactical symbols representing components and the connections between components. This means that components and links can describe the structure of any sub-domain of software applications. However, the component primitives and the connection primitives have no semantics in the top node. Semantics and new syntactical symbols will be added while moving down in the inheritance hierarchy. For instance, the semantics of a component in a real-time system is probably a task, and the links are the communication among tasks or precedence relations. In an administrative software application on the other hand, components are most certainly databases or user interfaces, and connections denote database transactions.

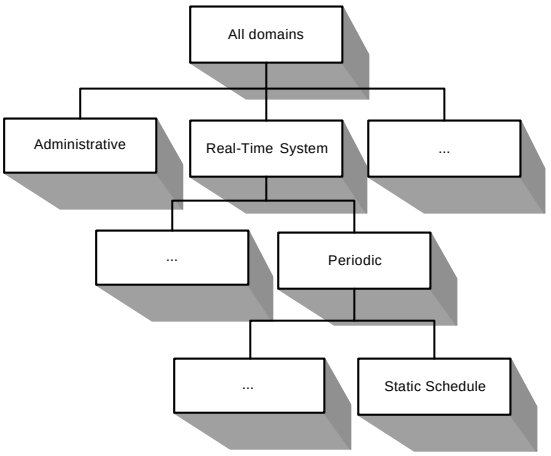

Figure 4. Architectural description and view hierarchy.

The nodes in Figure 4 are intentionally displayed in a 3-dimensional manner. Each side of a cubical node is a metaphor for a unique view of the architecture. There might be arbitrary many different views, depending on the needs for verification and analysis in a software development project.

In this Chapter, views important for the real-time systems domain are discussed. Note that not all views must be modeled in a project developing real-time systems. Only the views sufficient for the analyses required must be present in the architectural description. The names of the views and their contents are not standardized but we propose the following:

- Structural view
- Module view
- Logical view
- Hardware view

- Temporal view
- Communication view
- Synchronization view

Structural view

The structural view describes the overall architectural design and style, providing the highest level of abstraction. This is the natural starting point for an architect designing a software system. The structural view consists of software modules and their interconnections, i.e. the interfaces between them. The syntactical representation of modules and connections is optional but should be uniform within the development project for the sake of communication among engineers.

As design on this level is rather rapid, it is possible to design several competing architectures for evaluation and comparison. Once a software architecture satisfying the quality requirements is selected, it is settled. Depending on the required analyses, more views might have to be modeled in order to make a correct decision. For instance one or more of the views proposed in this chapter could be considered.

In Figure 5, the structure of a system consisting of four components is displayed. The arrows between the components represent function calls through the component interfaces.

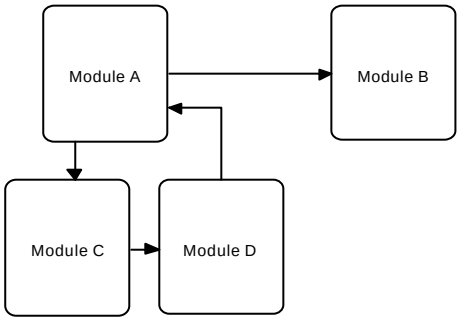

Figure 5. The structural view of the software architecture.

In the design methodology called *Module Approach to Software Construction, Operation and Test* (MASCOT), the structural view is modeled with a diagram called the *decomposed component level view* [Masc87]. This view provides a decomposition of a sub-system into its main constituents, i.e. its tasks.

The object-oriented methodology for real-time systems called *Hard Real-Time Hierarchical Object-Oriented Design* (HRT-HOOD), also has a structural view that is provided by the so-called parent-objects [BuWe94]. A parent-object is a component on its highest-level that may be further decomposed.

The corresponding abstraction for networks of timed automata is the processes. A system modeled in a network consists of a set of automata (processes). Each of these processes could be seen as a component. The interconnections are modeled using synchronization actions. Interconnections visualize the data flow. Information of the control flow is given by the logical view, which is discussed in Chapter 3.3.

Module view

The module view exposes all the functions, methods or sub-modules in all the components modeled in the structural view. A software component is a software module, which is further, decomposed into functions and sub-modules in order to unveil the division of functionality. This view should also describe the interactions between the functions. It is, for example, desirable that the interaction between functions in different components is held to a minimum. Some communication between components is necessary, but the communication must be performed trough well-defined interfaces that conceal the underlying functionality.

Hierarchical methods such as MASCOT and HRT-HOOD both provide means for component decomposition. In MASCOT the module view becomes the structural view as each component is refined, while in HRT-HOOD, the module view is described by child-objects derived from each parent-object.

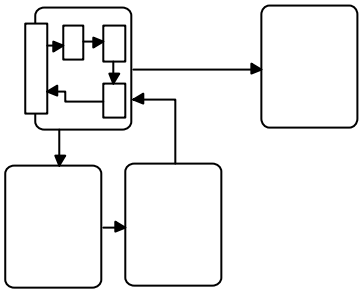

Figure 6. The Module view of components.

Logical view

In this view the functions from the module view is described in more logical details. It serves as a model of the actual implementation, which can be used as a low-level description or constitute the basis for formal verification. Some possible descriptions are state machines or algebra like CCS [Miln87]. These are all different ways of describing the functionality of software formally. State machines can be of different types depending on the application. For example, timed automata can be used for real-time systems as it provides a notion of time as well as concurrency [ALDI92]. If time is of no concern, an ordinary state machine can be used. CCS is a process algebra with which it is possible to model concurrent systems. Such algebra is useful when modeling communication and synchronization, which is essential when designing real-time systems.

In Figure 7, The logical view for the sub-components is modeled using time automata. The upper sub-module synchronizes with the lower sub-module by sending signal *a*.

From the software architecture perspective, the logical view may be on a far too detailed level since software architectures are descriptions of software systems on a higher level than algorithms. However, this view will eventually be implemented, if not in logic so in the chosen programming language which in itself is a formal description of the specification.

The logical view is of no interest when settling the architectural style. It provides a basis for formal verification and in the end the program source code.

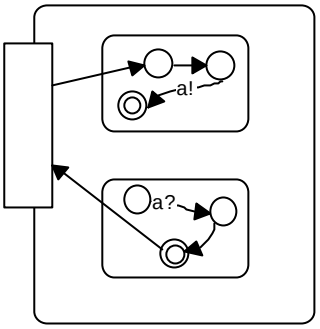

Figure 7. The logical view.

Hardware view

If the system is distributed, i.e. a set of interconnected and geographically separated CPUs, or a multi-processor system, i.e. a set of interconnected and geographically collected CPUs, there might be requirements of pre-allocated functionality among the nodes in the system. Such an allocation will affect the final architecture and the performance of the application. Yet another reason for having a hardware view description in the software architecture is the issue of portability. If software should be easy to move between different types of platforms, the dependencies to the hardware and the operating systems must be encapsulated from the rest of the software system. One can discuss whether this is a software architectural view or not, but as long as hardware has an impact on the software architecture, we consider it a view.

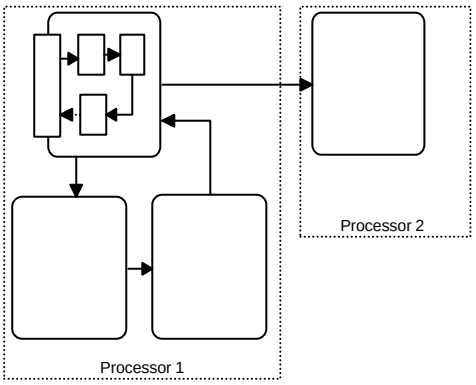

Figure 8. The processor allocation in the hardware view.

In the Yourdon Structured Method (YSM), the allocation of functions to hardware processors is called the *processor environment model* [Cool91]. Besides the function allocation, this view reveals the data that will be communicated among the processors.

Temporal view

The views discussed so far are common among different software families and consequently reside in the topmost node in the architectural hierarchy shown in Figure 5. The temporal view is, however, domain specific. As the correctness of a real-time system not only depends on correct function, but also correct timing, the temporal constraints must be

present in the architecture. By correct timing we mean not too early and not too late. In order to verify whether or not tasks in a real-time application will be schedulable, i.e. all temporal constraints are fulfilled such as all deadlines are met, we need a view of the temporal requirements.

The temporal view contains data such as release time i.e. the earliest start time of a task, the deadline i.e. the latest completion time of a task, the period time (the frequency) of a task, etc. We say that a *task model* determines the exact content of the temporal view. The exact appearance of a task model varies depending on the *execution strategy*. The execution strategy defines the rules that determine what task to execute.

As an example of a variation in the temporal view, consider a periodic task that samples a sensor in a process. As the sampling should be performed with some specific frequency in order to obtain a correct view of the process, a period specifying the interval between two consecutive executions of the sampling must be specified. In contrast, if the application is purely event trigged, i.e. tasks have arbitrary release times, there is no need for specifying period times. Instead, the minimum inter-arrival times must be specified for the tasks.

HRT-HOOD has a temporal view that is divided into two parts, one that describes the execution strategies for a class and one that provides the temporal attributes. The execution strategy can be either *cyclic* or *sporadic*. Depending on the execution strategy, classes can be assigned, e.g. period times, minimal inter-arrival times, and deadlines.

In timed automata, clocks and guards on clocks describe the temporal view.

Communication view

For telecommunication systems, and for real-time systems in general, it is desirable to model communication among tasks and processes. Communication is typically performed using messages and signals that are sent back and forth in the system, either locally on one processor or among nodes in a distributed system. For this purpose the communication view can be used. In Figure 9, the communication is visualized with Message Sequence Charts (MSC). The vertical line in each process depicts time which increase downwards. The horizontal lines between the processes depict the messages or signals.

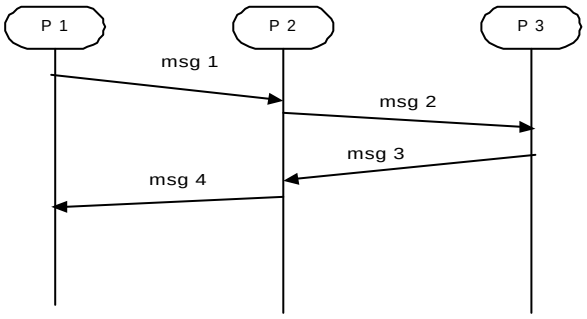

Figure 9. Message Sequence Chart

The MSC can be translated into ordinary finite state automata which makes it possible to formally verify them using, for instance, temporal logic [LaLe94].

Synchronization view

As real-time systems often are multi-tasking systems having several tasks running concurrently, it is necessary to synchronize access to shared resources in order to avoid inconstancy. Tasks that uses a shared resource must mutually exclude each other, i.e. only one task can use the resource at the time. There exist several techniques for handling mutual exclusion in real-time systems, e.g. semaphores, signals or separation of task in time. In addition, to guarantee precedence relations, i.e. requirements of the execution order among tasks in a system, synchronization is necessary.

What synchronization technique to choose depends on the provided infrastructure, i.e. the real-time operating system (RTOS), and the available task models. For instance, if the system is pre-run-time scheduled, i.e. a pre-runtime generated table defines the execution order of the tasks, time-vise separation of tasks can be used. On the other hand, if the system is event-triggered, and semaphores are the only means for synchronization provided by the infrastructure, the semaphore approach must be used.

The information unveiled in the synchronization view is implicitly present in other views discussed in this section. For instance, if synchronization is resolved by separation in time, this is visible in the temporal view, or if signals are used, this is visible in the communication view.

In MASCOT, communication and synchronization is modeled using *paths* along which entities communicate. A path can indicate a dependency to commonly used data, or a dependency to another entity that results in a sending/receiving of messages.

Communication and synchronization between processes can be modeled in timed automata by using synchronization actions.

3.1 Discussion

All the different views should not be designed in the beginning of a development project. Instead an iterative process is often preferred. For some applications, some of the views can be excluded. For instance, if there is no distribution and no requirements regarding portability, the hardware view may be excluded.

There exist relations among different architectural views. The relation between the structural view and the module view is obvious as the module view provides a decomposition of the architecture specified in the structural view. The logical view defines the ”low-level-design”, specified in some formal language suitable for formal verification of, for instance, the communication and synchronization among the modules in the software system. The schedulability of a distributed real-time system depends on how tasks are allocated, i.e. how the tasks are distributed. The allocation affects the utilization of each processor and the time spent on communication between tasks allocated on different processors.

4Architectural analysis

The main incitement for using software architecture notation when designing a software system is the ability to analyze and verify the design in an early stage of the development process. By comparing different candidate architectures, confidence in early design decisions is achieved. Such a comparison is done by listing pros and cons for each architectural solution according to the quality requirements put on the system. Furthermore, architectural analysis enables the possibility to get software metrics based on the high-level design, e.g. the level of coupling and cohesion within and between the different modules that constitute the software system [Fenton96].

In this report, the software system quality properties are divided into two different classes, functional and nonfunctional. Functional quality properties are those concerned with the runtime behavior of the software, e.g. performance or reliability, whereas nonfunctional quality properties are concerned with the quality of the software itself, e.g. maintainability or reusability. Most of these software quality properties are qualitative rather than quantitative, thus being practicable only for comparison between different architectures.

4.1 Methods for architectural analysis

An architectural analysis process is divided into two stages, *questioning* and *measuring*. The questioning phase generates questions that are answered by the measuring phase. Len Bass et. al. [BCK98], have categorized the questioning stage in architectural review and evaluation into three different classes namely *Scenario-based*, *checklist-based* and *questionnaire-based*.

Scenarios are a set of cases where the software architect asks a lot of ”what if” questions that reflect the requirements. It is however not a trivial task to construct the right questions and to know when to stop generating scenarios. This requires a lot of experience and knowledge, which can be achieved by being involved in many design projects. A *scenario* is always system specific, i.e. tailor-made for a particular application in a domain, whereas questions that are valid for all architectures in a particular domain resides in a checklist. The items in the checklist can either generate scenarios or be verified in the measuring stage directly. As an example, consider the domain of safety-critical real-time systems. The checklist contains the following items:

1. Is the system schedulable?
2. Is there error recovery code in the system to clean up after error detection?

The first item is verified directly by performing a mathematical schedulability analysis. The second item is too general and therefore it must be formalized into a set of scenarios before it can be answered. As scenarios are system specific, they can stress different types of errors in specific modules residing in the system. One possible scenario is: “*What happen when division by zero occurs in the control task*”. The scenarios can than be verified by, for instance, simulation or scenario execution, both described later in this chapter.

The questionnaire-based questioning typically stresses general logistical software architecture questions. These questions have usually very little to do with the quality of the

software itself, but rather focusing on issues such as documentation, and how the architecture was generated. Although the logistical questions do not examine the quality of the software product itself, it has impact on the quality since good quality requires a mature development process. Examples of such questions are: "Is a standard architectural description language used?", or "Is the intended work distribution supported by the architecture?".

There are a couple of measuring techniques available for architectural analysis namely *scenario execution, simulation* and *prototyping, mathematical methods* and *experience based knowledge reasoning*. The idea with *scenario execution* is to "execute" the question stated by a scenario on the architecture. By executing a scenario is meant that the effects on the architecture imposed by a scenario is investigated. This method is particular suited for analysis of non-functional quality properties.

Simulation requires a prototype implementation of the architecture. Such a prototype should be as small as possible, containing only the information needed for the analysis to be performed. Simulation is a method targeting on analysis of functional quality properties.

Experienced-based reasoning can be used for any of the two classes of quality properties. Actually, experienced-based reasoning is usually how the software architecture evaluation is done in industry today, although in a relatively unorganized manner. As an organization and its development process mature, more of the formal evaluation techniques will be adopted.

Mathematical methods can be used provided that a mathematical model of the architecture exists. Such a model is provided by, e.g. timed automata. More examples of mathematical measuring techniques are the schedulability test for real-time systems and statistical reliability modeling. These methods give a clear yes- or no answer, or a quantitative value that is comparable among all different types of software applications.

Figure 10 provides a schematic picture of how the different evaluation techniques relate.

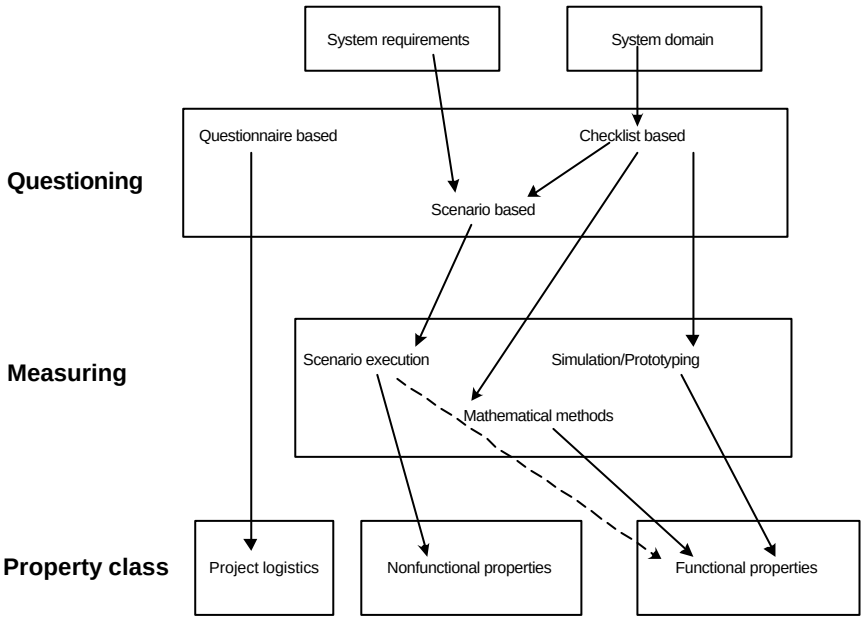

Figure 10. Schematic picture of the relations between the evaluation techniques

Although measuring techniques might give quantitative values, these values must be treated carefully. The quantitative values should be used as relative values when comparing competing software architectures. Moreover, if scenarios or experienced reasoning was used to obtain the values, the exact same set of scenarios and reasoning must be used when evaluating the competing or refined architecture. Otherwise, the measures are not comparable. Consequently, it is impossible to compare measured quality of a software architecture across the application domain i.e. within the same class of products but in different environments or applications.

4.2 Functional analysis

There exist functional quality properties in abundance, among which the properties of particular interest when designing safety-critical real-time system is listed in Table 1.

Performance	The systems capacity of handling data or events.
Reliability	The probability of a system functioning correctly over a given period of time
Safety	The property of the system that it will not endanger human life or the environment
Security	The ability of a software system to resist malicious intended actions
Availability	The probability of a system functioning correctly at any given time
Temporal constraints	Real-time attributes such as deadlines, jitter, response time, worst case execution times (wcet), etc.

Table 1. Functional quality properties

Performance

Certain functional properties of a software system are tricky or even impossible to predict using the architectural description level only, e.g. performance. Performance estimations must have the algorithmic solutions as input. As discussed in the introduction, software architecture is a description of the system on a higher level of abstraction than algorithmic solutions and data structures. However, by using prototyping and simulation techniques, performance in terms of ,for instance, event throughput or queuing length for events in a system, can be estimated [GRBO]. Since such a performance measure is not absolute, it can only be used when comparing two different architectural solutions, not when estimating, for instance, the worst execution time for handling an event in the system.

Reliability

There are mathematical methods based on probability theories such as Markov models for assessing reliability [Tram95]. However, these theories are developed for hardware where failures often are caused by physical wear such as corrosion, overheating, etc. Such failures are probabilistic in nature whereas software failures are mistakes (errors), made in the

specification, the design or in the implementation. These types of failures are certainly not probabilistic according to some distribution over time. Furthermore, software can never be worn out. Attempts have been made to apply the methods from the hardware community to software. In software, the statistics are the numbers of errors in the program or the likelihood of a failure in a point of time based upon the error distribution in the past [Fenton96]. To get such failure estimations, there must be an implementation of the application or at least a prototype. Anyhow, a description of the application on a lower level than the architecture is needed. With some heuristics from similar applications developed earlier experienced engineers can estimate the expected number of errors in the components. Such estimations are very complex, giving rough metrics. An alternative to directly measure the reliability of the architecture is to measure the testability. The testability is a function of the effort required in order to assure the required level of reliability or availability.

There are three different approaches to handle faults in order to achieve a reliable system [Lapr92]:

- Fault avoidance
- Fault removal
- Fault tolerance

Fault avoidance is about designing error free systems. This implies the use of structured design methodologies such as formal methods or semi-formal methods. Formal methods are based on mathematical models of the software system and the requirement specification. These models form the basis when proving correctness of the model with respect to the system specification. There exists a wide area of formal methods and formal modeling languages, each supporting different system domains. Semi-formal methods are, as the name suggests, less formal, i.e. they do not support techniques to exhaustively prove correctness of the models. Instead, they offer a structured way of reasoning, both when designing models of the system and when analyzing the models. The methods are usually based on some "formal" notation, e.g. *Unified Modeling Language* (UML)[BRJ98], ADLs, etc., representing the system model. Examples of such methods are *object-oriented analysis and design* (OOA/OOD), and software architecture techniques in general.

No matter how accurate the models are analyzed, there may still be errors in the implementation. These errors usually originate from the specification and from the mismatch when mapping the models to the source code. In order to improve reliability in the program, *fault removal* techniques can be applied. Fault removal is basically the task of finding the errors by testing and removal of them by error correction. Under the assumption that no new errors are introduced, the reliability will grow as errors are corrected. This assumption is, unfortunately, seldom true, implying that the whole system has to be re-tested after each increment. The results from testing and re-testing can be used for statistically forecasting of the failure rate (and consequently the reliability), of a software system. Such a method is the *reliability growth model*, first proposed for software by Jelinsky et. al. [JEMO72]. There exist an abundance of different approaches to model reliability growth; they are all based on data collected during testing, but differ in the way the statistical model is made.

Some faults are impossible to avoid regardless of how accurate the design and the tests are performed. If it is particularly important that a certain module in the system does not fail, fault-tolerance can be introduced. *Fault-tolerance* is a technique which can be interpreted in two different ways: it could be the ability of a software system to tolerate faults from its environment, e.g. the operator, hardware errors, etc., or it could mean that the system should be tolerant against design faults in the software itself. The two different fault-tolerance approaches are, naturally, solved using different techniques. For instance, to be fault-tolerant against hardware errors such as electromagnetic distortion, redundant hardware can be used, each with equivalent software running on them. This solution will however not tolerate software faults. Different approaches to be tolerant against software faults are *recovery blocks* and *N-version programming* [Storey96][CA78].

Recovery blocks are based on acceptant tests of the calculated values. If the processed value is not accepted the program tracks back to a recovery point where it is safe to continue the execution after having restored the system's state.

N-version programming is achieved by developing *N* different versions of the software; each developed by different and isolated design teams. All *N* different versions run in parallel at runtime and their respective results are voted upon. This technique has, however, been proven not so successful since all different versions of the software start out from the same specification, and since most design errors originate from the specification, they will contain common errors.

Even if the source code is absolutely correct, the compiler may still produce erroneous binaries. Faults introduced by the compiler can be tolerated by using the N-version approach. Each version has exactly the same code, but they are all compiled using different compilers.

It is important to note that the different techniques discussed above can be applied at any stage in the development process. For instance fault removal can be used when verifying the designed architecture against the system specification. Fault-tolerance is also a matter of architectural design. The techniques for fault-tolerance discussed above are all achieved using different architectural solutions.

Safety

Safety seems, at a first glance, very similar to reliability. There is however a clear distinction as safety is only concerned with failures that endangers human life and the environment, i.e. hazards, whereas reliability deals with all failures regardless of their consequences. However, before any safety analysis of the architecture can be performed, the hazards must be identified. This is done in a *hazard analysis* that is a reasoning based method for finding all hazards in the system that is going to be designed [Leve95].

There exist several techniques for assessing safety properties in software designs. Most of them are scenario based and work either backward or forward. If the method works backwards, the analysis starts with the hazard as a scenario, trying to trace down the responsible component. On the contrary, if the method works forward, the effects of an error in a component is investigated.

Some of the most well known forward methods are Failure Mode and Effects Analysis (FMEA) and Hazard and Operability studies (HAZOP). Both methods analyze the consequences of failures in the components. One commonly used backward technique is called Fault Tree Analysis (FTA)[Storey96]. FTA starts with a hazard, trying to determine its origin among the components. This kind of analyses give an understanding of where in the architecture fault-tolerance techniques should be introduced, or if already introduced, verifying whether the intended fault-tolerance is achieved or not.

Depending on the results from the safety analysis, changes in the design may have to be performed. Different design approaches to avoid catastrophic failures can be applied based on the severity of an accident caused by the hazard. The different approaches are [Leve95]:

- Hazard elimination
- Hazard reduction
- Hazard control
- Damage minimization

The severity is a quantified value that makes it possible to compare and rank hazards. Typically, the severity is given in terms of the cost or, lost lives, for the stakeholder if the accident occurs.

Substitution, *decoupling*, and *simplifications* achieve *hazard elimination*. By substitute a dangerous design possibility by a functionally equivalent, but not dangerous solution, the hazard itself is eliminated. For instance, if the system involves a very toxic chemical liquid, substituting the liquid with a non-toxic one eliminates the hazard. Moreover, by decoupling safety-critical parts of the software from non-critical software, the risk for an error in the non-critical part to propagate into the safety-critical parts is eliminated. There exist some known architectural solutions based on decoupling, e.g. safety kernels, firewalls, hierarchical architectures [Storey96].

Hazard reduction reduces the likelihood of the occurrence of a hazard. It might not be feasible or even possible, to eliminate the hazards. Then the designer has to design the system in such a way that the hazard is not very likely to occur. An example of hazard reduction is to erect a fence around an industrial robot, preventing humans to come close enough in order to get hurt.

Hazard control is applied in order to reduce the likelihood of an accident if a hazard arises. This can be achieved using *fail-safe design*, i.e. the system should be designed to detect the hazard and then transfer it into a safe state if such exists. There are, however systems where no safe state exists. A typically example of such a system is airplanes. These systems must keep operating even if something goes wrong. This is achieved using fault-tolerance such as *redundancy*. It is essential that an airplane keeps flying even if one engine breaks down by using the second engine. The performance will of course be reduced, but the airplane can still be maneuvered to its safe state on the ground.

Yet, if an accident still occurs, the consequences and losses must be reduced. This is achieved with damage minimization that strives to minimize the exposure of the accident to the environment or human beings.

Availability

Reliability and availability are strongly correlated. According to the definitions given in Table 1, reliability is the probability of a software system functioning correctly over a given *period* of time and availability is the probability of a software system functioning correctly at *any* given time. More generally, reliability is equivalent to Mean-Time-Between-Failure (MTBF) and the availability is a percentage figure given by the formula below:

$$Availability = 1 - \frac{MTTR}{MTBF}$$

MTTR is an abbreviation for *Mean-Time-To-Repair*, i.e. time spent on service. The relation is shown graphically in Figure 11 below. If any point of time is picked randomly along the y-axis, there is a probability of having correct functionality, i.e. the availability of the software system.

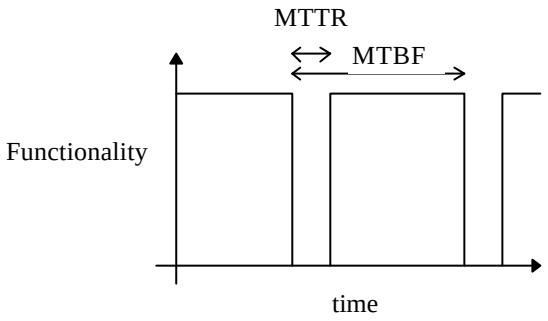

Figure 11. Availability and reliability

Security

Security is concerned with protecting a software system from malicious intended actions, e.g. intrusion by unauthorized users or locking out unintended accesses to safety-critical parts of the system. This can be achieved by different architectural solutions: safety/security kernels, firewalls, etc. which all are different ways of restricting the access to the system or sub-systems. As security can be achieved using different architectural solutions, it can be assumed that security is assessable by architectural analysis. A scenario-based method can be used. Typically, such a scenario could reason about what happens if an operator or a sub-module tries to access a protected region of the system. Another possible way of analyzing software architectures from the security point of view, is simulation, provided that the logical view of the software architecture contains sufficient information regarding rules for authorization and identification.

Real-time requirements

When designing real-time systems it is important to ensure the temporal correctness of tasks in the application. The timing must be just perfect, neither too fast nor too slow. The

information necessary for the verification of temporal constraints is provided by the temporal view of the architecture. A typical example of such an analysis are schedulability test, i.e. analyzing whether the task set is schedulable or not given the resources and temporal constraints given as release times, deadlines, worst case execution times (wcet), jitter, etc. The resources taken into account when analyzing the schedulability of a system are typically CPUs, communication busses, actuators, etc.

There exist a lot of mathematical methods for verifying the temporal behavior of a real-time system, all having different assumptions on the scheduling strategy and the task model [LILA73][ABDTW95]. A task model defines the temporal requirements put upon a task, i.e. priorities, period times, etc. The task model and the scheduling strategy is strongly coupled since the task model provide the input to the schedulability analysis.

In Figure 12, a classification of different scheduling strategies is illustrated.

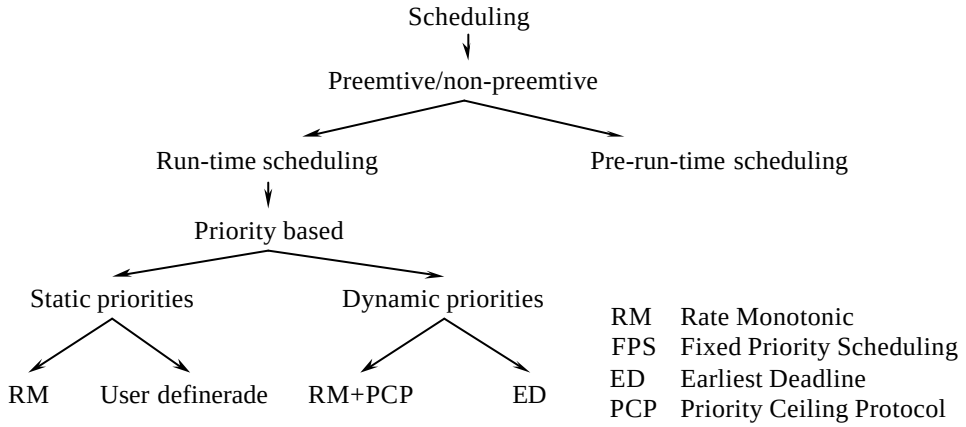

Figure 12. Classification of scheduling strategies.

4.3 Nonfunctional analysis

The number of nonfunctional quality properties is, as the functional quality properties in the previous chapter, very large. In Table 2, a subset of all such quality properties is listed, all being important in a mature and modern design process for real-time systems.

Cost	The cost for performing any action such as development, evolution and verification
Testability	How easy it is to prove correctness of the system by testing
Reusability	The extent to which the architecture can be reused
Portability	How easy it is to move the software system to a different hardware- and/or software platform
Maintainability	The aptitude of a system to undergo repair and evolution
Modifiability	How sensible the architecture is to changes in one or several components

Table 2. Nonfunctional quality properties

A very simple but yet powerful method for analysis of nonfunctional quality properties is execution of scenarios. Several of the direct and indirect quality properties listed in Table 2 can be examined and analyzed by using scenarios. By direct we mean an attribute that focus on the software only such as the reusability of a module or subsystem or the portability i.e. how easy or hard it is to move the system to another operating system or hardware platform. An indirect property is one that depends on a direct one. A typical example is the cost. The cost is always related to the action, for instance the cost associated with testing, development, maintenance, etc.

Cost

As discussed above, cost is an indirect quality property, always depending on other quality properties of the system. Typically, after a system has been released and been running for a while, new functionality is required from the customer or new features and improvements are desired within the organization. Then the cost is probably dependent on the reusability, maintainability and testability of the software. Cost estimations are probably one of the hardest tasks for every development project. The cost estimation for the design of a completely new system is extremely hard to achieve. Usually such estimations are based only upon historical experiences with similar systems. If no such experience is available, the estimation gets even more imprecise. The software architecture description could help illuminate the cost of developing a system or adding new functionality to an existing system. Partly by being a structured description of the application, helping the designer to get a full perspective of the application scope, but also by providing techniques for analyzing the effects of adding new features to an existing software system.

Testability

Testing is essential in order to prove functional correctness of a software system. It is also used for obtaining some confidence in functional quality properties such as reliability, performance, etc. A lot of time and consequently, money is spent in the testing phase of software development. To reduce the amount of time needed for testing of the software, the architecture must be designed so that it is easy to test, i.e. having high testability. The testability is dependent on three individual properties: *observability*, *controllability*, and for concurrent systems and systems dependent on time, *reproducibility* [Bind94]. Testability is consequently an indirect quality property as well.

In order for a test case to be useful, the result of it must be observed. If the components in the architecture are seen as “black boxes”, i.e. the structural view, only the interfaces are observable. The bigger interface, the more visibility. Apparently, bigger interfaces give higher *observability*, thus higher testability.

When performing a test, a particular input is given to the system or a sub-system. This input is the only way in which the test engineer can control the path taken in the program. If the path taken only depends on the input itself, maximum controllability is achieved. This is of course not the case in general. There are often data dependencies between different modules such as global variables etc. If those data dependencies, which are not controllable

by the test input data, affect the control flow, the controllability is decreased, giving lower testability.

Finally, when testing concurrent system or real-time systems in general, the order in which different processes in the system are executed will influence the observed result from a test. For instance, in a system controlling the water level in a tank, there is one process sampling the actual water level and one process calculating how to adjust the water level based on the measured value and some set value. If the control process executes twice without any intermediate execution of the sampling process, the result of the control decision will be different in the second invocation than if the water level was re-sampled in between. To get high testability, the order in which processes execute must be controllable or deterministic, i.e. high reproducibility [ThHa99].

Reusability

Reusing a software component to its full extent, without any modifications, is extremely difficult if not the domain in which the reuse is intended is the exact domain of the component origin. When a component or architecture is reused in the same application domain we call it a domain-dependent reuse. When containers are reused, i.e. lists, arrays, sets, etc., they can be reused across different application domains. An example of such reuse is the Standard Template Library (STL) for the object-oriented language C++. Reuse, which is possible across the application domains, is consequently called domain-independent reuse.

When analyzing the level of reusability of a component or a part of the architecture, one must consider not only the original application domain, but also how isolated and independent it is from rest of the system. The less dependencies, the more reusable, and vice versa.

The focus on reuse, in industry, has been intensified due to the potential cuts of cost. The time spent on implementation decreases when reusing components. Furthermore, components can be bought from third-party developers. Such components are called Commercial-Off-The-Shelf components (COTS).

Portability

To be able to analyze software architectures with respect to portability, the platform on which the system is going to run on has to be modeled as well. This to unveil the dependencies between the software components in the system and the *platform*. As platform we consider the hardware, e.g. processors, A/D converters, as well as the software providing the infrastructure e.g. operating systems. If the amount of direct dependencies, i.e. the number of components having a direct connection to the platform, is low, then the architecture as whole is quite insensible to a change of platform. Thus, having a high degree of portability.

Maintainability

Kazman et. al. [KAC96], have proposed a methodology for visualizing the amount of changes required in the modules or in the architecture when adding or changing functionality in the system. The amount of changes in the software architecture enforced by adding new functionality or error corrections, are referred to as *maintainability*. By using scenarios developed from the requirements of the new function, the existing architecture is analyzed.

The concept, direct scenarios, were introduced meaning scenarios that are directly supported by the existing architecture i.e. no major architectural changes are required. In opposite, an indirect scenario exposes the need for architectural changes, which is more difficult and costly to achieve. Remember that there is a difference between a direct or indirect scenario and the direct and indirect quality properties introduced earlier in this chapter. After having mapped the scenarios on the architectural structure and determined if the scenario is direct or indirect, scenario interaction should be revealed. Two or more indirect scenarios are said to interact if they affect the same module.

To make the potential architectural violations and changes in the system visible, graphical representation of modules were scaled in the ADL according to the amount of indirect scenario interactions.

5 Architectural design

Architectural analysis can, and should, be used as guidance when designing a software system. A software system can be implemented in several ways, all having different architectural solutions. By using architectural analysis, the architecture that fulfills the requirements best can be chosen. The workflow for designing architectures for a system is shown in Figure 13.

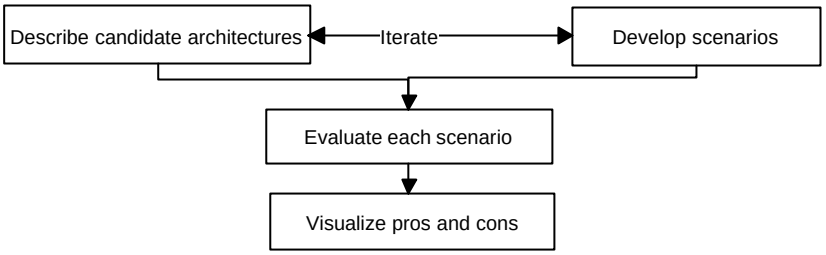

Figure 13. Architecture development and analysis process.

The first phase when developing a software system is to develop candidate architectures and a set of scenarios that reflects the requirements on the system. The number of scenarios to develop is related to the generation of ordinary test cases. Eventually, a state is reached where the added value of a new scenario is less then the effort required to develop the scenario itself. When this point in time is reached or when the development budget is violated, the scenario generation should stop.

Now we have the candidate architecture and a set of scenarios. By executing the scenarios on the architecture a table with the desired quality attributes can be constructed. In the table, all requirements are marked with plus signs and minus signs representing how well the architecture fulfills the requirements. If the result from the analysis is satisfactory, the next phase is to do low-level design and implementation. However, if the analysis results are not satisfactory, an alternative architecture must be developed on which exactly the same scenarios are executed. Consequently, the evaluation must be done all over again. The work of finding a sufficient architecture is highly iterative, meaning that the architecture can evolve by small steps until a reasonable solution is found. Consequently, changes suggested by the analysis may result in a complete redesign using a completely different architectural style or minor modifications in subsystems only.

The table produced in the analysis phase containing all the analyzed quality properties constitutes the input to a *tradeoff analysis*. In a tradeoff analysis the set of competing architectures is compared or the result from a refined architectural solution is compared with the result from the analysis of the preceding generation of the architecture. The objective of the tradeoff analysis is to choose the architectural alternative that best complies with the ranking among the quality properties.

A method for tradeoff analysis called *Architecture Tradeoff Analysis Method* (ATA) has been developed at the Software engineering institute (SEI) at Canegie Mellon university [Kazm98]. It is an iterative development method that is similar to the process shown in Figure 13.

A method called *Software Architecture Analysis Method* (SAAM) is also developed at SEI. The purpose of SAAM is to analyze software quality attributes by examining competing architectures [KBAW94]. To do so, they partitioning the functionality in the architecture i.e. identifies were in the different architectures the functionality of the system is allocated. The functional partitioning is system domain specific. Some domains already have a well-defined functional partitioning; a typical example of such a domain is compilers. Compilers are built with a front-end, a parser, a code generator etc. However, nothing is assumed about how functions are organized and structured, i.e. the architecture of the compiler. This partitioning gives a common description and common modules, each with the same functionality but organized in different ways. The communal description is an absolute condition for the comparison, which aims to unveil how well a certain quality attribute, is adopted by the architecture. Again, the analysis is based on scenarios, constructing input for a tradeoff analysis.

5.1 An example

As an example of how an architecture is constructed, analyzed and transformed in order to better comply with the requirements consider a real-time system that controls the water level in a tank. The system samples a water level sensor, takes a decision whether to let water out, or pour water into the tank. The system actuates a pump or a valve if the level has to be adjusted. As it is a real-time system, the temporal constraints on the system must be fulfilled, i.e. there is a functional quality requirement on timing. Moreover, the system should easily be modified to run on different platforms (real-time operating system and hardware), i.e. portability.

First, the structural view of the architecture is developed, identifying the components in the system and their interconnections. In this case, the interconnections represents transportation of data among the task using services provided by the RTOS. While portability is crucial, the operating system and the hardware view is modeled as well. The first candidate architecture is shown in Figure 14.

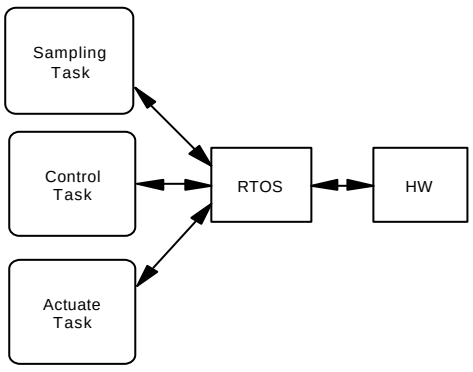

Figure 14. The first candidate architecture for a water tank controller

Next the compliance between the architecture and the required quality properties must be analyzed. Verifying the temporal behavior requires the temporal view of the architecture. For

this particular application, the period time, the estimated worst execution time (wcet), and the deadlines for the three tasks is shown in Table 3.

Task	Period time (T)	wcet (C)	Deadline (D)
Sampling task	1 ms	50 µs	60 µs
Control task	2 ms	200 µs	1 ms
Actuate task	2 ms	50 µs	1 ms

Table 3. The temporal view

The temporal behavior is verified using *exact analysis* where the worst case response time for all tasks is calculated. If the response times are less than the specified deadlines for all tasks, the system is schedulable [JOPA86]. Exact analysis requires priorities to be assigned to the tasks. In this particular example, priorities are assigned according to the *rate monotonic* algorithm where the task with the shortest period gets highest priority [LILA73]. Rate monotonic gives the sampling task high priority, the control task medium priority and the actuating task low priority. The exact analysis formula is recursive and calculates the worst case response time with respect to interference of the execution of tasks with higher priorities. The recursion stops when two subsequent calculations result in the same response time, i.e. a fix-point is reached. The formula is shown below:

$$R_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \qquad \forall j \in hp(i)$$ Denotes all tasks j with higher priority than task i.

The response times for the sampling task is 50 µs as no other task interferes with it since it has the highest priority. The response time for the control task is 250 µs. Finally, the actuate task has a response time of 300 µs. If the calculated response times are compared to the specified deadlines, it could easily be verified that the system is schedulable as the response times for all tasks are less than corresponding deadlines.

To assess portability, scenarios can be used. For the matter of simplicity, only one scenario is used in this example, namely: *"Move the system to another platform"*. The idea is to execute this scenario on the proposed software architecture to estimate the number of component being subjects to changes. As portability is the issue, the number of affected components should be held to a minimum. In the architecture suggested in Figure 14, all the components interact with the real-time operating system. Consequently, there are a lot of platform specific system calls embedded in each and every component, giving poor portability since every component has to be changed as a result of a changed platform. To increase the portability, architectural transformations have to be performed, i.e. the software architecture has to be refined. One possible transformation is to introduce a *proxy-component* between the task components and the real-time operating system. This transformation is shown in Figure 15.

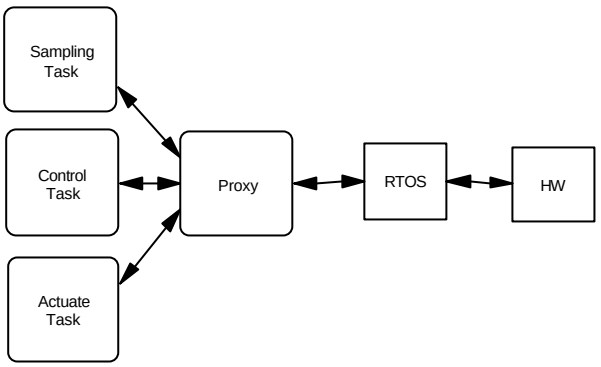

Figure 15. The architecture after the transformation

The proxy provides the tasks with all necessary services in order for them to perform their intended tasks, while hiding the actual system calls. To verify the new architecture according to the requirements, the scenario has to be re-executed. Now the proxy component is the only one affected by a changed platform, i.e. a maximal portability is achieved. However, the portability is achieved at the expense of an increased overhead for system calls. Therefore, the worst case execution times for the individual task components must be re-estimated and the exact analysis must be done all over again to verify the temporal behavior of the system. The phenomena that quality properties might affect each other in a negative manner, is referred to as *tradeoff*.

6 Conclusions

Software architecture is part of what generally is referred to as software engineering. Software engineering also includes a lot of other techniques like software metrics, formal methods, test methodologies, etc. Thus, software engineering is an umbrella for all techniques and methods needed to establish a ”science of engineering” practice in the software community. Software architectures are an important part of software engineering since it deals with high-level modeling and evaluation. The software architecture community is still very young, but the recent interests from the industry have launched a lot of research activities in academia. Especially relevant are the software architecture analysis methods as the analysis provides the information for early design decisions.

To make architectural analysis possible, the architecture must be described in a language with well-defined semantics. A language that describes software architectures is called Architectural Description Language (ADL). There exists a lot of different ADL:s, but few of them have received any particular attention since it is very difficult to design a language with syntax and semantics powerful enough to cover all possible application domains and that can be interpreted by all stakeholders in a project. As a consequence, software developers use their own description languages. An important property of an ADL is the architectural views, providing detailed information needed for the analysis. The number of views and the contents of each view will vary between different application domains and the required analyses. Finally, a description language with a well-defined semantics is also a necessary condition for developing tools that support architectural development and evaluation.

This report has described existing techniques for describing and evaluating software designs based on information mainly provided by the high level description, i.e. the software architecture. The ability to evaluate early design decisions is very important since early design decisions are crucial for the final result, both regarding correct functionality and cost. The earlier design mistakes are detected, the less time has to be spent on redesign. The properties analyzed using software architectures are called quality properties. In this survey, the quality properties are divided into two separate classes, functional and nonfunctional. Functional quality properties are concerned with the run-time behavior of the software system, for instance performance and reliability. In contrast, nonfunctional quality properties are concerned with the quality of the software itself. Examples of nonfunctional properties are reusability, maintainability, and testability.

Tool support for architectural development and evaluation is poor. It is possible to formalize knowledge in frameworks, guiding the designer in both architectural transformations and in the tradeoff analysis. There exist tools for some of the analyses, for instance tools for verifying the temporal behavior in a real-time system [ERGUSA97], but these tools are still islands in the ocean called software engineering. We need to discover, or build new islands and connect them to each other in order to get complete suits of tools, supporting the complete software development- and maintenance process. In mature engineering disciplines, such tool support is taken for granted. Software engineering tools will probably appear as the software community gets more mature, it is still very young, at least when compared to other traditional engineering disciplines.

7 References

[ABDTW95] N. C. Audsley, A. Burns, R. I. Davis, K. Tindell, and A. J. Wellings, Fixed Priority Pre-emptive Sceduling: An Historical Perspective, Real-Time Systems 8(2-3):173-198, 1995

[ALDI92] R. Alur, and D. L. Dill, A theory of timed automata, 1992

[BCK98] L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, Addison Wesley 1998

[Beng97] PO. Bengtsson, and J. Bosch, Scenario-based Software Architecture Reengineering, University of Karlskrona/Ronneby 1998

[Bind94] R. V. Binder, Design for Testability in Object-Oriented Systems, Communications of the ACM, Volume 37, No 9, pp. 87-101, 1994

[BRJ98] G. Booch, J. Rumbaugh, and I. Jacobson, The Unified Modeling Language User Guide, Addison Wesley ISBN 0-201-57168-4, 1998

[BuWe94] A. Burns, and A. Wellings, HRT-HOOD, a Structured Design Method for Hard Real-Time Systems, 1994

[CA78] L. Chen, and A. Avizienis, N-version programming: a Fault Tolerant Approach to Reliability of Software Operation, In proceedings of 8th Annual International Conference on Fault Tolerant Computing, pp. 3-9, 1978

[Clem96b] P. C. Clements, and L. M. Northrop, Software Architecture: An Executive Overview, Technical report CMU/SEI-96-TR-003 1996

[Clem96a] P. C. Clements, Coming Attractions in Software Architecture, Technical report CMU/SEI-96-TR-003 1996

[DaYo95] C. Daws, and S. Yovine, Two examples of verification of multirate timed automata with KRONOS, In proceedings of 16th IEEE Real-Time Systems Symposium, PP 66-77, 1995

[dijk68] E. W. Dijkstra, The Structure of ”THE”-Multiprogramming System, ACM on Operating System Principles 1967

[EHLS94] S. Edwards, W Heym, T. Long, M. Sitarman, and B. Weide, Specifying Components in RESOLVE, Software Engineering Notes, vol. 19, no. 4, 194

[ERGUSA97] K. Sandström, C. Eriksson, and M. Gustafsson, RealTimeTalk - a Design Framework for Real Time Systems - a Case Study, SNART 1997

[Fenton96] N.E. Fenton, and S. Lawrence Pfleeger, Software Metrics, International Thomson Computer Press 1996

[Garl93] D. Garlan, and M. Shaw, An Introduction to Software Architecture, Advances in Software Engineeri Vol 1 World Scientific Publishing Company 1993

[GHJV94] E. Gamma, R. Helm, R. Johanson, and J. Vlissides, Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley 1994

[GRBO] H. Grahn, and J. Bosch, A Simulation Approach to Predict and Evaluate the Performance of Software Architectures, University of Karlskrona/Ronneby 1998

[JEMO72] Z. Jelinsky, and B.P. Moranda, Software Reliability Research, Statistical Computer Performance Evaluation, pp 465-484, New York , SA, Academic Press, 1972

[JOPA86] M. Joseph, and P. Pandya, Finding Response Times in a Real-Time System, The Computer Journal, Volume 29, No. 5, pp. 390-395, 1986

[KAC96] R. Kazman, G. Abowd, L. Bass, and P. Clements, Scenario-Based Analysis of Software Architecture, IEEE Software 1996

[KBAW94] R. Kazman, L. Bass, G. Abowd, and M. Webb, SAAM: A Method for Analyzing the Properties of Software Engineering, Int. Conf. On Software Engineering IEEE Computer Science Press pp. 81-90, 1994

[Kazm98] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, The Architecture Tradeoff Analysis Method, Submitted to the 1998 International Conference on Software Engineering

[LaLe94] P. B. Ladkin, and S. Leue, What Do Message Sequence Charts Mean?, IFIP-Transactions-C:-Communication-Systems.n C-22, pp 301-316, 1994

[Lapr92] J.C. Laprie, Dependability: Basic Concepts and Associated Terminology, Dependable Computing and Fault-Tolerant Systems, vol. 5, Springer Verlag, 1992

[Leve95] N.G. Leveson, Safeware, System Safety and Computers, Addison Wesley 1995

[LILA73] C. L. Liu, and J. W. Layland, Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment, Journal of ACM, Volume 20, Nr. 1, pp. 46-61, 1973

[LKAV93] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann, Specification and Analysis of System Architecture Using Rapide, Stanford University technical report, 1993

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi, Uppaal in a Nutshell, In Springer International Journal of Software Tools for Technology Transfer 1(1+2), 1997

[Masc87] The official handbook og MASCOT, Version 3.1, Issue 1, 1987

[Miln87] R. Milner, Communication and Concurrency, Prentice Hall 1989

[Paul94] F. Paulisch, Software Architecture and Reuse – An Inherent Conflict?, Proceedings of 3rd International Conference on Software Reuse, pp 214, 1994

[SHGA96] M. Shaw, and D. Garlan, Software Architecture - Perspective on an emerging Disipline, Prentice Hall 1996

[Storey96] Neil Storey, Safety-Critical Computer Systems, Addison-Wesley 1996

[ThHa99] H. Thane, and H. Hansson, Towards Systematic Testing of Distributed Real-Time Systems, In proceedings of the 20th IEEE Real-Time Systems Symposium, 1999

[Tram95] C. Trammell, Quantifying the reliability of software: statistical testing based on a usage model, In 'Experience and Practice', Proceedings., Second International IEEE Software Engineering Standards Symposium,. , pp. 208 –218, 1995

[Vest94] S. Vestal, Mode Changes in a Real-Time Architecture Description Language, Proceedings of 2nd International Workshop on Configurable Distributed Systems, 1994, Page(s):136-146

Appendix A - Terminology

ADL - Architectural Description Language, Language for describing software architectures

Architectural style - Standard types of architectures identified with names and patterns

Architectural view - Provide the architecture description with information needed when analyzing it. The components and their interconnections are shown in the structural view.

Architectural transformation – Changing the architecture in order to obtain required functionality and quality

Availability - The probability of a system functioning correctly at any given time

Checklist based questions – Domain specific questions used when evaluating a software architecture

Cost - The cost for performing any action such as development , evolution and verification

COTS - Commercial Off The Shelf components

Design patterns - Named object oriented solutions in the object oriented community

Design space - A N-dimensional space where every axis represents a design parameter, scaled with the different design options possible for that particular parameter

Direct scenario - A scenario that is directly supported by the architecture

Fault-tolerance - The ability of software to detect and tolerate errors in the design and/or from its environment

Framework - An architectural pattern for a particular domain, widely used in the object oriented community.

Functional quality property – Quality properties concerned with the run-time behavior of the software system

Indirect scenario – A scenario that requires an architectural transformation to be supported by the architecture

Maintainability - The aptitude of a system to undergo repair and evolution

Modifiability - How sensible the architecture is to changes in one or several components

MTBF – Mean-Time-Between-Failure

MTTR – Mean-Time-To-Repair

Nonfunctional quality property - Quality properties concerned with the software itself

Performance - How fast or slow the system performs its functions measured in time or the systems capacity measured in event-throughput

Portability - How easy it is to move the software system to a different hardware- and/or software platform

Reference style - Architectural styles widely used in particular application domains, e.g. the pipe-and-filter Architecture used in compilers.

Reliability - The probability of a system functioning correctly over a given period of time

Reusability - The extent to which the architecture can be reused

Safety - The property of the system that it will not endanger human life or the environment

Scenario based questions – Application specific questions used when evaluating a software architecture

Scenario execution - Method for analyzing an architecture by asking "what if" questions

Security - The ability of a software system to resist malicious intended actions

Temporal constraints - Real-time attributes such as deadlines, jitter, response time, worst case execution times (wcet), etc

Testability - How easy it is to prove correctness of the system by testing

Tradeoff - A relation between two or more quality attributes where an increased level of on property results in a decrease of another property.

Questionnaire based evaluation – Questions used when evaluating project logistic properties of software architectures