System Architecture in a Mechatronics Perspective

De-jiu Chen
Mechatronics Lab
Machine Design, KTH
chen@damek.kth.se

Martin Törngren
Mechatronics Lab
Machine Design, KTH
martin@damek.kth.se

Abstract
System architecture for mechatronical systems is treated in ongoing research at the Mechatronics Lab at KTH. This paper provides a survey of research on architecture as a basis for further work. The conclusion is clear; there is a strong need to adopt the concept of architecture to overcome problems in current system development and to exploit the potential in architecture based development in order to better manage the design of complex systems. To clarify and define the scope of this concept, a definition of the term "system architecture" is elaborated. Then, the paper discusses some useful concepts in architecture based system development. At last, it introduces some design principles and usable methods for analyzing the architectural quality attributes with the focus on the software.

1 Introduction

The mechatronics perspective applied in this paper corresponds to a multidisciplinary one and is motivated by the dramatic increase in functionality incorporated through software in embedded systems in machinery, such as road or rail vehicles. The traditional subsystem based approaches in development and in the organization of companies, compare mechanics, software, hardware and control, are no longer sufficient in the development of highly integrated and complex computer control systems, [Wikander&Törngren98]. A particular set of functions, for example vehicle dynamics for cars, have to be designed and developed in tight integration with the mechanical system, its sensors and actuators. Today we find cost-efficient implementations of the functions in terms of software mapped to distributed computer systems. A node of such a system, its hardware and software, is typically part of performing a number of system functions. I.e., it is not sufficient to consider software, mechanics, hardware and control in isolation. In consequence, an interdisciplinary perspective to architecture is also required.

Developing and constructing the control system for machinery can be a complicated and formidable challenge for engineers. It is required that the system should not only provide logically and timely correct interactions with the environment to assure the system functionality, but also be dependable, modifiable, reusable and portable. All these requirements have complicated and interwoven interrelationships with each other because of the strong interdependencies among the system elements. It is thus a complex problem to clarify, balance and fit all these requirements and to make clear and correct design decisions. System integration is another inherent challenge in developing the complex system. The abstract medium, software, is adopted to implement a major portion of system functions, mainly because of its flexibility and the falling cost-performance ratio of the hardware. When increasing system functions are moving from mechanical system into the software system, the success of the software system design becomes more critical. In fact, an improper usage of the software medium does increase further the complexity by introducing much more elements and much larger "state space" in the system. Hence, the system can become much more difficult to be correctly designed, verified, tested, upgraded and even understood.

Traditionally, a system development process begins with requirement acquisition and ends with system delivery. Three major steps can be identified in the process [Gajski et. al. 94]. First, the desired functionality is defined and described. Second, the specified functionality is partitioned, and then scheduled and allocated to standard or custom components for more detailed component design. At last, each component's functionality is implemented as hardware or software. No matter what development process model or life cycle model is utilized, assuring the system quality by only emphasizing the final verification and test leads to late found design problems and costly iterations to remedy the problem. For instance, even

though some kind of verification and test can be carried out at the same time as the progress of the design, it may be impossible to find some design problems until it is too late. In fact, modifications are unavoidable in the design process. For example, when the users/clients have acquired a better understanding of the system as the design goes on, the older designs may have to be modified for the newer requirements. It is also the case when a design inconsistency with the requirements having been revealed by the designers. If eventual modifications and new design choices are made in an unconstrained or *ad hoc* manner, new design faults or inconsistencies can be introduced in the system. As a matter of fact, the earlier in the development process, the easier to make change. However, to reveal the necessities earlier and make correct changes, perspective and insight are required. In addition, "traditional" life cycle models do not cover the maintenance and upgrade aspects of the system explicitly.

Related to the need of emphasizing early design phases for the critical early design decisions and keeping integrity during the entire life cycle, it is general agreed that a focus on higher levels of abstraction is necessary. Therefore, the concept of architecture should be adopted in the design of the complex systems.

The next section introduces a definition of "system architecture" according to the Oxford dictionary. Section 3 discusses some concepts of the architectural description, including views, models and components. Section 4 describes the architectural styles. Section 5 introduces some usable techniques for architecture-based system development.

2 Architecture - What is it?

Based on the original meanings of the term "architecture" [Oxford95], the word "system architecture" could be defined as: *the art and science of designing and constructing systems; the design or style of a system or systems; the nature and structure of a particular system that determines the way it operates*.

The first definition indicates that system architecture is about the design issues; in addition, it points out both science- and practice-based methodologies should be implemented to delimit and guide the system design. The second definition suggests that the architecture exhibit also style(s), which is a distinctive manner of doing, performing or presenting a system or a set of systems [Oxford95]. The third definition points out that the system architecture is about the behavioral and structural issues of a particular problem solution. Especially, it is about the organization of components and their interrelationships in terms of high level design solutions.

We believe that these three meanings complement each other and indicate the entire scope of "architecture" and "architecting". That is, while the first and the second meanings give architecture's general features, the last meaning addresses its property for a particular problem solution. Incorporating architecture in the system development process gives a set of benefits. Some of them can be stated as the below.

- Architecture promotes an interdisciplinary design concept [Törngren&Redell99] [Wikander&Törngren 98]. By means of high-level solution properties, it provides the possibilities for analyzing and reasoning about the upcoming implementation characteristics. These quality predictions help to reveal the interwoven dependencies in the system caused by the strong integration of technologies in the domains of control, mechanics, software and hardware. Critical design decisions, e.g., partitioning functionality and corresponding implementation technology, can thus be handled systematically early in the design process (i.e., "top-down").
- Architecture facilitates the requirements acquisition and refinement. By explicitly expressing the satisfactions of given requirements on high-level (initial) solutions, it gives early opportunities for all stakeholders to reason about the feasibility or reasonability of the requirements and potentially reveal hidden ones as well. Eventual modifications can thus be carried out in a cost-effective way.
- Architecture provides a means to assure the design integrity. Since a chosen high-level solution structure also defines the corresponding design space, it can be used to guide and constrain the subordinate design and maintenance activities. Since eventual design inconsistencies can then be avoided or revealed early, it helps to promote the product quality and reduces the development cost.
- Architecture is closely related to handling the complexity of the system**.** By means of artifacts, e.g. architectural models/views and languages, only the essential system information at high abstraction levels may be represented. Thus, it is useful to assure the conceptual integrity among the designers and

other stakeholders and constitutes a basis for communications, which are important for all development activities.

- Architecture provides insights into particular problems. By representing mature high-level design solutions and rationales as architectural styles/patterns and principles, it gives a solid basis for high-level decision-making. Reusing these styles and principles leads to a much more effective design process for domain products or product families.

Although many definitions of the term "architecture" have been made (see *Table 1*), this definition manifests the scope of this concept and is comprehensive in that it encompasses several others. Many practices are encompassed in this definition, e.g., TTA (Time Triggered Architecture), Simplex Architecture [Sha et.al.96][Sha97], Guards (Generic Upgradable Architecture for Real-time Dependable Systems) [DeVa97], OSACA (Open System Architecture for Controls within Automation Systems). A similar approach for the definition can be found in [Jackson&Boasson95] for computer based systems.

Type	Approaches	Definitions
System Architecture	INCOSE SAWG [INCOSE97]	"The fundamental and unifying system structure defined in terms of system elements, interfaces, processes, constraints and behaviors". System architecting is "the activities of defining, maintaining, improving, and certifying proper implementation of the system architecture".
	IEEE AWG [IEEE P1471/D4.1]	"The highest-level conception of a system in its environment".
Software Architecture	Perry and Wolf [Perry&Wolf89] [Perry&Wolf92]	"Architecture is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for the design." The model: "Software Architecture = {Elements, Form, Rationale}"
	Garlan and Shaw (1995)	Software architecture is about structural properties of a system, including components, their interrelationship and principles about their use.
	Bass, Clements and Kazman [Bass&Clements&Kazman98]	"The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them".
	Luckham [Luckham *et. al.* 95a]	"A definition of a set of modules and the interactions between them, by means of interfaces, connections, and constraints."
	Selic [Selic96]	"The architecture of a software system refers to its highest-level modular decomposition and the interrelationship patterns between its modules. An architecture serves as a blueprint for implementation and also as the chief determinant of a system's ability to evolve."

Table 1-Some definitions of system architecture and software architecture.

3 Architecture Descriptions

To acquire those potential benefits, the architecture has to be described in concrete forms by means of artifacts. To avoid misunderstanding and decision-making mistakes, the architectural description must be unambiguous for understanding and commination, and comprehensive enough for reasoning and analyses.

To assist the system design activities and keep their integrity, it is required that the description should provide the indispensable and significant information from all concerned aspects [Rechtin&Maier97]. At a higher level, these concerned aspects can be summarized as,

- the context and the objectives of the system and subsystems. (Problem issues);
- the dynamic properties of the system solutions, i.e., the temporal behaviors of the architectural elements and their temporal interrelationships; (Behavioral and Performance Issues)
- the static properties of the system solutions, e.g., the logical functions of architectural elements and their logical interrelationships. (Structural Issues)

For a family of systems, the description should also support descriptions of architectural styles for comparison and reuse. For critical systems, it is required that more detailed information at lower abstraction levels should be provided for more precise prediction of the system properties.

It is clear that a large content of information may exist in an architecture description. A single representation for all these information becomes impractical or even infeasible. Therefore, multiple "orthogonal" architectural views are introduced as a way to handle the representation complexity and promote the clearness and unambiguity of the information representations. These views separate concerns within the architectural representation so that each view carries only the most essential information for a distinct or tightly related concern(s) [Rechtin&Maier97] [Ellis *et.al.*96] [Kazman *et.al.*96] [Hilliard&Rice98] [Clements&Northrop98].

All the views are then represented by means of different categories of architectural models [Rechtin&Maier97], e.g., system interconnect and flow diagrams, task and threads diagrams, data and event flow networks, difference equations and control algorithms, analytical models for performance, and system development meta-models. In these models, there are sets of individual abstract constructs named architectural components. Each of them exists in a distinguished context and has logical and temporal relations or dependencies to other components and the outer environment. The logical relations are often established by information dependencies. The temporal relations are usually given by application constraints (e.g., periodicity, deadlines, mutual exclusion, precedence, and communication latencies). When implemented in software, the former is often expressed data flows and the latter in control flows. This concept of architectural components is illustrated in [Törngren93]. The control application model of an embedded real-time system is composed of control components and logical channels. The control components are sequential programs executing periodically. Depending on the context of control components, they are further classified as computation components (of global or local servo type) or actuator and sensor components, which have a direct interface to a physical device. Their logical interrelations are expressed in and realized by logical channels (LC), which are unidirectional broadcast channels for ordered transmission of data. Since there are also temporal relations or timing requirements associated with the control components (e.g., release time, execution time, deadline, and jitter), a logical channel should transmit the data with upper delay bounds.

For generic software architectures, the architectural abstractions have been identified in a similar way. In [Ellis *et. al.* 96], they are expressed in components, and relations including connections and constraints, where

- components are the major structural elements in a view, e.g., functions in a functional view, data models in a data view, or hardware in a physical view";
- connections/connectors are the major relations between components of a view, e.g., run-time relationships like control or data flow, or other dependencies;
- constraints are laws the system must observe, e.g., constraints reflecting performance functional or non-functional constraints, style and protocol rules and resource constraints.

Here, the term "connection" and "connector" are used interchangeably. Sometimes, the former may be used for the relation between two components, while the latter indicates the entity realizing the relation. A similar approach can be also found in [Luckham et. al. 95a] and [Luckham et. al. 95b]. Other approaches do not address the constraint issues explicitly (e.g., in [Perry&Wolf89], [Perry&Wolf92] and [Shaw&Clements97]), or address only topological constraints (e.g., in [Shaw&Garlan96]). This approach has also been used to model real-time software, see for example [Paoli&Tisato95].

Currently, several approaches have been taken to identify the architectural views with different standpoints.

- Based on the concerned information, e.g., "what the client wants", "what the system is", "what the system does", "how effectively the system does it", the architectural views and their models are defined as [Maier96] [Rechtin&Maier97]: "objective and purpose models", "models of form", "behavioral (functional) models", "performance models", "data models" and "managerial models". This concept of the architectural views is illustrated in many popular modeling methods, e.g., Hatley/Pirbhai (Hately 1988), OMT-Object Modeling Technique (Rambaugh 1991), ADARTS (SPC 1991), MSA-Manufacturing system analysis (Baudin 1990).

- Based on the progress of the design process, the software architectural views and models are identified as in the "4+1 View Model of Architecture" [Kruchten95]. Similar concept can be found in [Clements&Northrop98] [Bass&Clements&Kazaman98] [Ran98]. These views are logical/conceptual view, development/module view, physical view, process/coordination view and scenario. This concept of the architectural views is adopted in UML-based tools.
- The SAWG of INCOSE [INCOSE97] has identified the system architectural views based on the system stakeholders' perspectives. To describe a system, there are four views: functional view (i.e., owner's perspective), operational view (i.e., user's perspective), implementation view (i.e., designer's perspective), technological view (i.e., builder's perspective).

Therefore, architectural views represent different aspects of the system, ranging from the requirement to implementation issues. Implementing multiple architectural views with separated concerns to describe the problem and solution spaces is necessary to facilitate comprehension and communication. However, interrelationships between these architectural views have to be maintained in the modeling framework to ensure all the benefits given by the architecture concept. This necessity has been illustrated in the "4+1 View Model of Architecture" and other well-known software specification or design tools, e.g., CORE (British Aerospace and System Designs 1979), Hatley/Pirbahi method (Hately 1988) and Statemate (Harel *et. al.* 1990). There are still work to be done on modeling frameworks for complex mechatronics systems [Törngren&Redell99]. Some related concepts of integrating architectural views can be found in [Issarny98], [Ran98] and [Wang98].

4 Architectural Styles

As mentioned earlier, styles are well-understood and widely-accepted distinctive manners of designing, performing or presenting a system or a set of systems. They may exist in different abstraction levels and design stages. At the architecture design stage, the styles of the solution structures and behaviors are especially interesting. These styles represent the well-understood and implemented common overall solutions of a family (or product-line) of systems, and thus provide a solution→problem approach to the system development. They give a set of benefits, e.g., merging the conceptual gaps (by indicating well-known overall solution properties, e.g., layered software system), eliciting hidden requirements early and providing certain quality predictions. Reusing these mature solutions makes the system development and maintenance works more efficient.

Another often used term for the mature solutions is "pattern". A pattern for the software systems has been defined as "predesigned 'chunks' that can be tailored to fit a given situation and about which certain characteristics are known" [Bass&Clements&Kazaman98]; or, "a particular recurring design problem that arises in specific design context", and its "well-proven generic scheme for its solution" by means of "the constituent components, their responsibilities and relationships and collaborating rules" [Buschmann *et. al.* 98]. Software patterns can exist naturally in several levels of abstraction. In [Bass&Clements&Kazaman98], they are classified as high-level system patterns (also called architectural styles), design patterns and low-level code patterns. In [Buschmann *et. al.* 98], these levels are classified as high-level architectural patterns, design patterns and low-level idioms. Thus, patterns exist at code, software module and system levels; patterns in lower levels refine higher ones. For example, since design patterns have lower abstraction level than architectural styles, they provide schemes for "refining the subsystems or components or their relationships" [Buschmann *et.al.*98]; or, establish vocabulary and defines the solution space for finer-grained design problems [Bass&Clements&Kazaman98].

Currently, the architectural styles/patterns for other subsystems than the software system are still underdeveloped. However, in the automatic control, there are a set of well-known mature solutions, e.g., schemes for feed-back, feed-forward, adaptive controls [Åström&Witternmark97], passive and active perception controls in robotics [Buttazzo96], deliberative and reactive controls [Pettersson99]. Since these mature solutions specify the control structures, detailed analytical design methods and data processing requirements, there is potential for them to be further elaborated and classified as styles/patterns.

For the software architectural styles, there are two major approaches: CMU and Pattern Community. Both of them address that a style is not architecture, but an abstraction of the essential features commonly existing in a set of architectures.

The CMU Approach:

In this approach [Shaw&Garlan96], a software architectural style is "a pattern of structural organization" defining "a vocabulary of components and connector types and a set of constraints on how they can be combined". Similar definitions can also be found in [Shaw&Clements97] and [Bass&Clements& Kazaman98]. In particular, the styles represent a family/class of architectures with a common vocabulary of components, connectors and configurations, the underlying computation model, the semantic model and invariant properties, rationale, principles and guidelines. A style can also be thought of as a set of design decisions or "constraints" on architecture, which decide "the component types and their patterns of interaction". Since software architectural views can have different notations or constructs, there can be different architectural styles for different architectural views.

This approach analyzes various existing architectures and tries to deduce a set of fundamental ways of composing architectural components. The classification criteria are based on some software features, i.e., type of components, control and data issues. The identified architectural styles are described in [Shaw&Garlan96] and [Bass&Clements&Kazaman98]. In fact, based on the predominant characteristics of component interrelationships at the implementation/software level, all these styles can be grouped in two major categories: data- and control-relation-oriented styles. The data-relation-oriented styles (e.g., the data-centered styles and the data-flow styles) are mainly discerned by the way that data are passed among components. While the control-relation-oriented styles (e.g., the call-and-return styles, the virtual machine styles and the independent components styles) are mainly discerned by the way that control are carried out.

To support decision-making, trade-off among alternatives and guide the design process, a finer classification of these architectural styles and their variants are introduced in [Shaw&Clements97] (also [Bass&Clements&Kazaman98]). The major classification criteria are still the implementation level interrelationships among the components, but finer-grained discriminations are carried out. In addition, these interrelationships are more strictly defined. They form a solid base guiding the design choices, e.g., the rules of thumb for choosing styles to fit the problem in [Shaw&Clements97] or [Bass &Clements&Kazaman98].

The Pattern Community Approach:

In this approach [Buschmann *et. al.* 98], an architectural pattern is "a fundamental structural organization schema" for system-wide structural properties. It provides "a set of predefined subsystems" with specified responsibilities, rules and guidelines for organizing their relationships. Architectural patterns can be thought of as "templates for concrete software architecture".

This approach documents mainly the structural properties of proven solutions together with their contexts and problems. The classification of architectural styles is based on the structural/compositional characteristics of OO solutions, e.g., functionality of objects and their collaboration/interaction rules. By documenting proven software systems for their "fundamental structural organization schema", the Pattern Community has identified several architectural patterns [Buschmann *et. al.* 98]. Depending on the primary concerns of the design problems, these architectural styles are categorized into four groups: "From Mud to Structure" (including Layers, Pipes and Filters, Blackboard styles), "Distributed Systems" (including Broker style), "Interactive Systems" (including Model-View-Controller (MVC) and Presentation-Abstraction-Control (PAC) styles) and "Adaptable Systems" (including Refection and Microkernel styles).

Compared with the CMU approach, this approach addresses more the rules of composing functionality of subsystems with Object-Oriented technology to satisfy the requirements of systems. Instead of classifying the styles based on the characteristics of components and their interrelations, the architectural patterns are organized in categories according to their context or intention. Here, the subsystems become the only architectural components.

5 Architectural Principles and Methods for X-abilities

There are a number of requirements or quality attributes imposed on the system. Some of them are discernable during the run-time, e.g., functionality, predictability and dependability (including reliability, availability, safety and security). The others are related to life cycle management but also have strong influence on the system quality, e.g., modifiability and maintainability (including extensibility, compressibility and portability), reusability, integrability, testability and verifiability. Some of the architectural principles and methods for these quality attributes are briefly introduced below.

An essential aspect of system performance is that of timing predictability of system activities. In order to assure this quality attribute, scheduling theory can be used as the means to find an optimal feasible assignment of contending activities/tasks/elements to spatial and temporal resources. At the architectural level, these analytical methods provide the most important quality prediction of the system control performance. In current implementation practices, there have been two basic scheduling approaches to the overall design of time-critical applications, i.e., off-line scheduling and priority-based scheduling. The choice between these two approaches depends on the characteristics of the application and has deep impact on the system architecture. For example, if jitter control is the most important issue, off-line scheduling might be more suitable; if the application needs also to fulfill the modifiability/maintainability attribute, the fixed-priority approach might be a better choice. While the predictability of the system is mainly decided by the engineering principles, many other quality attributes depend more on the system's architecture.

For high dependability, error detection and handling becomes one part of the system functionality. However, the solution may be architectural or not, depending on the technique used. Error detection and handling by implementing replication/redundancy and diversity to architectural components is an architectural design problem. On the contrary, error detection and handling by checkpoints and exception handling may not be an architectural design problem when it is only done at the component level. For analyzing and predicting the system reliability quantitatively at the architectural level, Markov modeling [Kitchin88] can be used. This flexible, graphically assisted technique defines the operational states of a system component or subsystem and the transitions between them. It provides also a probability model specifying "the likelihood each sate is entered over time and the distribution of waiting time in the state when entered".

Modifiability and maintainability is largely a function of the locality of any change [Bass&Clements& Kazman98]. This property can be acquired by clearly defined components' responsibilities or functions so that one modification of the system should concern as few components and interrelationships as possible. (Compare with the principle of encapsulation – Parnas 1972 [Cooling91].) Integrality, interoperability, testability and reusability depend mainly on the external complexity of the components, their interactions mechanisms or protocols, and the degree to which responsibilities have been cleanly partitioned. It is frequently recognized that the separation of concerns is essential. However, a poor aggregation or partition can increase the system complexity. It makes the system more difficult to be understood and analyzed, and unable to satisfy the performance goals. Therefore, good architectures should have low coupling and high cohesion of the architectural constructs. (Compare with the Yourdon principles- Yourdon 1979 [Cooling91].)

There are many methods for analyzing these quality attributes achieved in software architectures. One of the them is the well-known and used SAAM (Software Architecture Analysis Method) [Kazman et.al.96] for analyzing individual quality attributes achieved in a software architecture. It is designed for quality attributes that are too vague to be measured directly (e.g., modifiability). All analyses are based on scenarios, which are "brief narratives of expected or anticipated use of a system from both development and end-user viewpoints". They can be thought of as the particular instances of each quality attribute important to the stakeholders of the system. By exercising the architectures with the scenarios, the system's fitness with respect to a set of desired quality criteria can be determined. The method allows also the coupling and cohesion of the architectural constructs to be measured with respect to a certain scenario(s). SAAM provides a set of well-proved activities/steps and dependencies between them (see Figure 1) to reveal the properties of the architecture with the scenarios. It is suggested as a canonical method for scenario-based architecture analysis of computer-based systems. The analyses are started by

describing the candidate architecture(s) with a syntactic architectural notation accepted by all concerned stakeholders (i.e., Architecture Description). Then, all important uses of the system by all stakeholders are represented in the scenarios. Since the first and second steps (i.e., Architecture Description and Scenario Developent) are interdependent, iterations between them are required to determine the proper level of architectural description and the proper set of scenarios. Thereafter, all these collected scenarios are evaluated separately (i.e., Individual Scenario Evaluation). If a scenario execution can not be supported and a change of the candidate architecture is required, this scenario is classified as an "indirect" one. The results of this step are summarized in a table that lists all direct and indirect scenarios. In the table, the impacts and required changes caused by the indirect scenarios are also described and listed. In the next step (i.e., Assess Scenario Interactions), "scenario interactions" measuring "the extent to which the architecture supports an appropriate separation of concerns" are evaluated. When different indirect scenarios may require changes to the same components or connections, these scenarios are said to be "interact" in those architectural units. At the last step (i.e., Overall Evaluation), each scenario and the scenario interactions are weighted subjectively in terms of their relative importance. These parameters form the base for the overall ranking of the candidate architecture(s).

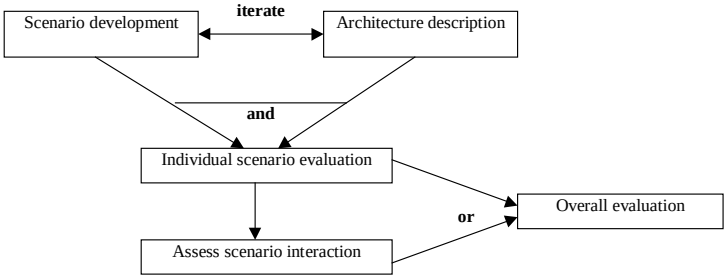

Figure 1- Activities and dependencies in scenario-based analysis (adopted from [Kazman et.al.96]).

To further "evaluate a software architecture's fitness with respect to multiple competing quality attributes", ATAM (Architecture Tradeoff Analysis Method) [Kazman et.al.98a] [Kazman et.al.98b] can be used. It characterizes the dependencies and conflicts between these quality attributes explicitly**,** and thus reduces the risks of decision-making. ATAM implements a spiral refining process running from scenario and requirements acquisitions to architectural tradeoffs (see Figure 2). It is started with eliciting system usage scenarios (i.e., Collecting Scenarios) and identifying "the attribute-based requirements, constraints, and environment of the system" (i.e., Collecting Requirements/Constraints/ Environment). After the design solution space has been defined by the requirements and scenarios together with engineering principles, candidate architectures can be generated. These candidate architectures are described in multiple architectural views, where separate architectural elements and properties are used for particular quality attributes (i.e., Describing Architectural Views). Meanwhile, the collected scenarios can be mapped or realized on these candidate architectures (i.e., Realize Scenarios). Based on the previous work, each quality attribute is then analyzed in isolation in order to facilitate the work of individual attribute experts (i.e., Attribute-Specific Analyses). These analyses result in statements about system behavior related to values of particular attributes. Thereafter, the sensitivities of individual quality attributes to particular architectural elements are identified (i.e., Identifying Sensitivities). It is carried out by varying the attributes, and evaluating subsequent changes in the models. "Any modeled values that are significantly affected by a change to the architecture are considered to be sensitivity points". At the last step of the loop (i.e., Identifying Tradeoffs), the eventual defects in the candidate architecture model can be revealed by means of the interaction of attribute-specific analyses and the location of tradeoff points. The tradeoff points are located at architectural elements where multiple attributes are sensitive. After one iteration loop, it is possible to check in which degree the given requirements are fulfilled and which architectural elements have influences on them. Often, a new iteration loop is necessary to improve or eliminate certain revealed characteristics. Since the new iteration is based on the previous results, it produces a new, more elaborated and detailed architectural design.

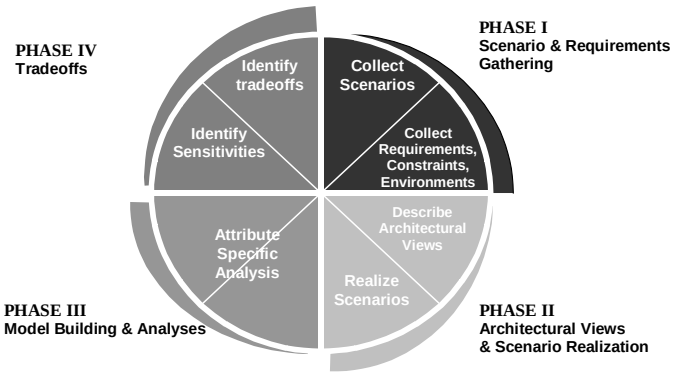

Figure 2-Steps of the ATAM (adopted from [Kazman et.al.98a] [Kazman et.al.98b]).

6 Summary

To handle the complicated nature of computer control systems and to remedy the problems in the traditional development process, the concept of architecture needs to be adopted in the design of the complex mechatronical systems. Having good system architecture is a necessary condition to assure the final quality of the designed system. It provides the high abstraction level representations of the upcoming system, which constitutes the means to satisfy the needs of emphasizing early design phases for the critical early design decisions and keeping integrities during the entire life cycle. Since architecture tries to represent mature design solutions in architectural styles, it can also provide a systematic way to reuse earlier design decisions or experiences, modules and COTS (Commercial Off-The-Shelf) products. Moreover, architecture tries to capture essential science- and practice-based principles that have major influences on the system functions and thus guide the design process.

The abstract architecture has to be described in concrete forms by means of artifacts, e.g., languages, diagrams, charts or mathematical models. To avoid misunderstanding and decision-making mistakes, the architectural description must be unambiguous enough for understanding and commination, and comprehensive enough for reasoning and analyzing. Therefore, multiple architectural views with separated concerns should be implemented to describe the properties of the mechanical, control, software and hardware systems.

Architectural styles represent the well-understood and implemented common overall solutions of a family (or product-line) of systems. They provide a solution→problem approach to the system development. It Using architectural styles gives a set of benefits, e.g., merging the conceptual gaps, eliciting hidden requirements early, providing quality predictions, and promoting reusing. With these styles, the system development and maintenance works can become more efficient.

To analyze the system architecture and evaluate its satisfaction of the desired quality attributes, there are two distinct approaches. To predict the behaviors, predictability and reliability of the system, analytical methods (e.g., state machines, scheduling theories and Markov Modeling) can be utilized. To predict and assure other system quality attributes (e.g., modifiability), non-analytical methods (e.g., SAAM and ATAM) are very useful.

Further research should consider how to provide the reasonable prediction of the upcoming system properties with architecture. Since some properties can not be fully created at the architectural level, it is critical that the necessary assumptions (e.g., reliability estimations and execution time budgets) do not undermine the benefits given by the concept for analyzing and decision-making. To facilitate this work, architectural styles (especially control styles) should be further elaborated for the system.

7 Acknowledgements

This work has been supported in part by ARTES.

8 References

[Bass&Clements&Kazaman98] L.Bass, P.Clements and R.Kazaman, *Software Architecture in Practice*, ADDISON-WESLEY, ISBN 0-201-19930-3, 1998.

[Buttazzo96]G. Buttazzo, *Real-time issues in advanced robotics applications*, Real-Time Systems, 1996, Proceedings of the Eighth Euromicro Workshop on, Page(s): 133 –138, ISBN: 0-8186-7496-2.

[Buschmann *et. al*. 98] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*, John Wiley&Sons Ltd., ISBN 0-471-95869, 1998.

[Clements&Northrop98] Paul C. Clements, Linda M. Northrop, *Software Architecture: An Executive Overview*,Software Engineering Institute**,** Carnegie Mellon University, Technical Report CMU/SEI-96-TR-003, ESC-TR-96-003.

[Cooling91] J.E. Cooling, *Software Design for Real-time Systems*, Chapman and Hall 1991. ISBN 1-85032-279-1.

[DeVa97] DeVa, *Guards-TTA joint selective open workshop*, 29 Sept - 1 Oct 1997, LAAS-CNRS Toulouse, France.

[Ellis *et. al.* 96] W. J. Ellis, et.al., *Toward a Recommended Practice for Architectural Description*, Proceedings 2nd IEEE International Conference on Engineering of Complex Commuter Systems, Montreal, Quebec, Canada, October, 1996.

[Gajski&Vahid&Narayan&Gong94] D.D. Gajski, F. Vahid, S. Narayan94, J. Gong, *Specification and Design of Embedded Systems*, Prentice Hall, 1994, ISBN 0-13-150731-1.

[Hilliard&Rice98] Rich Hilliard, Tim Rice, *Expressiveness in Architecture Description Languages*, ISAW '98. Proceedings of the third international workshop on Software architecture, November 1-2, 1998, Orlando, FL.

[INCOSE97] *A Guide to CASE Tools in Support of System Architecting- Background, Capabilities and Selection*, Draft Version, Systems Architecture Working Group (SAWG), INCOSE (International Council on System Engineering), January 1997.

[Issarny98] V. Issarny, T. Saridakis, A. Zarras, *Multi-View Description of Software Architectures*, ISAW '98. Proceedings of the third international workshop on Software architecture, November 1-2, 1998, Orlando, FL.

[Jackson&Boasson95] K. Jackson, M. Boasson, *The Benefits of Good Architectural Style in the Engineering of Computer Based Systems*, Systems Engineering of Computer Based Systems, 1995.,Proceedings of the 1995 International Symposium and Workshop, Pages: 103 – 113, ISBN: 0-7803-2531-1.

[Kazman et.al.96] R. Kazman, G. Abowd, L. Bass, P.Clements, *Scenario-based analysis of software architecture*, IEEE Software Nov. 1996, Vol. 13, Issue: 6, ISSN: 0740-7459, Pages: 47 – 55.

[Kazman et.al.98a] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, *The Architecture Tradeoff Analysis Method*, CMU/SEI-98-TR-008, ESC-TR-98-008, July 1998.

[Kazman et.al.98b] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, *The Architecture Tradeoff Analysis Method*, ICECCS '98, Proceedings of Fourth IEEE International Conference on Engineering of Complex Computer Systems, 1998.

[Kruchten95] P.B. Kruchten, *The 4+1 View Model of architecture*, IEEE Software, Vol. 12, Issue: 6. Pages: 42 – 50, Nov. 1995, ISSN: 0740-7459

[Luckham *et. al.* 95a] David C. Luckham, James Vera, Sigurd Meldal, *Three Concepts of System Architecture*, Stanford University, 1995, CSL-TR-95-674.

[Luckham *et. al.* 95b] David C. Luckham, J. kenney, L. M. Augustin, J. Vera, W. Mann, *Specification and Analysis of System Architecture Using Rapide*, IEEE Transactions on Software Engineering, Vol. 21. No. 4, April, 1995.

[Oxford95] *Oxford Advanced learner's Dictionary of Current English*, 5th edition, Oxford University Press, 1995. ISBN 0194314219.

[Perry&Wolf89] Dewayne E. Perry, Alexander L. Wolf, *Software Architecture*, AT&T Bell Laboratories, 1989.

[Perry&Wolf92] Dewayne E. Perry, Alexander L. Wolf, *Foundations for the Study of Software Architecture*, ACM SIGSOFT Software Engineering Notes, 17:4 , October 1992.

[Pettersson99] L.Pettersson, *Control System Architecture for a Walking Robot*, Licentiate thesis, KTH, TRITA-MMK 1999:3, ISSN 1400-1179, ISRN KTH/MMK-99/3-SE.

[Paoli&Tisato95] F. De Paoli, F.Tisato, *Architectural abstractions for real-time software*, Software Engineering Conference, 1995. Proceedings 1995 Asia Pacific, on Pages: 199 – 208, 1995, ISBN: 0-8186-7171-8.

[Ran98] Alexander Ran, *Architectural Structures and Views*, ISAW '98. Proceedings of the third international workshop on Software architecture, November 1-2, 1998, Orlando, FL.

[Rechtin&Maier97] Eberhardt Rechtin, Mark W. Maier, *The Art of System Architecting*, ISBN 0-8439-7836-2, CRC Press, 1997.

[Selic93] B. Selic, *Modeling real-time distributed software systems*, Parallel and Distributed Real-Time Systems, 1996, Proceedings of the 4th International Workshop 1996, Pages: 11 – 18, ISBN: 0-8186-7515-2.

[Sha et.al.96] L. Sha, R. Rajkumar, M.Gagliardi, *Evolving Dependable Real-time Systems*, The Proc.Of the IEEE Aerospace Conference 1996.

[Sha97] L. Sha, *Shifting the Computation Paradigm of Real-time Control Systems*, in SNAERT' 97 Preprints, Lund University of Technology, Sweden, August 21-22, 1997.

[Shaw&Garlan96] Mary Shaw, David Garlan, Software *Architecture- Perspectives on An Emerging Discipline*, *Prentice Hall*, ISBN 0-13-182957-2, 1996.

[Shaw&Clements97] M. Shaw, P. Clements, *A field guide to boxology: preliminary classification of architectural styles*, The Twenty-First Annual International Computer Software and Applications Conference, 1997 (COMPSAC '97). ISBN: 0-8186-8105-5.

[Törngren93] M. Törngren, B. Garbergs, H. Berggren, *A Distributed Computer Testbed for Real-Time Control of Machinery*, Proc. 5th Euromicro Workshop on Real-Time Systems, 1993. IEEE Computer Society Press, ISBN 0-8186-4110.

[Törngren&Redell99] M. Törgren, O. Redell, *A Modeling Framework to Support The Design and Analysis of Distributed Real-time Control Systems*, Invited Paper. To appear in the Journal of Microprocessors and Microsystems, Elsevier, special issue based on selected papers from the Mechatronics 98 proceedings.

[Wang98] Z. Wang, *Architecture Abstraction Tower*, ISAW '98. Proceedings of the third international workshop on Software architecture, November 1-2, 1998, Orlando, FL.

[Wikander&Törngren98] J. Wikander, M. Törngren, *Mechatronics as an Engineering Science*, Proceedings of the 6th UK Mechatronics Forum International Conference, 1998. ISBN 0-08-043339-1.

[Åström&Witternmark97] Karl J. Åström, Björn Witternmark, *Computer-Controlled Systems, Theory and Design*, Prentice-Hall 1997, ISBN 0-13-314899-8.