Dissertation

Software Architecture Recovery of Product Lines

Ausgeführt zum Zwecke der Erlangung der akademischen Grades
eines
Doktors der technischen Wissenschaften
unter der Leitung von

o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Roland T. Mittermeir
(Universität Klagenfurt)
o.Univ.-Prof. Dipl.-Ing. Dr. Mehdi Jazayeri
(Technische Universität Wien)

Eingereicht an der Universität Klagenfurt
Fakultät für Wirtschaftswissenschaften und Informatik
von

Wolfgang Eixelsberger
Slependen, Norway / Feldkirchen, Austria
Matr.Nr. 8660222

EHRENWÖRTLICHE ERKLÄRUNG

Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit vollkommen selbständig verfasst und außer dem im Literaturverzeichnis angeführtem Schrifttum bei der Abfassung keine andere Unterstützung in Anspruch genommen habe. Die Arbeit ist noch keiner anderen Prüfungsbehörde vorgelegt worden.

September 2000

ACKNOWLEDGEMENTS

First and foremost I would like to thank my thesis advisors, Professor Dr. Roland Mittermeir, for providing academic guidance and intellectual support throughout my dissertation research and Professor Dr. Mehdi Jazayeri, for his advice, his support and his encouragement.

I am grateful to Dr. Rene Klösch and Dr. Harald Gall of the Technical University of Vienna for their friendship and technical support while working on this dissertation. Their feedback on publications and earlier versions of this work was invaluable for the progress in my thesis work.

Special thanks to ABB Corporate Research Norway, which was not only an inspiring place to work but also provided the opportunity to carry out this research. Many thanks to the former president of ABB Corporate Research, Dr. Markus Bayegan, and my department leader Asmund Maeland. I am especially grateful to Lasse Warholm and Håkon Beckman for their friendship and for the support in analyzing the case study. Lasse's and Håkon's lively presence in talks has always been educational and inspiring.

I must thank the ARES consortium (European Commission, Nokia RC Finland, ABB Corporate Research Norway, Philips RC Holland, Imperial College London, the Technical University of Madrid, and the Technical University of Vienna) for funding the ARES project that made this research work possible. I am also grateful to my fellow members in the ARES project. Special thanks to Sturle Mastberg, Trond Løkstad, Michaela Ogris, Manfred Kalan, Berndt Bellay and Keng Ng for a great deal of technical help and for their comments. Thanks to Jake Diephuis for improving the English language in this work.

My parents have always supported me throughout my life with trust and encouragement. They have always been a constant source of inspiration to me. Many thanks to both of you.

Table of contents:

Abstract

To benefit from software assets for the development of product families, it is necessary to understand and evaluate existing software architectures. The software architectures of existing systems are often not documented or have changed in the evolutionary process. In addition to the missing architectural description of product family members, the description of the common architecture, also called the family architecture or reference architecture, is missing. The reference architecture of an existing product family support architectural assessment and the forward engineering of new members of the product family.

To support the process of software architecture recovery of product lines, techniques were developed which are documented in this dissertation :

- SARM (Software Architecture Recovery Methodology) which is a methodology specifically designed for the recovery of architectures of product lines including the reference architecture. SARM uses information of members of a family of legacy software systems and transforms it in a stepwise approach to reference architectural representation.
- *Commonality analysis* is a method that is part of SARM and supports the recovery of the reference architecture. The *commonality analysis* identifies the reference architecture based on the architectural description of the family members.
- ASDL (Architecture Structure Description Language) is an architecture description language that supports not only the description of basic attributes of the architectural elements, but also additional attributes, so-called properties. The aim of the language is to completely represent the conceptual architecture of real-time systems and to provide input for the reference architecture recovery process. ASDL was developed in context with SARM and supports the *commonality analysis*.

The techniques were applied and evaluated in an industrial case study. The case study is a family of embedded real-time software systems for controlling train movement. A main result of the application of the recovery techniques is expected, significant quality improvement and saving when developing additional family members of the case study.

The development of these techniques and the application on an industrial case study was done as part of the ARES (Architectural Reasoning for Embedded Systems)[1] project.

[1] ARES was an international research project supported by the European Commission under ESPRIT framework IV contract no. 20477. The project was pursued by Nokia RC Finland, ABB Corporate Research Norway, Philips RC Holland, Imperial College, Technical University Madrid, and the Technical University of Vienna.

1. Introduction

In first part of this chapter the motivation for this work will presented. Further a short overview will be given of the technologies to resolve the identified problems.

1.1 Motivation

As a consequence of increasing system requirements (reliability, safety, reusability and maintainability) and technical changes, software systems have reached a new level of complexity.

The increase in complexity is caused by the facts that

- current software engineering moves from single product development to family product development
- software systems operate in heterogeneous environments
- systems are developed for customers with similar but not equal requirements (product diversification)

The architecture of a system plays a central role in complexity management. Architectural decisions influence almost all phases of development from design to testing and maintenance. Techniques that support reasoning about software architectures are therefore vital. Research is ongoing in the areas of architectural description, architectural assessment and architectural recovery. The architecture of existing legacy systems is of high interest for the development of successors of such systems. Legacy systems—which are systems successfully in use over a longer period of time—contain a high amount of valuable information. Particular information, required to reason about specific architectural properties of the software at a given time, may not exist in a cognizable form, is incomplete, or is no longer valid due to changes that have occurred since the information was created.

Architecture recovery is the process that extracts or completes architectural information about a software system or a family of systems (e.g. the architectural style, high level components and their relation). Recovery of architectural information supports the construction of a complete and unambiguous description of the architecture of a system. The recovery of software architectures from legacy systems facilitates system understanding, helps in identifying commonality within a family of software systems and thus facilitates the forward engineering process. Recovery of software architecture from existing systems is a relatively new concept. Experiences about recovery, description and assessment of software architectures of product families are currently lacking.

The goal of this dissertation is to define and evaluate a generally useful methodology for architectural recovery of product lines. In addition, the dissertation gives an overview of the current state of research in the area of software architecture in general and, specifically, the research and state-of-the-art of architectural recovery.

1.2 Problem description

Pressman [Pre94] p.17 uses the term ‘chronic affliction’ instead of ‘software crisis ’ to describe what the software community has endured in the past three decades. Each decade saw its specific problems and the software community reacted with suggestions for solutions. The 90’s faced its own specific problems and provided solutions such as software patterns but there are still unsolved problems that have to be solved in the new decade.

Brooks [Bro82] p.42 compares the building of cathedrals with software engineering activities and concludes that “Even though they have not taken centuries to build, most programming systems reflect conceptual disunity far worse than that of cathedrals. ” The reason for this is the separation of design into many tasks done by many designers. The most important consideration is therefore the achievement of conceptual integrity.

The software community is still facing a wide range of problems. The following sections give an overview of some of the most important problems related to industrial real-time system development.

Software products are often developed for one specific customer and later on refined for other customers with different requirements in terms of a product family. Refinements happen on the implementation level (algorithms and data structures) and on the architectural level (the overall system structure).

1.2.1 Complexity

Software systems have become more and more complex in the last few years. The increase in complexity is caused by many factors, e.g. the implementation of demanding new functionality. Complex functionality that has not been covered by software before has become an important factor and challenge in the software industry. A further step-up is the development of families of systems, in which requirements of future systems must be anticipated.

An example illustrating the increase in functionality is a train control system. Early versions of train control systems supervised the activities of the train engineer and issued warnings when the engineer acted in an irregular way. Current versions of train control systems completely control the train but are supervised by the engineer, who makes the final decisions in uncertain situations. In the near future, trains will be completely controlled by software without the opportunity for humans to directly interact with the system in use. Trains will cross national boarders with different train traffic rules and will be able to recognize the different system and react respectively.

The number of developers and programmers increases correspondingly to the additional functionality. Information about the overall structure of the system and general design guidelines must be conveyed to all members of the system development team to ensure full functionality of the software system.

The absence of a detailed understanding of the software architectures restricts the software engineer in making principle choices among high-level design alternatives. High-level relationships among similar systems are often not understood. The possibility of specifying new systems in the same system family is therefore significantly reduced.

The increasing complexity of software systems requires new complexity-reduction techniques. The human mind is incapable of handling all the details of a complex system at one time. Software systems are however not the first complex technical systems that have exceeded the capability of one human mind. Other examples are the construction of large buildings, factories and manual telephone switching systems. Breaking the conceptual systems down into smaller, more comprehensible parts reduced the complexity of such systems.

1.2.2 Legacy systems

Legacy systems are pre-existing systems, created using other design methods and technologies [JGJ97]. Legacy systems contain important knowledge, including requirements, design decisions, and business rules. Many organizations face the problem of reusing valuable information of their legacy systems. Legacy information helps in controlling maintenance costs, in improving system performance and it supports the development of successors of the legacy system. The number of large systems built completely from scratch is diminished. Most of the current large systems are built in one way or another on legacy systems by reusing code and/or design information.

Information about the overall structure of the system, the architecture, is however often not available or exists only in a restricted form. Block and line diagrams are typically used to convey architectural information. These diagrams are informal and the semantic is not as powerful as required. It is, for example, not possible to express design guidelines in an efficient way. The informal diagrams do not allow the assessment of different architectural solutions. Assessments are essential for choosing the most appropriate architectural style. Architectural information is not only available in informal block-line diagrams but also in source code, design documentation, views representing domain knowledge and in the developers' mental models. The architectural information is however not directly usable from the mentioned sources because the information is inherent and hidden. The recovery must extract the inherent information in an organized way.

1.2.3 Product Line Management

In many organizations, software systems are designed one system at a time. Future changes in the underlying technology or requirements are often not anticipated. Many organizations face at the same time the challenge of responding quickly to changing requirements of legacy systems and to requirements of new customers. Over time, families of systems were developed in these organizations without explicitly maintaining the multiple dependencies between the family members.

The product development cycle is shrinking because of rapid changes in functional and non-functional requirements. Technological changes (e.g. new operation systems for embedded systems) or changes in standards (e.g. 'Railway Applications: Software for Railway Control and Protection Systems') require the development companies to act accordingly by developing new product releases. The number of product line members is growing rapidly in many industries.

Each software family member needs significant investment in requirement analysis, design, documentation, implementation and testing. The consequences of designing one system at a time have been underestimated. Many organizations realize that they can no longer afford to develop one product at a time [BCC+97].

Program families are sets of programs whose common properties are so extensive that it becomes advantageous to study the common properties of these programs before analyzing individual differences [RS96].

Strategies are needed for the management and enhancement of product variations for international markets [DKO+97].

1.2.4 Non-functional requirements

A non-functional requirement of a system handles a particular aspect of the functionality of a system and is specified by non-functional properties. Non-functional properties address aspects related to reliability, performance, safety, compatibility, cost, maintenance or development of a software system [BMR+96] p. 389. Other non-functional requirements are real-time specific, e.g. security, availability, throughput, portability, response time, modifiability, portability, interoperability, reusability and migration. In the ARES book such requirements are defined as quality requirements [JRL00] p.9. Throughout this work, the requirements defined in this section will be named as non-functional requirements.

The increase in functionality demands increased reliability, fail-safeness and robustness. National and international regulations must be taken into consideration while developing such systems. As legislation systems change, software designers and programmers are confronted with stricter laws in case of disasters caused by faulty software.

Especially real-time systems must fulfill both functional requirements that are strongly related to data correctness and non-functional requirements that are often related to time correctness [SG96] p.188. A system that correctly calculates data response but fails to meet timing-bounds can have as a dangerous effect as a system producing spurious results in time [Jos96] p.3.

1.2.5 Problem synopsis

- Software systems become more complex because of increased functional and non-functional requirements
- Concepts for organizing and modeling such complex systems are currently not available
- High-level information of product line members is not at the disposal of system architects and designer
- Reuse of software components is restricted because of missing component management techniques

1.3 Requirements for an Architectural Recovery Methodology

- Recovery and description of the architecture of a software system - The methodology should support the recovery of the architectural views that represent the architectural decisions made by designers of the legacy system. The description of the architecture should represent the system in way close to the mental model of the developers. The methodology should also provide input to reference architecture recovery processes.

- Identification and Description of Reference Architecture - The methodology should provide techniques that allow the re-architect" to recover the reference architecture of a family of systems. The reference architecture should be described in a way that makes it suitable for system evolution and the derivation of new family members.

- Application to existing systems - The methodology must be applicable to typical complex, real-world legacy systems. These are systems, that are designed and developed by a number of different developers, running on different platforms and developed with different development approaches.

- Semi-formal approach - The methodology should provide a number of clearly defined steps that allow any „re-architect“ to repeat the recovery process. At the same time, the methodology should be adaptable and extensible.

- Notation support - The methodology should be supported by well-known views, supporting languages and available tools. The time for introduction to the methodology should be rather short; therefore, concepts well-known to architects and developers should be used.

1.4 Software Architecture Recovery Methodology (SARM)

SARM (Software Architecture Recovery Methodology) is a set of methods that uses information from members of a family of legacy software systems and transforms it with a stepwise approach to reference architectural representation[JRL00],[EOG+98],[Eix98]. The goal of SARM is to extract and represent complete architectural information of a family of software systems.

SARM provides 7 steps that lead from legacy system input to the reference architecture of a system (detailed information can be found in chapter 4):

Step 1:	Prepare input to recovery process
Step 2:	Extract structural architecture (components, interfaces and connectors)
Step 3:	Extract architectural properties (e.g. safety, timing constraints)
Step 4:	Describe the architecture (structural architecture & properties) with an ADL

In steps 1 to 4, the architecture of a single system is recovered and described. These steps are repeated for every (or selected) member(s) of a product family.

Step 5 to 7 recover the reference architecture of a family of products.

Step 5:	Extract the structural reference architecture from two or more structural architectures of family members (commonality analysis)
Step 6:	Extract reference architecture properties
Step 7:	Describe the reference architecture (structural architecture & properties) with an ADL

System shareholders can use the architectural descriptions for communication purposes and architectural assessment. The reference architectural descriptions may be used for architectural reasoning and for the development of new members of a product family.

Detailed information about SARM can be found in chapter 4; the application of the methodology is described in chapter 6.

1.5 The ASDL language

ASDL (Architecture Structure Description Language) [EOG+98], [EKO+98], [Eix98] is a flexible and extensible language for describing structural aspects of software architectures and their properties. ASDL is specifically designed for the description of software architectures of real-time system product lines. The aim of the language is to completely represent the conceptual architecture of real-time systems and to provide input for the reference architecture recovery process.

A number of Architecture Description Languages (ADLs) have been developed to support the description of software architectures. Current ADLs are, however, not sufficient in supporting specific properties of software systems such as safety and fault tolerance. ASDL supports the requirements for describing such architectural properties in addition to the high-level structure of the overall application. ASDL also supports the architecture recovery process SARM.

```
comp_begin
comp_id: B24
comp_name: depsm_a
comp_group: comp_A
comp_desc: The depsm_a component decodes the …
comp_role: primitive_component
comp_type: data_processing
safety_critical: yes
intf_name: TransPSM
intf_desc: The TransPSM provides …
intf_role: producer
intf_type: intf_out(shared data structure(field, D_dec_SAw))
…
comp_end

conn_name:    LOnfA
conn_desc:    The LOnfA connects ...
conn_intf:    TransPSM - Val
communication type:
        data_flow:    n.a
        control_flow:    unidirectional (...)
        con:  asyn
info_exch:    discontinuous
```

Figure 1: Component description with ASDL

Figure 1 gives an example of a component including one interface and a connector described in ASDL. ASDL keywords are written in *italic*. Each property is either described in plain text or by a property keyword defined in a property description set (like an enumeration type).

Components are either composite or primitive. Primitive components are not further decomposed; composite components consist of at least another composite or primitive component. Components contain one or more interfaces described by four attributes: *intf_name, intf_desc, intf_role, intf_type*. The component description ends with the keyword *comp_end*.
Connectors link interfaces of components together and are described by properties such as information exchange type (*info_exch*).

Detailed information about ASDL can be found in chapter 5; the application of the language is described in chapter 6.

1.6 Overview of the dissertation

Chapter 2 gives an introduction about work related to software architecture recovery. The concepts of software architecture, software architecture recovery, design recovery, architecture description languages, domain specific architectures, and product lines are introduced. Chapter 3 presents an overview and a comparison of architecture description languages.

Chapter 4 introduces the Software Architecture Recovery Methodology (SARM). SARM consists of 7 steps that raise the level of abstraction from the recovery input to the extraction and description of the reference architecture of a legacy system family. A specific language was developed for describing structural aspects of a software architecture. A specific commonality analysis process was developed based on that language. Both approaches are described in more detail in subsections of chapter 4. The approach of commonality analysis is introduced in section 4.2.5.

The Architecture Structure Description Language (ASDL) is introduced in chapter 5.

Chapter 6 describes the case study that was used for the verification of SARM. The case study consists of three members of a product family. The application of SARM on a single family member is described in sections 6.3.2 to 6.3.5. Sections 6.3.6 to 6.3.8 describe the identification and description of the reference architecture of the system family.

The appendix consists of 4 parts: section 7.1 gives a list of abbreviations; section 7.2 describes in detail the syntax and semantic of the Architecture Structure Description Language (ASDL) and section 7.3 lists the references used in this dissertation.

2. Related Work

In this chapter an overview will be given about related topics in the area of software architecture recovery.

2.1 Software Architecture

Webster's dictionary [Ran96] defines architecture threefold: “1. the profession of defining buildings, communities, etc. 2. the character of style of building. 3. the design or structure of something.” Because of historical reasons, the term ‘architecture’ has been strongly connected to building construction. However, as ‘3. the design or structure of something’ indicates, it is also applicable to other disciplines. In HW design, for example, the term has been used for a number of years.

Product families of complex products are constantly under development and maintenance in the industry. The complexity of software applications and software development organizations is growing. Support in handling the ever-growing complexity has been the main research topic for years. Software architecture is one of the latest developed concepts for complexity management.

Although the structure of software has been of concern for decades, software architecture has only recently emerged as an explicit focus of research and development [SDK+95]. Software architecture emerged in the beginning of the 90s as an explicit focus of research [SG95]. Software architecture recovery is an even younger research area and increasing research activity can be observed [GP95]. However, no universally accepted definition exists for the term ‘Software Architecture’ [GP95],[CN96]. Software engineering is a wide field with many different application types and domains. To agree on terms in the software architecture area will be difficult.

Garlan provided a definition of software architecture that has became one of the most cited and accepted in the software community - “The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.” [GP95].

The three main elements of a software architecture (components, connectors, design principles/guidelines) are widely accepted in the architecture community and often used in the literature [MQ94], [SDK+95], [GP95], [GAC+95] [PW92], [KC94], [BS92].

In a later work, Shaw and Garlan gave a more comprehensive definition of software architecture -“Structural issues include the organization of a system as a composition of components; global control structures; the protocols for communication, synchronization, and data access; the assignment of functionality to design elements; the composition of design elements; physical distribution; scaling and performance; dimensions of evolution; and selection among design alternatives. This is the *software architecture* level of design.” [SG96]p.1.

The concept of software architecture goes beyond design [Bha93], [AAG95]. Design is oriented to data structure and algorithm, whereas architecture is oriented to the structure of a system. Software architectures help in choosing among different kinds of structures and provide abstractions or generalizations of structures and of families of similar structures.

Software architecture also includes information about the properties that are related to the structure. Properties influence the selection of a structure and are therefore also consequences of that structure. Properties are, for example: performance, modifiability, portability, scalability, integrability, throughput, latency, gross performance, component consistency, security and reliability. Connector properties are typically protocol specifications which include rules about the types of interfaces the connector is able to relate, rules about the sequence of information exchange, and commitments of the interaction, e.g. performance. Properties are the glue that keep the system together and are often expressed as rule-statements [AF97].

Software architecture descriptions also contain rationale that document and justify constraints on components and connectors [TTC95], [PW92]. Rationale is the justification for the particular choices of structure and style [GAC+95], [IW95]. Rationale document design decisions that lead to decomposition into components and connectors. Rationale also support system evolution by serving as a maintenance guide. Rationale are most often derived from the system requirements [PW92]. Additional concepts (e.g. mapping and architectural style [MQ94], global system properties and life-cycle issues [AAG95]) are domain specific and can be added to increase the expressiveness. Other important concepts such as ‘product line engineering’ must be added.

2.1.1 Origins

Software architecture is not a completely new concept. The decomposition of software into a manageable structure has been an open issue since beginnings of modern programming. In 1968, Edsger Dijkstra [CN96] described the importance of software partitioning and structuring. Dijkstra was working on an operating system and used the concept of layered structuring for grouping programs in layers with constraints on program communication. Programs in one layer could only communicate with programs in adjoining layers. The layered organization allowed a conceptual integrity, simplified development and maintenance. Dijkstra’s work can be seen as the beginning of the study of software architecture.

Other important contributions to the roots of software architecture came from David Parnas [CN96]. Parnas not only worked on software structures but also on program families and argued about the importance of carefully making early design decisions. As one of the first in the community, he determined that early design decisions are the hardest to revise and therefore deserve careful consideration [Par76].

System designers have used software architecture concepts implicitly for a number of years. The modern research on software architecture first started in the beginning of the 90's.

Ghezzi et.al. [GJM91] p.65 examined important goals for software architecture - *design for change* and *product families*. *Design for change* refers to a technique in which the software architect tries to anticipate the change that the software may undergo through its lifetime. The resulting design should be flexible enough to accommodate changes easily. Designing a software architecture for a *product family* "reduces the change of inconsistencies and reduces the combined maintenance effort expended on all the products"[GJM91] p.71.

Perry and Wolf [PW92] proposed a model of software architecture which is a set of elements, form and rationale. Elements are either processing, data or connecting elements, form are the constraints on the elements and rationale provides are the underlying system constraints. The reasons for "slow progress in the development and evolution of software systems is that *we have trained carpenters and contractors, but no architects*" [PW92] p.52.

The book about software architecture from Mary Shaw and David Garlan published in 1996 [SG96] was an important milestone in establishing this research area in the community. The term 'software architecture' became one of the buzzwords in software engineering in the beginning of the 90s.

2.1.2 Why Software Architecture

There exists a number of arguments, why it is worthwhile to introduce software architectural concepts in the software engineering process:

- **Complexity management** - The design and specification of the overall system structure has become more significant then the choice of algorithms and data structures. For handling the complexity of algorithms and data structures, a wide range of programming environments, design notations and test tools exists. A complete representation of the architecture of a system is essential to the analysis and description of the architectural properties of a complex system. Brooks [Bro] p.42 suggests, that "It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas."

- **Organization management** – According to Cockburn it has long been said that "Architecture follows Organization" and "Organizations follow Architecture" [Coc96]. The architectural structure is closely related to the development team structure; work assignment and responsibilities; project, test and integration plans and other organizational aspects. This principle is valid in both directions. The organizational structure with its communication structures influences to a high-degree the architecture of a system.

- **Basis for communication** - Comprehensive understanding of software architectures allows the software engineer to make correct choices among design alternatives and allows him or her to communicate system design decisions to system shareholders. System shareholders (also called system stakeholders) - designer, programmer, tester, customer, etc.- are interested in different aspects of the software system. Software architecture provides a common 'language' for the system shareholders. The common language allows discussing, reasoning and negotiating the different aspects of the system and leads to increased quality and lower development and maintenance costs.

- **Early design decision** - Choosing the right architecture is crucial to the success of a software system design. Deciding on the wrong architecture can lead to disastrous results. Architecture is the first artifact in design that represents decisions on how requirements are to be achieved. Decisions on the architectural level represent the earliest decisions and are therefore the hardest to change and deserve the most careful consideration. The term 'architectural risk' [FS97] is sometimes used to describe the risk associated with putting aside architectural decisions until late in the project. Work done before the architectural decision will be compromised, which will result in effort for reworking.

- **Product line development** - The availability of a precise description of a software architecture supports recognition of common paradigms between software systems. High-level relationships among systems can be understood and therefore new systems can be built as variations on old systems. A well-structured reference architecture enables a quick response to changing requirements and changing markets. Boehm and Scherlis [BS92] identified that "at the level of a product line, where the capabilities, costs and schedules of multiple systems are considered in the aggregate, architectures can be made regular, and both organizational incentives and technology support can be brought to bear within individual projects."

- **System quality** - Quality attributes, such as performance, modifiability and security depend strongly on the architecture of a system. Performance depends, for example, on the inter-component communication, especially in real-time systems where components are physically distributed. Decisions on the architectural level influence the amount, frequency and structure of communication and coordination activities. "A well-defined architecture allows an engineer to reason about system properties at a high level of abstraction." [GMW97]. Complete architectural descriptions are an important aid in verifying the actual quality attributes versus the requirements. Software architectures also support the certification process (e.g. following a quality norm, as ISO 9001) where subsystems can be certified independently of each other.

2.1.3 Architectural Decisions

A system architect makes architectural decisions. Architectural decisions are early design decisions that will most likely remain constant across members of the program family that one may reasonably expect to produce [CN96]. Architectural decisions are of a higher-order than design decisions. Design decisions may never contradict architectural decisions.

An analogy can be found in the legal system. Constitutional laws are of a higher order than simple laws. Constitution laws build a framework for the legal system and restrict the way 'simple' laws may be defined. A 'simple' law may never contradict a constitutional law. Architectural decisions correlate in this case to constitutional law. They are higher-order decisions and restrict the design 'space' for system designers. In case the architect designs the system as a layered system, components on one layer can only communicate with programs in adjoining layers. The designer is therefore not allowed to design a component that communicates with a component in a non-adjoining layer.

Here are some examples of architectural decisions:

- Architectural partitioning - An architect partitions a system into manageable units. The result is a structural architecture with architectural elements. Architectural elements produce an architectural plan, from which component engineers can develop components that work for application systems developed by different groups or at different locations [JGJ97].
- Global error handling policy - The global error handling policy ensures correct system behavior in case of errors. Depending on the type of systems, different error policies can be developed. In embedded real-time systems, the demands on the error policy will be high. System architects must decide if the system is a fault-tolerant system or a fail-safe system. In addition, the software architect must decide, if the system is build around a central error-handling module or if each system component has its own responsibility for error handling. An example is of this is a "watchdog" that monitors the operation of the system. The components of the system send periodic life-signs to the watchdog. In case the watchdog fails to receive a life-sign, it assumes that something went wrong with the component and reacts accordingly.
- Architectural style – Architectural style consists of a set of well-formed constraints that must be satisfied by any architecture written in the style [MQ94]. Architectural styles provide a mechanism for capturing and reusing common structures in architectural design [GMW97]. The choice of the architectural style may be restricted because of predefined HW and/or system software (operating system, run-time system, CORBA, in-house system, e.g. signals layer). Architectural style rules limit interdependencies between architectural elements. In the case of components, the limited interdependencies result in easy substitution of components.
- Mapping of software to hardware - In early phases of system development the partitioning of implementation into software and hardware is often open. The architect starts with hardware-software co-design and decides then what components are implemented in hardware and software.

- General safety concept - The safety concept includes information about the fail concept (fault-tolerant system, failure detection or fail-safe) and the according connection to hardware (e.g. single hardware and multiple software).

Especially in embedded systems the architect has to map architectural elements to processors, busses, protocols and task threats.

2.2 Architecture Description Languages (ADL)

In many cases, software architectures are still described by simple block-line diagrams or circle-and-line diagrams. The blocks (circles) in such diagrams usually represent components and the lines represent connectors. Such diagrams give an intuitive picture of the software architecture of a system but prompt the following questions:

- What are the blocks (circles)? Are they components and if they are, are they run-time elements or processes or files or something else? What are the main tasks of the components?

- What are the lines? Do they represent connectors for information exchange or control or data flow? Are there time limitations related to the exchange? Does software or does hardware implement them?

In addition, such diagrams are informal and are difficult to test for completeness and correctness. To support architectural modeling and design, formal notations are needed. Architecture Description Language (ADLs) are such formal notations.

The IEEE Architecture Planning Group (APG) found, that ‘an architectural description is a model – document, product or other artifact – to communicate and record a system’s architecture.’ [EHP+96]. Since 1995, the APG has been working on the next generation of architecture-related standards and practices for the IEEE.

Architecture description languages (ADL) are formal languages for representing architectures of systems [Cle96a] and are therefore used to avoid the problems of block-line diagrams. ADLs focus on the high-level structure of the overall application rather than the implementation details [Ves93]. ADL must explicitly model components, connectors, and their configurations and must be tool-supported [MT97]. Many aspects of software systems can be represented with architectural description languages. Such aspects are functional behavior, allocation of resources, performance, fault-tolerance and flexibility [ADG98]. A number of ADLs have been developed as general-purpose languages and for specific domains.

ADLs fulfill different purposes:

- Basis for communication - System shareholders (project team members, manager, customers) use architectural representation for understanding and development.
- Description of earliest design decisions - Earliest design decisions, such as the mapping of requirements to components, are the hardest to change and must be chosen carefully.
- Project blueprints - Architectural decisions are the basis for structuring development teams, work assignments, schedule and fine-grain design.
- Product line development blueprint- An architecture may be reused in other systems using the reference architecture representation.

ADL have to support these roles by providing following characteristics [SG94]:

- Composition - ADL should permit the description of a system as a composition of independent components and connectors. It must be possible for the architect to divide a complex system hierarchically into smaller parts. Architectural elements must be independently understood and reused.

- Abstraction - ADL should be capable of describing architectural elements and the role they play in a clear and explicit way. The kind of relationships between architectural elements must be made clear without overloading the description with implementation details.

- Reusability - ADL should allow the reuse of architectural elements and architectural patterns in different architectural descriptions. Generic patterns of architectural elements should permit the system architect to describe a family of system architectures.

- Configuration - ADL should localize the description of the system structure and also dynamic reconfiguration. ADL should separate the description of composite structures from the elements in those compositions.

- Heterogeneity - ADL should allow the combination of multiple, heterogeneous architectural descriptions. Components can play different roles. The ADL must allow the definitions of different architectural patterns in a single system. The same is valid for the combination of components written in different languages.

- Analysis - ADL should allow rich analyses of architectural descriptions. For example, connectors associated with protocols should be analyzed for context correct use of the connector.

Medvidovic [Med96] compared different definitions of ADLs and in what sense they differ from related concepts like MILs (Module Interconnection Languages) and IDLs (Interface Definition Languages). Medvidovic clarified that ADLs are languages that provide features for modeling a software system’s conceptual architecture. The building blocks of ADLs are components, connectors and architectural configurations.

Architectural configurations contain properties like evolution, constraints and non-functional properties.

2.3 Architectural Elements

The following section gives an explanation of the main architectural elements: component, component interfaces, connector and architectural properties.

2.3.1 Component

A number of definitions exist for the term ‘software component’. Abowd defined components as ‘primary computations of the application’ [AAG95]. Mary Shaw defines components as ‘such things as clients and servers, databases, filters, and layers in a hierarchical system.’ [SG95]. Kogut gives a code-oriented definition of components: ‘Components can be small pieces of code (such as modules) or larger chunks (such as stand alone programs like database management systems).’ [KC94]. All of these definitions are developed from and are suitable for different domains and different needs.

As there are several definitions for “software architecture”, there is no widely accepted definition available for the term “component”. Different viewpoints, different programming language concepts and different domains have produced a number of definitions that are often unrelated. Here some examples:

- define the primary computation of the application [AAG95]
- are such things as clients and servers, databases, filters and layers in a hierarchical system [SDK+95]
- can be small pieces of code (such as modules) or larger chunks (such as stand alone programs like database management systems) [KC94]
- an abstractly characterized unit that can provide capabilities or services to others [TTC95]
- acts as either a source of data, a sink for data, or a transformational filter which converts input data into a different form of output data. [WRM+95]
- represent architecturally-relevant units of computation and data storage [ADG98]

When designers discuss a software architecture, they typically see the system as a collection of interacting components. Components are important for abstraction, functional decomposition, modularity, encapsulation, information hiding and organizing a development team. Components are starting points for organizing development teams. Requirements for components are accordingly high. Jacobson requires that components “must be powerful, have simple well-defined interfaces, be simple to find, understand and use, and have a wide area of application. Moreover it should be possible to build new components with the aid of existing components" [JCJ+94]. The software architecture provides a blueprint for component integration.

A component is described as a set of ports and the component specification. At the programming language level, components are often represented as a set of related functions, modules or classes. Components interact through ports with their environment. Ports can be passive (e.g. data structures) or active (e.g. functions writing data to the port of another component). The type of port determines the type of connector that relates two components. The composition of components and connectors must satisfy the specifications of both the components' interfaces and the protocols of the connectors.

The component specification determines the functionality performed by the component. Components typically act as data source, data sink or transformation filter [WRM+95]. Components that are 'specified-for-reuse' create an incentive to achieve conventionalized software architectures for classes of related systems [BS92].

Components may be either primitive or composite [SG96]. Composite components define configurations in a notation independent of conventional programming languages of our choice. Composite components define configurations in a notation independent of conventional programming languages.

The OMG (Object Modeling Group) is currently developing a component standard called the Object Management Architecture (OMA). OMA will include standardized interfaces including specifications for interactions between ActiveX and JavaBeans [Kie98].

2.3.2 Interface

Interfaces are the parts of a component that provide access to the services of the component . Other components can change or access the state of the component by using the services of the component. Interfaces realize protocols, which define the valid flow of information between connected components. A component can provide a number of interfaces, but it always has at least one.

A component interface is the partial specification of a component and represents interactions in which the component may participate. Connectors use the features provided by interfaces for communication between components. Component interfaces include more than the mechanisms that a component uses to make its functionality available to clients [BW96].

An interface contains constraints, such as pre- and post-conditions or information about the ordering of operations to ensure consistent interactions [JGJ97].

Brown and Wallnau [BW96] defined a list of considerable parts of component interfaces:

- application programming interface

- secondary storage requirements, processor and network requirements
- required software services (operating system, or from other components)
- security assumptions (access control, user roles, authentication)
- embedded design assumptions, such as the use of specific polling techniques
- exception detection and processing

Interface Description Languages (IDL) are used to define interfaces for components and component systems. CORBA and DCOM introduce different IDL's that allow the specification of component interfaces independently from a programming language [BH97]. CORBA IDL, for example, supports the definition of operations, multiple inheritance, and C++-like types.

2.3.3 Connector

Connectors define the way in which components communicate with each other (e.g. procedure calls, pipes, remote procedure calls…). Connectors relate interfaces of components together (e.g. sends data to, communicates with, invokes [SG96]). Examples of connectors include pipes, procedure calls, message passing, and event broadcast [AAG95]. Interactions between components can be simple and familiar (procedure call, shared variable access) but they can also be complex and semantically rich (client-server protocols, database accessing protocols, asynchronous event multicast, and piped streams) [SDK+95]. There is little consensus on the role of connectors in an architectural model and on the necessity of making connectors first-class model elements [ORT98].

Connectors may be static or dynamic. Static relationships can be directly identified in source code. Dynamic connectors are temporal connections and dynamic interactions between components. Dynamic connectors may be difficult to identify in source code.

Connectors are defined by their roles and a glue specification. The glue specification describes how the activities of interacting components are coordinated. Connector roles describe the expected behavior of the connected components. Connector roles must be compliant with the component roles of the interacting components [AG94].

Some of the roles may be controlled or performed by a component, e.g. an object request broker (ORB). The ORB, for example, uses an object reference to locate the server process and to control the subsequent communication between both processes [BH97].

Connectors have protocol specifications. The protocol specification includes [SG96]:
- rules about types of interfaces the connector is able to mediate for
- rules about the order in which things happen
- commitments about the interaction (e.g. ordering performance)

2.3.4 Architectural Properties

Properties are essential or distinctive attributes or quality of a thing [Ran96]. Software architecture properties are essential attributes of the software architecture. The software architecture is the collection of architectural elements (components, interfaces, and connectors) and architectural properties are therefore essential attributes of architectural elements.

Architectural properties are additional information to architectural elements. Properties are not implemented on their own; they are part of the implementation of architectural elements. Properties contain design specifying information. The property *safety* strongly influences the design of the architectural element component. The property *safety* makes no sense for a system without relation to architectural elements.

System or global properties are properties of the component representing the overall system. Global system properties are aggregate properties, such as throughput, latency, gross performance, component consistency, security and reliability [GMW97],[AAG95].

Architectural properties influence and limit the degrees of freedom in the architectural design process. The design decisions are related to functional or non-functional requirements. Architectural properties are implemented by a set of techniques in the legacy system (e.g. software redundancy, time-outs, scheduling algorithms and others). Properties are the glue that keeps the system architecture together.

Properties can be divided into functional and non-functional properties [BMR+96] p.389. Functional properties deal with a particular aspect of a system's functionality. Functional properties are usually related to a specific functional requirement (e.g. throughput, latency). Non-functional properties represent a feature of a system that is not covered by functional properties. Non-functional properties typically address aspects related to the safety, reliability, maintenance or development of a software system.

The following is an overview of some important architectural properties:

Constraints

Constraints are used to assign functionality to individual components and to restrict the behavior of the interfaces and connections [LV95]. An architectural style consists typically of a set of constraints on a specific architecture.

Semantic constraints – e.g. a data store is not allowed to change values stored in it

Topological constraints – e.g. a pipe-filter system is not allowed to contain cycles, or components are only allowed to communicate with components in the same layer or in adjacent layers.

Constraints
- enforce that all requirements defined by the system requirements are accomplished
- enforce safety requirements
- enforce performance and timing requirements
- enforce the usage of communication protocols

Rationale

Rationale are the set of reasons on which a particular course of action, practice, or belief is based [Col92].

For software architectures it means that rationale record reasons for architectural design decisions:
- decomposition into components
- definition of connections between components

Rationale demonstrate that architectural elements define a system that, if implemented, would satisfy the collection of system stakeholders' need statements [GAC+95]. Rationale serve as a plan for evolution and as a guide for software maintenance.

Safety properties

Architectures for safety-critical real-time software systems are designed accordingly to safety-related properties. The bases for all safety-critical systems are fault prevention (development organization, tools and methodologies) and fault detection (diversified software / n-version software and hardware). Depending on the safety requirements, safety-critical systems can be of a low-demand (emergency shut-down systems) or high-demand (flight control system) mode of operation.

Guidelines must be developed to be used by designers in the component design process. N-version components with diversity guidelines and component design guidelines include limitations and information about tools and programming concepts.

Typical safety properties of real-time systems are:
- basic safety concept
- diversified software-related information
- granularity of safety-critical components

Standardization properties

Standardization properties contain information about:

- Techniques and measures necessary to achieve the required safety integrity level. A safety integrity level is a classification number which determines the techniques and measures that have to be applied in order to obtain the required risk reduction.

- Safety relevant information for components and connectors about the verification state (previously verified, not previously verified), if a component is safety-related or not and the safety integrity level of the component.
- Safety-related significance of all hardware/software interactions.
- Design features used for maintenance of the safety integrity of data.

Architectural style

Architectural styles and design patterns might help the reverse architect in understanding the architecture used in a system. In older systems the use of such idioms happened more on an accidental basis. Therefore, the effort to discover patterns and styles might be great, but it pays off in the following phase where the overall structure of a system is made more clear.

There are basic architectural patterns like layered systems, client-server systems or blackboard systems. Identifying the style of a system helps the general understanding of the architectural structure.

Architectural patterns

Architectural information is principally reflected in the data and processing model that were used to design the system. Certain patterns and „idioms" recur in views of the system. These patterns reflect design and architectural principles used by designers in creating new architectures. The architectural recovery process must be able to identify the occurrence of such patterns and relate their occurrence in different views of the system. Research results about patterns can be found in [GHJ+95], [Pre95].

2.4 Software Architecture Recovery

Software architecture recovery is the process that extracts or completes architectural information about a software system or a family of systems (e.g. the architectural style, high level components and their relation) [EWK+97].

2.4.1 Why Software Architecture Recovery

Architecture recovery is necessary because the particular information required to reason about specific properties of the software

- may not exist in a cognizable form
- is incomplete
- is no longer valid because of changes that have occurred since the information was created (measure for architectural conformance)

Kazman and Carrière [KC97] use the term 'shared hallucination' to describe the common state in which developers refer to an undocumented architecture. In this situation, the developers believe that they are talking about the same architecture without noticing the mismatch of their mental models. Software architecture recovery sidesteps this dangerous situation by providing a correct architectural model.

Woods et.al. [WCK99] recommends checking the evaluation of an architecture's properties because they are critical to successful system development. Therefore, it is necessary to focus on *architectural conformance* to understand to what degree the realized architecture conforms to the intended architecture. For this, two architectures must be available: (1) the intended architecture (2) the architecture as it is realized in the implemented system. The latter must be identified by software architecture recovery.

Architectural recovery is a backward or reengineering activity; however, it is more extensive than design reengineering. Vlissides describes the special requirements for architectural recovery: "When applied to software, 'reverse engineering' refers to analyzing a particular software system in an effort to recover its design. In contrast, 'reverse architecture' refers to analyzing many software systems in an effort to recover *recurring* designs and the rationals behind them." [Vli95]. Design recovery is a sub-field of reverse engineering. Design recovery uses, in addition to source code, domain knowledge, and heuristics, for support of program understanding [Big89]. The goal of design recovery is to recover all the information that is needed for someone to fully understand the system under consideration. The design recovery community faces similar problems as the architecture recovery community. The support of structural re-documentation is limited for program understanding. Structural re-documentation uses only source code as an input base.

Since software architectures determine the gross structure of a system, architectural representations enable software developers to explicitly describe, assess, and manage architectures of embedded software systems. The analysis of architectural descriptions of related products will also help to discover domain-specific architectural patterns that may serve to merge separately developed software into a family of systems. A system family or product line is a group of similar products with specific features and functionality required by different customers. All product line members share a common architecture and are built on a common set of components. Identifying, assessing and describing the common architecture of a product line support low-cost production and maintenance of applications.

Software architecture recovery has the aim of recovering software architecture related information from existing software systems. For the recovery of software architectures of product lines, the recovery process must be applied on representative members of the product line. The result of the recovery process is an architectural description for each product line member. This approach is therefore only applicable for product families with a rather small number of family members.

2.4.2 Architectural Recovery

Paul C. Clements [Cle96] sees technology for component and connector identification within the context of *architecture archaeology*. Object finders and program dependency graph generators are tools that might be modified to help uncover an architecture. This area is however source-code-oriented with the according disadvantages of architectural recovery, e.g. missing information about architectural properties.

Re-architects are often confronted with the concept assignment problem [BMW94]. The concept assignment problem describes the difficulty of relating source code elements to real-world and application-domain concepts. Source code of legacy systems contains all necessary information for correct execution of the system. It seems therefore logical that all necessary system information should be extractable from the source code. However, since architectural abstractions are generally implemented by 'low-level' programming language constructs, it is difficult to identify such abstractions in the source code. This discrepancy is even more evident in case of distributed systems [MK98].

Architectural recovery has to ensure architectural conformance. Architectural descriptions have to ensure that the architectural documentation matches with what has been implemented. The documented and the implemented architectures must correspond [KC97].

Many questions about architectural recovery are posed and addressed in this dissertation:
- How should manual recovery to the first level of abstraction be performed?
- How can the legacy code, the design documentation and the domain knowledge be analyzed to identify and extract architectural information?
- Which transformation rules and representation notations are necessary for the step(s) between the first level of abstraction and the architectural representation?

2.4.3 Software Architecture Recovery Methods

In the following sections, a number of software architecture recovery methods are presented.

2.4.3.1 ARM – Architecture Reconstruction Method

ARM (Architecture Reconstruction Method) [GAK99] is a semi-automatic method based on pattern recognition that supports architecture conformance analysis. The recovered architecture is analyzed for conformance with the documentation, especially the documented design patterns. The ARM is supported by the Dali architecture workbench (see section 2.4.3.2). The Architecture Reconstruction Method consists of 4 steps:

1. Developing a concrete pattern recognition plan – Constructing a pattern recognition plan consists of three steps: (1) recognizing nested lower-level patterns (2) recognizing the pattern's key element; (3) recognizing other elements

of the pattern and hence the entire pattern. The recognition plan is used to recognize a pattern in the system under consideration.

2. Extracting a source model – The source model represents the source elements and the relations between them. The output is a source model that contains the information used for the detection of pattern rules.

3. Detecting and evaluating pattern instances – Query tools (provided by Dali) are used to execute a recognition plan with respect to the source model. The output are all the pattern instance candidates. The candidates are compared with the designed pattern instances by the „re-architect". The result is a set of validated pattern instances.

4. Reconstructing and Analyzing the architecture – The „re-architect" aligns the recognized architectural pattern instances with the designed pattern instances. The recovered architecture is analyzed for conformance with the designed architecture.

2.4.3.2 Dali – A support environment for software architecture reconstruction

Woods, Carrière and Kazman [WCK99],[KC99] made the observation that no particular static set of reengineering and analyzing tools was sufficient for software architecture reconstruction. Therefore Dali was developed, a workbench that supports many implementation languages, many target execution platforms, and many techniques for architecture analysis. Dali allows the integration of a variety of tools and currently integrates several commercial and research tools (Rigiparse, Imagix, SniFF+, RMTool, IAPR, LSME).

Dali assists the „re-architect" in the definition of architectural patterns, view creation, view fusion, and pattern recognition based on SQL repository. Depending on the available input, the „re-architect" may choose between different views that support the recovery process.

Dali was used in a product-line migration effort [WCK99] for an industrial company. The recovery process specific for this case study included the following tasks:

1. Task A: Develop Component Information/Identification Form – In an identification form, the core set of components and connectors is identified.

2. Task B: Identify Components and Layers – The major architecture components are named and assigned to the layered structure of the system under consideration.

3. Task C: Populate Information Forms – The characteristics of the component are defined as architectural signatures. This is an important input for the next task in which rules for clustering software into architectural components are developed.

4. Task D: Develop Architectural Rules – Information describing the architectural components are stored in Dali and architectural rules are developed.

5. Task E: System Extraction to Database – Source code and makefiles are parsed into an intermediate representation stored in Dali.

6. Task F: Apply Rules to Database – Dali is used to match the rules defined in Task D and E versus the extracted system.

7. Task G: Map Models to Architectural Views – Using the tools provided by Dali the extracted models are mapped to a standard UML template.

8. Task H: Evaluate Architectural Views – The architectural views are evaluated for use in the ongoing development of the current system architecture. This verification will identify if the views are able to support the design of the current system architecture.

The process of the product-line migration effort is ongoing and no further experiences of the tool and the recovery process have been published.

2.4.3.3 Architectural slicing

Zhao [Zha99], [Zha98] developed the architectural slicing method to extract reusable architectures from existing architectural specifications. The two-phase algorithm removes from an architectural specification components and connectors so that the semantics of the specification are still maintained. The remaining architectural slice consists of reduced architectural elements. In reduced architectural elements, statements that are not of interest for the slice criterion are sliced away. The results are the saving of space in large-scale software systems and an increase in reusability.

The concept of *program slicing* is a technique used in the context of programming languages and usually uses control flow graphs and dependence graphs. Program slicing identifies parts of programs that affect values of a variable (called slicing criterion). From this idea Zhao derived the technique of *architectural slicing*. Architectural slicing operates on a formal architectural specification and not on source code written in a programming language. An architectural slice is a reduced architectural specification that represents a specific functionality which may be reused in new system's design. It removes from the specification those components and interconnections between components which are not of interest to the architect

An architectural slice is computed by two phases:

(1) compute a slice over the Architecture Information Flow Graph (AIFG) of an architectural specification
(2) construct an architectural slice

The Architecture Information Flow Graph (AIFG) illustrates information flows in a software architecture. The AIFG is an arc-classified diagram whose vertices represent the ports of components and the roles of the connectors. If an information flow from a role of a connector to a port of a component exists, then a connector-component flow arc is introduced.

In step (1) depth-first or breath-first graph traversal algorithm is used to traverse the graph starting at a point of interest. The result is a slice of the AIFG.

In step (2) each element of the slice is mapped to the original architectural representation (e.g. in WRIGHT) so that all unnecessary information is sliced away.

The resulting architectural slice represents the behavior of a specified set of architectural elements (components or connectors). The architectural slice can be used for designing new system architectures. Furthermore, architectural slices support the identification of parts of architectures that might affect or might be affected by modified components.

Architectural slicing is a second-step architectural recovery method since it already uses an architectural description that is recovered from a legacy system. Architectural slicing identifies additional information which makes it interesting for architectural recovery.

2.4.3.4 ManSART (MITRE Software Architecture Recovery Tool)

ManSART [CH98] [YHC97] [CHR+96] is a tool facilitating the recovery of architecture-level descriptions in the form of object and directed graph representations. ManSART uses a library of recognizers to identify architectural features in C or Fortran source code.

ManSART is available as ManSART/C and as ManSART/FORTRAN. Both tools are built on workbenches of the Refine tool. The tool parses source code into an abstract syntax tree representation from which architectural features are extracted.

The ManSART view representation consists of a component-connector graph, associations to source code fragments, and links to the conceptual framework defined for architectural styles. A repository view shows, for example, procedures and data-files and connectors representing data-flow between a procedure and a file.

ManSART was used to document software written in C and FORTRAN containing more than 170 kLOC. ManSART supports the identification of architectural styles, mapping between the styles and source code idioms used to implement the styles.

2.4.3.5 Mendonça & Kramer

Mendonça and Kramer [MK98] [MK96] identified limitations of reverse engineering approaches. They also documented how the limitations prevent the approaches from recovering architectures effectively from legacy systems. They classified reverse engineering tools and systems into five distinct frameworks:

- Filtering and Clustering Framework - This approach is based on a parser that extracts primitive software components and their definition/use relationships. This information is stored as a set of relations in a relational repository. The user may apply filtering and clustering operations to create different models of the system under consideration. A major limitation is that complex architectural relationships (e.g. inter-process communication) are implemented by different code fragments and are hardly extracted by extraction tools.

- Compliance Checking Framework - This approach is based on the same relational repository as used in the filtering and clustering framework. However, the user provides the system with a high-level repository that he expects to find in the system under consideration. The recognition system tries to find the model in the repository. The same limitations apply as described in the filtering and clustering framework.

- Analyzers Generators Framework - This approach is based on an AST representation annotated with comments and control/data flow information. AST are complex structures which contain language-independent nodes. The analysis and manipulation of such structures requires a deep knowledge of the user about the syntax of the underlying programming language. The analyzers generators framework approach may therefore be input to the following two approaches.

- Program Understanding Framework - This approach is based on a program abstraction library that represents user knowledge about programs. A recognition engine traverses the AST in an attempt to find program constructs that match program abstractions. Since program abstractions (e.g. plans, clichés) represent rather low level abstractions, this approach is of minor help in the architecture recovery process.

- Architecture Recognition Framework - This approach is based on a library of recognizers for identification of architectural elements of a set of architectural styles. The user provides the recognition engine with styles expected in the system under consideration. The source code is parsed into an annotated AST representation on which the recognition engine tries to identify the style. A major limitation is that this framework recognizes only one architectural style at a time and does not allow the combination of recognized styles.

Mendonça and Kramer propose an idealized framework that combines features of the architecture recognition framework and the frameworks based on relational source models. The AST model is not the basis for composing high-level models, but rather the basis for the creation of a relational repository. While analyzing the system, the user could apply analysis tools similar to those used in the filtering and clustering process.

2.4.3.6 Ownership-architecture centered recovery

Bowman and Holt [BH98] followed Conway's Law and analyzed how the organization of system developers can be used to form an ownership architecture. An ownership architecture relates developers to the code that they have developed. Conway's law indicates that organization of a software system will be congruent with the group that designed the system. Mapped to the level of software architecture, Conway's law means that the ownership architecture serves as the predicator of the as-build-architecture.

By examining the organization of developers, the assignment of persons to teams and subsystems can be identified. The ownership architecture indicates which person was assigned to which subsystem. Subsystems are related if there where people who have worked on both subsystems.

Bowman and Holt analyzed three large software systems and recovered three architectures for each system: (1) conceptual architecture, (2) ownership architecture and (3) concrete architecture. In one case a significant discrepancy was identified between the conceptual architecture, recovered from system documentation, and the concrete architecture recovered from the source code. The ownership architecture was closer to the system structure than the conceptual architecture. All three case studies showed that the ownership architecture is a good predictor for the concrete architecture.

2.4.3.7 FAMOOS approach

The ESPRIT project FAMOOS (Framework-based Approach for Mastering Object-Oriented Software Evolution) developed a pattern for extraction of architectural knowledge from legacy systems [TDM97]. The pattern is built around the problem, that knowledge of application architecture is hard to obtain. This is caused by missing or inconsistent documentation, or when developers are no longer available or difficult to access.

The solution is to build a prototype of the architecture that needs to be understood. The pattern recipe is an iterative process [TDM97]:

1. identify main focus points of the system (e.g. communication, user interfaces)
2. produce initial design based on available documentation and developer knowledge

3. produce prototype according to initial design
4. build prototype until the prototype is so complete that it ‘runs’
5. compare prototype design with legacy design (forces to find out about design decisions)
6. alter prototype and inconsistent documentation according to discussions with designers
7. goto step 4

The approach was tested on a user interface system for the remote control of a mail sorting machine. The interface system consisted of 350.000 lines of C++ code. Following the authors, the prototype helped to extract the architecture of the system. Problems occurred because re-engineering and forward-engineering activities interfered with each other. The effort for architecture extraction seem to be too great in the case study.

Another pattern-oriented idea for reverse architecturing was developed by John Vlissides [Vli95], a member of the well-known gang-of-four, which published an important book about design patterns [GHJ+95]. An architect generates designs from experience, whereas design pattern development involves generating experience from designs.

2.5 Domain Specific Software Architectures (DSSA)

A domain-specific software architecture (DSSA) is [Arm93]:

- A standard software architecture constructed for a domain, or family, of applications.
- A specification for assemblage of software components that are
    - specialized for a particular class of tasks (domain)
    - generalized for effective use across that domain
    - composed in a standardized structure (topology)
    - effective for building applications
- The high-level packaging structure of functions and data, their interfaces and control, to support the implementation of applications in a domain.

Producing specific applications for recurring problems is costly and inefficient. The objective of the DSSA approach is to develop families of related systems. DSSA tries to identify a general solution within a domain. The general solution is to provide a software architecture, a specific set of tools and a set of components for reuse in other applications of the same domain. Software applications are developed out of DSSA on an industrial basis by using component-programming concepts. Effective use of the DSSA approach is dependent on effective tools, environment and process support.

The effective use of DSSA results in savings in life cycle costs. However, the costs for training, reusable architectures and tools is not insignificant and must also be taken into consideration. Profitability is achieved when the architecture is reused on new applications in the same domain. Generally, to pay for the overhead for designing, documenting and supporting the DSSA, at least three systems must be generated from the architecture [TTC95]. Lockheed Martin came to the same result and found that 'Biggerstaff's rule of three and empirical evidence gathered to date indicate that at least 3 systems must be generated from the architecture before the overhead for designing, documenting and supporting the DSSA is paid for.' [Tra95]

A domain-specific software architecture (DSSA) consists of the following elements (see [HPL+95]):

- reference architecture - describing a general computational framework for a domain of applications
- component library - containing reusable chunks of domain expertise
- application configuration method - used for selecting and configuring components to meet particular application requirements

Major contributions to research and development of DSSA was done by the DARPA (Defense Advanced Research Projects Agency) Domain Specific Software Architecture Program. The five-year program started in 1991 with industrial, military and academically partners. Industrial partners were Lockheed Martin, Honeywell, Teknowledge, GTE and others. Honeywell estimates that '… where a reusable architecture can be rigorously specified and supported by tools, reductions in cost and schedule by factors of 3 to 10 should be achievable (overall life cycle effort should be reduced to 10-33% of current levels).' [Hon96].

The first step in DSSA is to analyze the application domain and hence develop the domain model. The analysis should involve customer inputs and inputs from experts of the domain. The purpose of the domain model is to provide an explicit understanding of the various aspects of the domain.

The domain model consists of:

- scenarios
- domain dictionary
- context diagram
- entity/relationship diagrams
- data flow models
- object model

In the context of DSSA, the reference requirements drive the design of the reference architecture. They specify the functional requirements and also the non-functional requirements. The architecture is designed to be reusable, extendable and configurable. The reference architecture should provide information for application developers who support the generation of new applications and the modification of existing applications.

The DSSA approach promises increases in productivity, quality and reliability. Productivity increase is achieved through reuse and automation. The quality is increased because new systems are constructed from well-proven architectures and components.

2.6 Product Lines

"A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission." [CN99]. Following the product line approach, a common set of core assets is used. These core assets (e.g. predefined components) allows the building or instantiation of members of a product line.

Software architecture codifies structural commonality among members of a program family so that high-level design decisions inherent in each member of a program family need not be re-invented, re-validated, and re-described [CN96]. The software architecture representing structural commonality among members of a program family is called *reference architecture*. A reference architecture works as a template for a family of systems.

There are a number of reasons why it is useful to investigate into software product-line architectures. A product-line architecture

- enables software developers to respond quickly to changing requirements
- supports the rapid introduction of new products or releases
- reduces the complexity of code development and maintenance
- improves the quality of new products or releases
- reuses information, experience and concepts of well-proven legacy systems
- supports the sharing of product knowledge
- enhances the reliability because the reusable parts have been successfully used in systems before
- supports the activities for producing documentation and training material

The key of successful product line management is the effective use of commonalties across the product line. Systems are no longer programmed line-by-line, but rather by composing well-tested components. Commonality management is however not only about reusing components or other software products, but also the reuse of case studies, documentation, schedules and budgets, maintenance plans and also the software architecture.

An example of commonality management of product lines is FAST (Family-oriented Abstraction, Specification and Translation) [Wei98]. FAST was developed at Lucent Technologies as part of the domain engineering process. FAST supports the developer in gaining confidence that he or she is building a family that is worth building and that the family is appropriate to the market. The FAST commonality analysis is performed in a series of meetings where a Commonality Analysis Document is produced. The document contains information about the dictionary of terms (key technical terms of the domain), commonalities (list of assumptions that are valid for all family members) and parameters of variation (quantifies the variability). The Commonality Analysis Document is used as the basis for family architecture work (e.g. design work), as reference documentation, as basis for reengineering a family or as a plan for evolution of the family.

A well-designed software architecture which is applicable across the product line is essential. Reusable components must be designed in such a way that they are easily tailor able to changing requirements. The concept of reference architectures (or sometimes called product line architectures) was developed to support the construction of product lines. Reference architecture allows for the sharing of a common software architecture across a product line.

2.6.1 Reference Architecture

In many application domains an area of repetitive problems can be identified. Many of these repetitive problems can be solved by standardized, component-based software.

A reference architecture (often also called a product line architecture) is a "description of the structural properties for building a group of related systems (i.e. product line), typically the components and their interrelationships. The guidelines about the use of components must capture the means for handling variability discovered in the domain analysis or known to experts." [CN99]

In reference architectures, a set of design constraints and conventions is defined. The constraints and conventions dictate such things as global system organization and provide a common vocabulary of design elements (such as parsers and protocol layers) [Gar95].

Advantages of the use of reference architectures [DKO+97]:

- provides the development team with product line knowledge
- reduces the complexity in developing and maintaining code
- streamlines the production of documentation

A reference architecture represents generic architectural information that is valid for all members of a product family. Recognition of an architectural style in a domain is the basis of (at least part of) a reference architecture for the domain [TTC95].

A reference architecture supports communication between system shareholders and the architectural decision process. The architecture of new family members can be derived from the reference architecture. The advantage is that new family members will be built on well-proven architectural components and domain specific styles. By using such shared structures the costs and the time effort for building additional family systems will be reduced. In this case the reusable components are generalized versions of components.

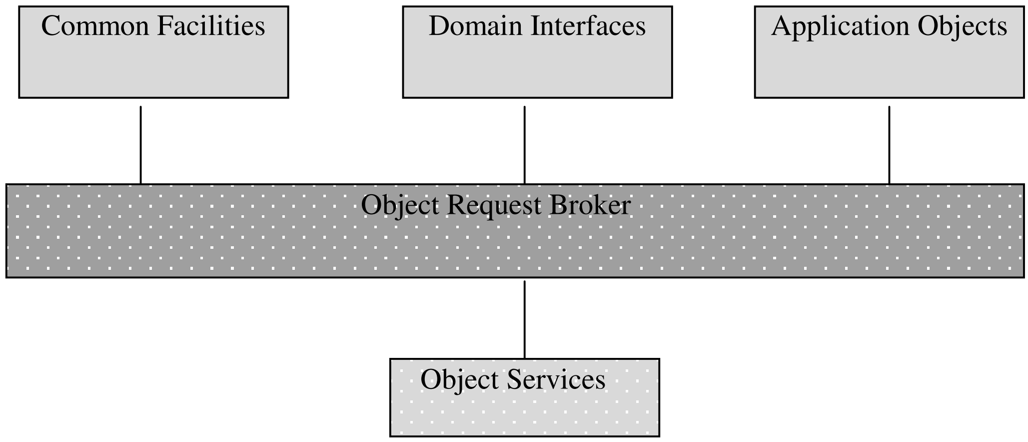


Figure 2: Reference OMG Architecture [Sta98]

Figure 2 shows one of the most well-known reference architectures that exists today – the Object Management Architecture (OMA). The Object Management Group (OMG) provides an infrastructure for the interaction of objects in a heterogeneous domain. Therefore, OMG defined a reference architecture called the Object Management Architecture (OMA). The central component in OMA is the Object Request Broker that, for example transmits method calls from client to server objects. The Object Request Broker is standardized and currently available as CORBA 2.0 (Common Object Request Broker Architecture) [Sta98]. Similar concepts are found in Microsoft's DCOM and Java RMI.

3. Overview and Comparison of Architecture Description Languages

A large number of ADLs have been proposed for modeling software architectures. Examples are ACME [GMW97], Aesop [GAO94], AML [Wil99], C2 [MOR+96], Darwin [NKM96], Meta-H [Ves96], Rapide [RDT97], UniCon [SDK+95] and WRIGHT [All97]. Most of the languages have been developed in research institutions or universities. None of the languages is widely used in the industry and currently no ADL can be identified that is a candidate for a standard ADL.

For describing the results of an architecture recovery process, it is necessary to have a supporting ADL at hand. The requirements for ADLs specifically for product line recovery are described in the following chapter.

3.1.1 ADL requirements

Requirements for ADLs supporting description of software architectures of product lines:

Architecture Properties – The focus on components and connectors is not enough for complete description of software architectures. ADLs must also provide notations for specification of property information. Architectural properties describe specific architecture-oriented aspects of a system. The recovery and reasoning of architectural properties is an important aspect for the development of reference architectures. Examples of architectural properties are safety and data communication.

Timelines - Real-time systems contain performance requirements. Real-time systems must not only produce correct but also timely results. “The correctness of real-time systems depends not only on the logical correctness of the results, but on the time when they are produced as well.” [AGP98] In the case of hard real time systems, a missed deadline can lead to an erroneous computation and a system failure. In many other domains the time aspect is not of such importance as in real-time systems. Therefore ADLs defined for real-time systems must be much more focused on time-aspects than other systems. Performance requirements must already be taken into consideration in the stage of architectural design. It is therefore necessary that timelines can be described by the ADL of choice.

Safety Aspects - For development of failure detection or fault-tolerant systems, diversified software are used. Several versions of a program (component) from the same specification is developed and executed in (semi) parallel. A voter acts as a comparator and compares the results of the several versions to ensure safety requirements. Safety requirements are already important on the architectural level and must therefore be supported by the ADL.

Commonality analysis – In the architecture recovery process it is important to get support for the commonality analysis that identifies the reference architecture of a system. Recovery of product line architectures is based on product line member architectures. Even for a product line with a relatively small number of members, the amount of recovery data usually is very high. A commonality analysis can only be based on recovery data that are available in a structured manner.

Expressiveness - An ADL must provide notations that allow the architects to completely specify system vocabulary and mental model. An ADL must provide notations that make it simple for the architect to translate the mental model into the ADL.

Some requirements for ADLs are contradicting. ADLs should on one hand be simple to understand, supported by a graphical syntax and for general purpose but should on the other hand be supported by a formally defined semantic that allows different types of analysis and model checking. ADLs should contain information about different aspects of an architecture such as the structure, the control structure, and communication protocols but at the same time not be overloaded with information that complicates understanding and analysis.

3.1.2 Existing ADL surveys

A number of ADL surveys exist:

- Medvidovic and Taylor [MT00], [Med96] classified and compared following ADLs: Aesop, MetaH, LILEANNA, ArTek, C2, Rapide, WRIGHT , UniCon, Darwin and SADL.
- Cattaneo [Cat98] provides a brief evaluation of Aesop, WRIGHT , Darwin, Rapide and UniCon.
- Kogut [KC95] and Clements [Cle96a], [KC95] analyzed the following ADLs: ArTek, CODE, Demeter, Modechart, PSDL/CAPS, Resolve, UniCon and WRIGHT .
- Vestal [Ves93] compared four ADLs: LILEAnna, MetaH, μRapide and QAD.

None of the surveys concentrates on the specific requirements for architecture recovery and description of real-time systems and product families. It was therefore necessary for this work to compare ADLs with focus a on real-time and product family aspects and to identify an ADL supporting the description of product families of real-time systems.

3.1.3 Survey of four ADLs

The following sub-chapters give an overview of four ADLs. The languages have been chosen representing prominent languages (Rapide, WRIGHT) product-line supporting languages (Darwin) and real-time specific languages (ROOM). Section 3.2 to 3.5 give an introduction to the examined ADLs. In section 3.6 the comparison criteria are introduced and the four ADLs are compared according to the criteria.

3.2 DARWIN

Darwin is a declarative architecture description language developed at Imperial College London [NKM96], [MDK94], [JRL00]. Darwin evolved from Conic, a configuration language designed for distributed systems. Darwin has a graphical and a textual syntax. The textual representation has a richer semantic than the graphical syntax that allows the specification of conditionals and iterations. Conditionals and iterations are evaluated at binding-time.

In Darwin, a system is represented in terms of component types and services that the components provide or require. The Darwin specification of a system is built as a hierarchical composition of components that forms a tree. Composite component types are constructed from primitive or composite component types. Primitive components are the leaves in the component tree and are the only components in the system encapsulating behavioral aspects. All other components provide only structural aspects. Configurations are built by creating instances of component types and by binding services together. The system hierarchy is flattened into a collection of concurrently executing computational component instances at execution time.

Component interfaces are either services that the component provides or services that the component requires. Components in Darwin are black-boxes without any information about internal component information such as the behavior.

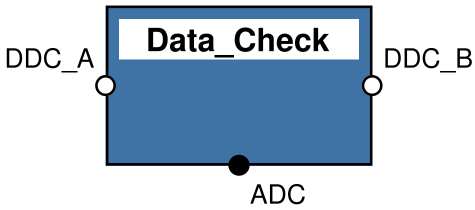


Figure 3: Component *Data_Check*

Figure 3 shows the component type *Data_Check* represented in Darwin. Components have interfaces which represent the services provided (*ADC*) and services required (*DDC_A, DDC_B*). Filled-in circles represent services provided by a component, empty circles represent services required by a component and gray colored circles represent services without clear definition if the service is required or provided. The name of the services is local to the component type specification.

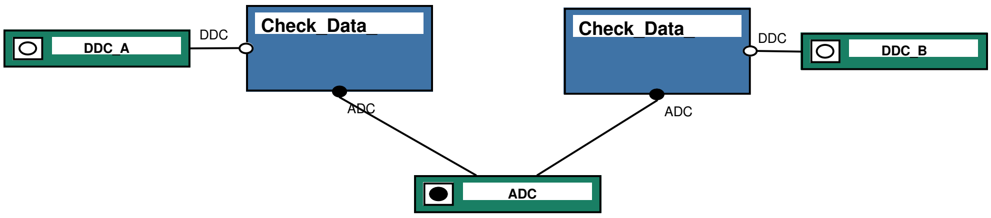


Figure 4: Two sub-components of component *Data_Check*

Component *Data_Check* is a composite component that consists of two sub-components. Figure 2 describes the two sub-components of component *Data_Check*. *Check_Data_A* and *Check_Data_B* represent n-version components which have the same input through input port *ADC*. The components are safety-critical components that implement the same functionality, but for safety reasons, the components are designed and implemented by different teams. The output (*DDC*) from both components are through the *DDC_A* and *DDC_B* ports of the component *Data_Check*.

```
①    component Check_Data(int hardware_platforms = 2)
     {
             provide
                     DDC <stream, s_type>;
             requir e
                     ADC <stream, a_type>;
     }

②    component Data_Check
     {
             inst
                     check_A : Check_Data(hardware_platform);
                     check_B : Check_Data(hardware_platform);
③            when hardware_platform == 1
             inst
                     mlink_client : Mlink_Client;
④            provide
                     DDC_A@ SharedMemory(F42A);
                     DDC_B@ SharedMemory(F42B);
⑤            require
                     ADC;
⑥            bind
                     check_A.ADC – ADC;
                     check_B.ADC – ADC;
                     DDC_A – check_A.DDC;
                     DDC_B – check_B.DDC;
⑦            when hardware_platform == 2
             provide
```

```
            DDC_A;
            DDC_B;
        require
            ADC;
        bind
            check_A.ADC – ADC;
            check_B.ADC   – ADC;
            DDC_A – check_A.DDC;
            DDC_B – check_B.DDC;
}
```

Figure 5 - Darwin code of the examples in figure 4.

① Start of component type definition. The parameter *hardware_platform* specifies the number of hardware platforms. The service *DDC* is defined which provides information to the environment and the service *ADC* which requires information. The type of the services is defined in angle brackets. The communication mechanism used to implement the services is *stream*; the datatype communicated is a user defined type (*s_type, a_type*).

② Declares two instances of the component type *Check_Data*.

③ In case the condition is true, the following definition is valid.

④ Describes services that the component provides to its environment. Additional information in the Darwin source code can be inserted as tag-information (indicated by the @ symbol). SAA supports the definition of tags but the tag information can not be displayed with the current version of the tool.

⑤ Describes services that the component requires from its environment.

⑥ Defines the binding between services. The service *check_A.ADC* is bound to the interface requirement *ADC* which is a service on the higher component level.

⑦ In case the condition is true, the following action is valid.

Darwin provides semantics for its structural aspects by using the π-calculus, which is Milner's calculus of mobile processes [MDE+95]. The π-calculus models component interaction and composition. Services and the binding are interpreted as agents. Components are represented by agents, which are the composition of the services. The binding corresponds to the composition of agents. The π-calculus provides the semantic for checking if a DARWIN program correctly specifies the component instances and bindings at runtime. The π-calculus as used in DARWIN only supports modeling the semantics of composite components and not the modeling of semantics of primitive components [Med96].

Provided services are modeled as channel names. Each binding declaration is a process that transmits the name of that channel to a component that requires the service. Binding of services is only valid if the connecting service types match. The SAA (Software Architect Assistant) tool performs such compatibility checks. Currently the tool performs only a simple name equivalence test.

Darwin allows the definition of composite interfaces. Composite interfaces represent a grouping of related services to a single interface. Composite interfaces are an abstraction mechanism that simplifies the large number of services of large systems.

Connectors connect components via connecting component services. The connector protocol is pre-defined by the component services since connectors connect *provide* and *require* services. The connector protocol is not independent of the connecting components.

Connectors represent a flow of information without specifying the type of flow (data or control flow or both). Additional connector information, such as the type of information flow (e.g. synchronous, HW information), must be recorded with the tag mechanism.

Darwin allows the definition of *template architectures* by defining generic types. Placeholders can be defined that are replaced at build-time with real component types. A further feature of Darwin is the possibility to build *partially defined component types*. Some of the parameters of a partially defined component type are defined at an early development stage. In a later development stage the missing parameters may be defined and the system specification completed. Partially defined component types allow new component types to be derived from a general component by adapting the values of its parameters. This is useful in the development of product families as it provides a simple mechanism for deriving many different, more specialized family instances from a generic family description.

DARWIN provides two views, the structural view and the service view. The structural view represents components and interfaces. The service view allows a portal to be either specified as a service provided or a service required by the component. The structural and service views contain complementary information. The service view is a refinement of the structural view. Together, they describe the static structure of an architecture. The service view leads towards an implementation by providing CORBA connection. Interfaces can be specified in CORBA IDL and primitive components can be implemented as CORBA objects. Other views such as the behavioral view and performance view are not directly supported by Darwin but can be attached to the basic structural view.

Labeled Transition Systems (LTS) were also developed at Imperial College and can be used for behavioral description and analysis. LTS is based on state machine concepts and describes all the states a component can reach and all the transitions that may be performed. LTS is supported by the *Labelled Transition System Analyzer* (LTSA) which is a verification tool for concurrent systems.

The *Software Architect's Assistant* (SAA) is a tool which supports the graphical input and editing of Darwin. The SAA allows the conversion from the graphical to the textual representation and vice-versa. SAA is also a syntax checker, controlling Darwin models for syntactical correctness.

Regis [MDK94] is another tool developed by Imperial College. Regis is a programming environment and distributed platform supporting the execution of distributed programs. Service type information is interpreted by *Regis*. If CORBA is used as the distributed platform, the service type defines an IDL specification.

3.3 RAPIDE

Rapide [LKA+95][LV95][Luc96][RDT97] is an event-based, concurrent, object-oriented architecture definition language. Rapide provides constructs for building architectural prototypes that can be executed for simulation. Formal constraints are provided that support constraint-based definition of reference architectures. The constraints also support the testing of systems for conformance to reference architectures and architecture standards.

The building blocks of architectures defined in Rapide are a set of specifications (called interfaces) of modules, a set of connection rules defining communication between interfaces and a set of formal constraints defining legal and/or illegal patterns of communication. The architecture itself is an executable specification of a class of systems.

Rapide represents a set of languages called the *RAPIDE language framework*. The framework consists of the Types, Patterns, Architecture, Constraint, and Reactive Programming languages:

- types language - used for describing the interfaces of components
- pattern language - used for describing patterns of events
- architecture language - used for describing the flow of events between components
- constraint language - used for describing abstract constraints on component behaviour
- reactive programming language - used for writing executable components

The motivation for dividing the language into sublanguages is to simplify changes to the language and to facilitate reuse of parts of RAPIDE in other languages [RDT97].

The *architecture* language is a sub-language of the *reactive programming* and the *pattern* language.

```
① type Producer(Max : Positive) is interface
②     action out Send(N : Integer);
      action in Reply(N : Integer);
③ behavior
Start => Send (0);
(?X in Integer) Reply(?X) where ?X <Max => Send(?X +1);
end Producer;

① type Consumer is interface
②     action in Receive(N : Integer);
      action out Ack(N : Integer);
③ behavior
      (?X in Integer) Receive(?X) => Ack(?X);
end Consumer;

④ architecture ProdCon() return SomeType is
      Prod : Producer(100);
      Cons : Consumer;
⑤ connect
      (?n in Integer)
      Prod.Send(?n) => Cons.Receive(?n);
      Cons.Ack(?n) => Prod.Reply(?n);
end architecture ProdCon;
```

Figure 6 An Interface Connection Architecture with asynchronous connections [Luc96]

Figure 6 shows an example of an interface connection architecture described in RAPIDE.

① Definitions of two interface types, *Producer* and *Consumer* , which define the interfaces of producer and consumer objects. Flexible modification of architectures is supported by dynamic subtype substitution. The interfaces in this example define asynchronous communication features called actions.

② The *Producer* and *Consumer* interfaces contain, respectively, two actions. In the case of the *Producer,* the two actions are the **action out** *Send* and the **action in** , *Reply* . Objects of this type can generate the event *Send* and can receive the event *Reply* . Rapide specifies what components require from their environment and what they provide to the environment.

③ The interfaces also contain reactive behavior transition rules. In the case of *Producer,* the first rule triggers on a *Start* event and reacts by generating an out event, Send(0).

The second one triggers on *Reply* events containing integer data under the condition that the data is less than ‘Max’. A Send event is then generated containing the integer plus 1. In the case of *Consumer* the rule indicates that a *Receive* event can be received and it reacts by generating an *Ack* event containing the same data.

④ This section describes the ProdCon architecture. The architecture contains two components, *Prod* and *Con*, each of an interface type.

⑤ The *connect* section contains two reactive connection rules. The first rule connects the **action out** *Send* of the *Prod* component with the **action in** *Receive* of the *Cons* component. In case *Prod* generates a *Send,* the system reacts by generating a *Receive* event of *Cons* with the same data. The second rule connects the **action out** *Ack* from *Cons* with the **action in** *Reply* of *Prod*.

The interfaces (①) are defined in the RAPIDE types languages whereas the architecture declaration (④) is defined in the RAPIDE architecture language.

Rapide explicitly represents dependencies between events with the partially ordered event set (poset) execution model. Components send and receive events. Events consist of an instance of an event type together with one or more values. A poset can be seen as a set of linear sequences of the same set of events. An event poset captures dependencies or possible ordering of events. Posets show which event caused which other events and which events happened independently. Causality and timing of events form a partial order of events. An architecture simulation is the set of events that are generated together with the timing and the causal history. Constructs modeling asynchronous communication between modules are called *actions* and those modeling synchronous communication are called *functions*.

Other languages for concurrent event-based simulation do not encode dependency between events [LKA+95]. They usually specify events without causal relationships. Rapide on the other hand defines patterns of events with causality and timing relationships. Events in Rapide abstract away computations that are not of interest on the architectural level.

Conformance rules in Rapide determine which modules may be plugged into interfaces. When the module is plugged into the interface, the module becomes an executable component and the behavior of the interfaces becomes a constraint on the component behavior.

Rapide evolved from VHDL (for event-based and architecture concepts), C++ (for type systems), and TSL (for event patterns and formal constraints on concurrent behavior) [LKA+95]. VHDL (**V**HSIC **H**ardware **D**escription **L**anguage) is a language designed for describing electronic systems. The development of VHDL started with the VHSIC (**V**ery **H**igh **S**peed **I**ntegrated **C**ircuit) project of the US Department of Defense. VHDL was accepted as a standard by IEEE in December 1987. VHDL has influenced Rapide especially by adopting event patterns for triggering processes.

Since VHDL is used in modeling HW systems it would by interesting to see if Rapide could be used for HW-SW co-design. No evidence could be found in the literature about experiences in this area.

Event pattern mappings in Rapide define how a system is related to a reference architecture by mapping of executions of an instance architecture into executions of a reference architecture. The conformance to the reference architecture constraints can be checked in Rapide.

Rapide is an event processing language, and therefore behavior-oriented, but it does not give information about architecture design decisions (e.g. global error control).

Rapide supports the building of executable models. A set of use cases (or scenarios) can be used to execute an architecture. The execution results in posets that show timing and performance properties of the architecture. The architectures are simulated by the Rapide compiler/runtime system.

3.4 UML-RT / ROOM

ROOM (Real-Time Object-Oriented Modeling language) [BCK98], [SR98], [Smi97] is a visual modelling language with formal semantics. It is designed for specifying complex, event-driven, and distributed real-time systems. ROOM appears to be representative of the current generation of ADL tools and is widely and successfully used in the telecommunication industry [BCK98].

UML (Unified Modeling Language) [Dou98], [FS97] is a general-purpose modelling language that unifies the methods of Booch, Rumbaugh (OMT), and Jacobson. UML was standardized by the OMG (Object Management Group) by the end of 1997. UML provides a set of general purpose modelling language concepts and may be used to model systems across domains.

UML-RT (UML for Real-Time) [SR98], [Lyo98] is a real-time modelling method that combines UML 1.1 and ROOM modelling concepts. The intention of UML-RT is to support developers of complex real-time software systems. This opens very interesting aspects for future directions of ADLs specifically for real-time systems. UML-RT has been co-developed two tool vendors (ObjecTime, Rational Corporation) and has professional tool support. This is not the case for most of the other ADLs. UML‘s standard tailoring mechanism (stereotypes, tagged values, constraints) were used to build a library of specific UML concepts. These concepts can specifically be used in combination with the other UML concepts for modelling architectures of complex real-time systems.

UML-RT combines UML, role modelling, and ROOM concepts. Role modelling is a technique that facilitates the capturing of structural communication patterns between software components. A role model in UML is a specialization of a class model and a generalization of an instance model. Properties of an application, valid for all instances of the application, are captured in a role model. The role model shows the roles classes play and how they interact. UML provides a specific diagram called a collaboration diagram for role modelling.

Selic and Rumbaugh [SR98] describe how constructs originally specified in ROOM can be specified using UML. No new UML modelling concepts were needed to describe ROOM constructs. Only UML tailoring mechanisms as stereotypes, tagged values and constraints were used. UML *capsules* correspond to *actors* used in ROOM. UML interfaces represent behaviour without information about the implementation structure.
A ROOM port implements behavioural and structural information. In UML-RT a port is modelled by the *port* stereotype, which is a UML class stereotype. The class related to the port implements the protocol defined for the port. The UML *realizes relationship* (dashed line with solid triangular arrowhead) is used to represent this relationship.
ROOM bindings are modelled as UML connectors. An UML connector is declared through an association role in a collaboration diagram. No specific extension is needed to model ROOM bindings in UML-RT.

Egyed and Gacek [EG99] investigated ways of describing and identifying causes of architectural mismatches across (UML) views and components. A view integration framework was introduced which included a set of activities and techniques for identifying mismatches. The view integration framework is used to validate new information added by a user versus the system model. This ensures the conceptual integrity.

Three stereotypes have been introduced in UML-RT that are specific for modelling real-time systems. Stereotypes in UML are used to refine the semantics of a UML metaclass. The three stereotypes introduced in UML-RT are *Capsule* , *Port*, and *Connector* . UML-RT collaboration diagrams, use capsules, ports and connectors to specify the structure of software components. Collaboration diagrams are also used to capture architectural design patterns

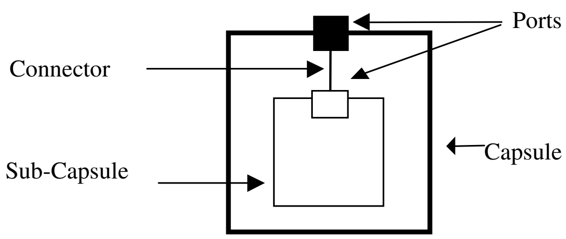


Figure 7: UML-RT capsule collaboration diagram [Lyo98]

Capsules are architectural components in UML-RT that correspond to the ROOM concept of *actors* . The functionality of capsules is described in a state diagram. Capsules can be decomposed into sub-capsules. In collaboration diagrams the decomposition of a capsule class is described. Capsules communicate with the environment through ports, which are signal-based architectural components.

A **port, which** corresponds to ROOM *port,* is an object that implements a specific interface. Ports allow one to export multiple different interfaces from the capsule. Each interface plays a specific role. Ports provide protocols that define the valid flow of signals between the connected ports (the ports are mutually complementary). The implementation of ports de-couples internal capsule implementations from requirements of the environment and are therefore reusable. Ports are instances of a protocol class and may be either *relay ports* or *end ports* .

- *Relay ports* pass all messages through without delay. Relay ports allow the flow of messages through the port of an actor (capsule) to a sub-actor (sub-capsules). For the environment, it looks like the actor is providing the port and not the sub-actor. The state machine of the actor is not interfered by signals flowing through the relay port.

- *End ports* are access points to the behavioral model of an actor. The actors' state machine executes messages arriving at end ports. Messages sent by actors are produced by the actors' state machine. End ports are sources and sinks of all messages sent from and received by actors. End ports may either be *public end ports* or *protected end ports* . Public end ports are boundary objects of the encapsulating actor and the state machine. Protected end ports are ports of sub-actors of the encapsulating actor. The difference to the relay port is that protected end ports are used by the state machine of the encapsulating actor.

Connectors correspond to ROOM *bindings* and realize the communication relationships between capsules. A connector supports a specific signal-based protocol. Connectors can only interconnect ports with complementary protocol roles.

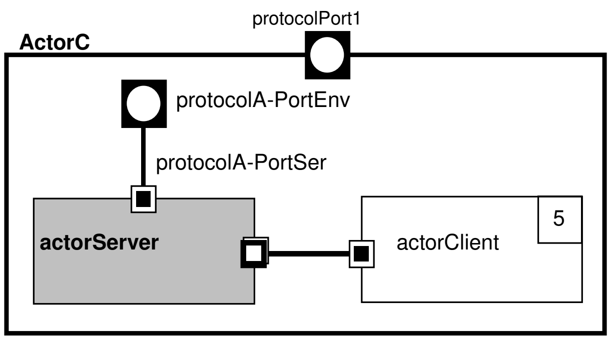


Figure 8: Example of ROOM notation [SR98]

Figure 8 shows an example of ROOM notation. *ActorC* has 2 subactors, *actorServer* and *actorClient* . *protocolPort1* is an end port while *protocolA-PortEnv* is an internal port. The end port is used for communication between *ActorC* and its environment. The internal port allows communication between the encapsulating actor *ActorC* and the sub-actors, in this case *actorServer* . The port *protocolA-PortSer* of the actorClient is an example of a conjugated port.

UML provides a number of diagrams which are of special interest for modeling software architectures:

- Class diagram which presents important abstractions in a system and how they relate to each other.
- Use Case diagram which gives a step-by-step description of how the system reacts to input stimuli.
- Interaction diagrams which represent sequences/behavior between system entities. UML provides three interaction diagrams:
    - State diagram representing the behavior of one object/class.
    - Component diagram representing the mapping between the logical element in the model and the implementation artifact.
    - Deployment diagram representing processors, devices, and connections between them.

Structural aspects in UML-RT are modeled with class and collaboration diagrams. Behavior is described by extended, hierarchical, finite state machines (similar to SDL state machines [BH93]).

3.5 WRIGHT

WRIGHT (the name refers to the architect Frank Lloyd Wright) is an architectural description language based on the formal description of the abstract behavior of architectural components and connectors [All97] p.9. WRIGHT [All97], [ADG98], [AGI98] describes architecture configurations (components, interactions between components) and architectural styles. WRIGHT distinguishes between component and connector types and their instances.

Component - WRIGHT component descriptions specify *interfaces* and *computation*. The abstract behavior of components is described using a CSP-like notation as part of the connector description.

> *Interface* - WRIGHT components have interfaces called ports. A component interface consists of a number of ports. Ports support the structure analysis of the component interaction. A port is a partial specification of a component and represents an interaction in which the component may participate. The port defines a local protocol with which the component interacts with its environment.
>
> *Computation* - The computation shows how ports of a component are tied together and describes how interactions are carried out. The computation provides the entire specification of the component. Ports do not contain redundant information about the computation.

Connector - A WRIGHT connector represents an interaction among a collection of components. The description of a connector is divided into a set of *roles* and the *glue*. WRIGHT makes connector types explicit to achieve two purposes: (a) applicability of analysis, (b) independence of components. WRIGHT permit describing connector constraints by defining protocols of interaction. Other ADLs allow constraints via interfaces and semantics [Med96], p. 20. A connector in WRIGHT consists of a set of roles and the glue.

> *Role* - The role defines the behavior of one participant in the interaction. A connector description consists of two roles; if a connector represents a procedure call, the roles are caller and definer. Roles specify the protocol that is expected from the connecting component ports. A port can choose to use only a subset of the connectors capabilities.
>
> *Glue* - The glue of a connector defines how the connector roles will interact with each other. By using the descriptions of connectors it can be determined if a connector satisfies properties, e.g. if the communicating components behave accordingly to the communication protocol.

WRIGHT focuses strongly on connectors and is thereby distinguished from other ADLs. Descriptions of connectors are used to determine whether the connectors satisfy critical properties. Such properties are the internal consistency of the protocol and the conformance to the interface specification. WRIGHT places no constraints on components so that any combination of components can be used as long as the connector constraints are fulfilled. Connectors in WRIGHT are therefore component independent and reusable.

WRIGHT uses a CSP-like (**C**ommunicating **S**equential **P**rocesses) notation for specifying computation, port, role and glue. WRIGHT uses a subset of CSP and extends this subset in three syntactic ways [AGI98]:

(1) The CSP variant distinguishes between initiating events and observing events. Initiated events and observed events are used to visualize how components control interactions.
(2) The CSP variant uses a termination symbol (“§”) to visualize the successful termination of a process.
(3) The CSP variant uses a quantification operator for constructing new processes combining the process expression parts with the operator.

CSP uses operators for sequencing (“ →”, “;”), alternative (“ □”, ” []”) and parallel composition (||). Alternatives can be described using internal (deterministic) or external (non-deterministic) choice. Internal choice ( ) and external choice ([]) specify the controlling element. Parallel composition specifies parallel processes that ‘may interact by jointly engaging in events that lie within the intersection of their alphabets.’ [AGI98].

Style - WRIGHT supports the formalization of an architectural style and provides the description of a family of systems. A WRIGHT style has two parts, the vocabulary and the constraints on configurations. The definition of components and connectors is the vocabulary. Constraints define style predicates that must be true for a system instance. The architectural configuration is specified through attachments of ports to roles. The style guarantees that the architecture will have a set of desirable properties.

① **Style** Client-Server
② **Component** $\overline{\text{Client}}$
  Port p = $\overline{\text{request}}$ → reply → p □§
  Computation = internalCompute → $\overline{\text{p.request}}$ → p.reply → **Computation** □
§
  Component Server
  Port p = request → $\overline{\text{reply}}$ → p [] §
  Computation = p.request → internalCompute → $\overline{\text{p.reply}}$ → Computation [] §

③ **Connector** $\overline{\text{Link}}$
  Role c = $\overline{\text{request}}$ → reply → □c §
  Role s = request → $\overline{\text{reply}}$ → s [] §
  Glue = c.request → $\overline{\text{s.request}}$ → **Glue**
    [] s.reply → $\overline{\text{c.reply}}$ → **Glue**
    [] §
④ **Constraints**
$\exists!$ s ∈ Component ∀ c ∈ Component : TypeServer(s) ∧ TypeClient(c) ⇒ connected(c,s)
EndStyle

⑤ **Configuration** Simple
  Style Client-Server
  Instances C : Client ; L : Link ; S : Server
  Attachments C.p **as** L.c ; S.p **as** L.s
EndConfiguration

Figure 9 Client-Server System described with WRIGHT [ADG98]

① WRIGHT defines a system by declaring a set of components and connector types, in this case termed as ***Style*** *Client-Server* .

② The component *Client* has one port named *p*. The component *Client* makes a request through the port and waits for the reply on port p or it terminates (§). The $\overline{\text{request}}$ event is an initiated event marked by an overbar. Initiated events and observed events are used to visualize how components control interactions. In the example, the Client initiates requests ($\overline{\text{request}}$) while the server observes the request (request).

The internal choice used in the *Port* definition of the *Client* indicates that the *Client* decides whether it makes a request or terminates. The external choice used in the

Port definition of the *Server* indicates that the *Server* is expected to respond to any number of requests. In this case the *Server* may not terminate prematurely.

The component *Client* iteratively makes a request and waits for a reply on port *p* or the component terminates successfully. Successful termination is indicated by the termination symbol (§).

③ The connector role *c* is identical to the port *p* of the *Client* (and also the role *c* to the port *p* of the *Server*). The port could also choose to use only a subset of the connector's capabilities. The glue specification describes the way the roles of the connector interact with each other. A client request must be transmitted to the server. The server reply must be transmitted back to the client.

④ The constraints define a set of predicates that ensure the correct connection of server types and client types.

⑤ The configuration describes the instances of components and connectors and how they are assembled to form the specific architecture.

Parameterization is used to expand WRIGHT descriptions from single systems to families of systems. Any part of a type description (e.g. type of a port or role) in WRIGHT can be temporarily defined with a placeholder. The placeholder is replaced with a parameter when the type is instantiated. Parameterization can also be done by 'number'. For example, if the architect wishes to define a client - server system in which the number of clients is unknown at definition time, the architect can define the server number parameter as a range of integer ("<min>..<max>"). The number must however be defined at instantiation time and can not be changed during system execution. In WRIGHT it is not possible to define dynamic architectures where components are created dynamically or the configuration is changed at runtime. This restriction is based on the fact that CSP can only describe a static configuration of processes.

3.6 ADL comparison

In sections 3.2 to 3.5, four ADLs (Darwin, Rapide, ROOM, UML-RT, Wright) were introduced. In this section, the four ADLs are compared with focus the on the ability of the ADLs to describe and analyze product lines of real-time systems (see 3.1.1 for a description of requirements for ADLs).

A number of sources were reviewed for the language comparison (e.g. [NKM96] [RDT97] [Lyo98][All97]). Basic terms found in documentation differ sometimes. The following table shows terms found in the language documentation and the mapping to the terms used in this comparison.

Terms & Concepts	**Darwin**	**Rapide**	**ROOM**	**UML-RT**	**WRIGHT**
Component	component	component	actor	Capsule	component
Interface	service	constituent / interface	port	port role	port
Connector	binding	connection	binding	Connector	connector

Table 1: Mapping of language terms and concepts

3.6.1 Component

Does the ADL support the modeling of abstract components as types?

All languages allow the modeling of abstract components as types.

Darwin, Rapide and WRIGHT also allow the parameterization of component instances. Darwin allows one to build *partially defined component types* that are defined at an early development stage and specified in a later development stage. Rapide uses *constructors* and adapted event patterns for parameterization. In WRIGHT, parameterization of components is possible by specifying the computation specification and the number of component ports.

Is it possible to represent specific component properties (e.g. component roles, safety-critical component)[2]? Is there any support for representing timing information of the component?

Darwin, Rapide and WRIGHT do not provide specific concepts for the definition of properties. Darwin provides tags which can contain any textual information and can be added as a comment to the Darwin textual representation. Tag information is however not part of the languages but only free text. ROOM (UML-RT) has a number of standard tailoring mechanisms - stereotypes, tagged values, constraints. The tailoring mechanisms were used to build a library of specific UML concepts. Stereotyped metaclasses, for example, are derived from existing UML metaclasses and are just like 'basic' UML metaclasses, but are specialized and/or extended. Sterotypes may be used to build specific component properties.

[2] Two ADLs which are outside of the scope of this survey, with strong support for non-functional properties are MetaH and UniCon [Med96] p.27

3.6.2 Interface

Are interfaces defined independent of components? Independent interfaces support de-coupling of implementation and definition and increases of reusability. If independent interface are allowed, the behavior of the component must be represented by other means than service representation.

Darwin provides a strong coupling between the interfaces (services) and the components because interfaces represent the complete component functionality to the environment.

In Rapide, conformance rules determine which modules may be plugged into interfaces. Only after the module is plugged into the interface does the module become an executable component and the behavior of the interfaces a constraint on the component behavior. Therefore Rapide supports the de-coupling of interfaces and components.

In ROOM (UML-RT), the collection of all interfaces (ROOM ports) of a component (ROOM actor) defines its type. In ROOM, components and interfaces cannot be defined independently.

WRIGHT components provide interfaces called ports which support the structure analysis of the component interaction. A port defines a local protocol with which the component interacts with its environment. Therefore WRIGHT provides a strong coupling between a component and its interfaces.

Are interfaces defined independent of connectors?

Darwin connectors are created by the *bind* construct that links a *provide service* of one component with a *require service* of another component. The connectors are therefore not independent of the interfaces.

In Rapide, interfaces are connected by connecting rules. Rapide builds a strong connection between interfaces and connectors (as explained in criteria before). Interfaces can therefore not be defined independently from connectors.

ROOM (UML-RT) interfaces (ports) are connected by *bindings* . Interfaces with compatible protocols can be linked together. ROOM interfaces define the protocols whereas the connectors link the protocols. The connectors are independent of the component interfaces.

A WRIGHT connector represents an interaction among a collection of components. WRIGHT makes connector types explicit to achieve independence of components (and their interfaces).

Is the representation of specific properties (like timing properties) supported?

Darwin allows the definition of additional information by *tags*, which is free text without specific property information. Darwin does not provide specific properties.

Rapide allows the definition of connecting rules as a kind of property of the communication. However, Rapide does not provide any flexible concepts for defining specific properties, such as timing properties.

In ROOM (UML-RT), information that cannot be described by ROOM language concepts may be described by using programming languages (like C or C++). The information described by programming languages becomes part of the ROOM model and finally forms an executable model. ROOM provides therefore a concept for complete description but on a low-level of abstraction which is not suitable for the purpose of architectural description.

WRIGHT uses CSP for describing properties of connectors but without providing a flexible property concept for describing specific properties. Timing properties for example cannot be described in WRIGHT .

3.6.3 Connector

Do the connectors provide protocol specifications, e.g. rules about types of interfaces the connector is able to be connected to?

Darwin and Rapide connectors are simple bindings without protocol specifications.

In ROOM (UML-RT), interfaces (called *ports*) define protocols. Ports are instances of a protocol class. Only ports with compatible protocols can be linked together. The connector (binding) in ROOM does not provide protocol specification; only the interfaces provide protocol information.

A connector in WRIGHT is defined by a set of roles and glues. The glue of a connector defines how the connector roles will interact with each other. Roles in WRIGHT specify the protocol that is expected from the connecting component ports.

Does the connector abstract away complex protocols and make them reusable by defining connectors as types?

In Darwin, ROOM (UML-RT) and Rapide connectors are not defined explicitly but rather modeled in-line (cannot be subtyped, reused or named). The connectors of the languages contain no connector semantics, complex connectors may not be defined as connector components and connectors are not defined as a connector type.

WRIGHT models connectors as first-class entities and distinguishes connectors types from connector instances. WRIGHT bases the connector type on protocols defined by using a CSP-like language.

3.6.4 Configuration

What communication patterns are used to represent communication between components in the configuration?

In Darwin, the type of an interface (a service in Darwin) defines the interaction mechanism used to implement the service, e.g. a port. The information about the interaction mechanism is not interpreted by Darwin but by the underlying distributed platform. If Regis [MDK94] is the platform, this information is used to directly select the correct communication code. Regis provides a number of predefined communication classes and allows the users to define their own interaction mechanism.

Rapide is an event-based architecture description language. A component can communicate with another component by generating events which trigger connection rules in the architecture. The architecture executes the connection rules which results in calls to the other component's interface functions. The basic communication paradigm is asynchronous event passing.

ROOM (UML-RT) is based on the concept of hierarchical extended finite state machines, based on Harel's state-diagram principles. Details about event handling are not represented graphically in ROOM but must be represented by using programming languages. Code generated out of the graphically specified model is compiled together with state transition code (with runtime-system functionality) written by tool vendors for simulation or execution.

WRIGHT uses a CSP-like notation for modeling connector types as abstract patterns of interaction. The connector types define discrete, asynchronous component actions which may be extended to other interaction types, such as synchronous communication.

Is it possible to define constraints on architectural elements (e.g. n-version related constraints as the definition of n-versions of a component in a safety-critical element)?

All of the surveyed languages implicitly define constrains. One such constraint is the restriction that a component may only be accessed through its interfaces.

Darwin provides only implicit constraints, e.g. the binding of services is only valid if the connecting service types match. Tags provide a concept of defining implicit constraints that are not part of the language, but rather plain text.

Rapide provides formal constraints that support constraint-based definition of reference architectures (through Rapide's timed poset constraint language). The constraints also support the testing of systems for conformance to reference architectures and architectural standards. A building block of architectures defined in Rapide is set of formal constraints defining legal and/or illegal patterns of communication.

ROOM (UML-RT) and WRIGHT establish constraints by defining interaction protocols. WRIGHT puts even more constraints on connectors by defining protocols for each role. Neither language provides a flexible concept for defining constraints on architectural elements.

How is hierarchical composition solved?

Components in Darwin are either primitive or composite. Primitive components have a behavioral specification and composite components have structural specifications. Together they represent the hierarchical composition.

In Rapide, *maps* relate interfaces to architectures and provide hierarchical information.

In ROOM (UML-RT), stereotypes for capsules are used for representing major architectural elements. The relation of capsules to sub-capsules is indicated by composition relationships in a class diagram.

WRIGHT 's definition language CSP does not support hierarchical composition. The composition of components is represented indirectly by the glue of a composite connector.

How is the internal component semantics (necessary for consistency analysis, enforcement of constraints) implemented?

In Darwin, the internal component semantics is represented with the π-calculus. A Darwin model is a collection of independent processes which communicate via named channels. The π-calculus models component interaction and composition which can check if a Darwin program correctly specifies the component instances and bindings at runtime.

Rapide implements internal component semantics by introducing partially ordered event sets (posets). Posets allow the description of sequential and concurrent behavior. Posets utilize an algebraic language to specify constraints on the abstract state of a component.

ROOM (UML-RT) is based on object-oriented principles and represents behavior by extended finite state machine concepts.

WRIGHT specifies protocols of interaction for each port by specification of event traces. WRIGHT does not directly represent component semantics but represents this information in interfaces.

3.6.5 Product Line

Does the ADL support the description of specific architecture styles (e.g. pipe-filter style)?

Darwin allows the definition of template architectures by defining generic types. Placeholders can be defined that are replaced at build-time with real component types. Because of this, Darwin allows the definition of any architectural style.

Rapide, ROOM (UML-RT) and WRIGHT provide no features that allow the description of a specific architectural style. All of the languages provide features that allow the definition of any architectural style. Very few of the currently existing ADLs support a specific style. An example is C2, which is designed for definition of graphical user interface intensive applications.

Is it possible to define dependencies between the configuration of family members?

In Darwin, the template architecture or the partially defined component types are defined at an early development stage. In a later development stage the missing parameters may be defined and the system specification completed. Through parameterization the defined model loses the dependency to its template architecture.

Rapide provides formal constraints that support constraint-based definition of reference architectures. The constraints also support the testing of systems for conformance to reference architectures and architecture standards.

ROOM (UML-RT) provides no concept for defining dependencies between the configuration of family members. However, by defining stereotypes of existing relations between components, a dependency relation between the configuration of family members may be defined.

WRIGHT supports the formalizing of an architectural style and for this reason can describe a family of systems. The architectural configuration is specified through attachments of ports to roles. The dependency relation is part of the language.

3.7 Comparison summary

The four ADLs are similar in the sense that they model architectures around the same building blocks (component, connector, interface, configuration). Each of the languages has its special characteristics and provides special features.

Darwin is a simple, yet powerful ADL. Darwin allows the definition of interesting concepts like *template architectures* or *partially defined component types*. The π-calculus provides the semantic for checking if a DARWIN program correctly specifies the component instances and bindings at runtime. The SAA (Software Architect Assistant) supports the graphical input and performs compatibility checks. The programming environment and distributed platform Regis supports the execution of distributed programs.

Rapide provides constructs for building architectural prototypes that can be executed for simulation. Rapide represents a set of languages called the *RAPIDE language framework*. Rapide explicitly represents dependencies between events with the partially ordered event set (poset) execution model. An event poset captures dependencies or possible ordering of events. Posets show which event caused which other events and which events happened independently.

UML-RT (UML for Real-Time) is a real-time modelling method, that combines UML and ROOM modelling concepts with the intention of supporting developers in modelling complex real-time software systems. Stereotypes in UML are used to refine the semantics of a UML metaclass. Three stereotypes have been introduces in UML-RT that are specific for modelling real-time systems. UML is standardized and widely used as a software engineering method supported by a number of UML tools from different vendors. UML-RT has been co-developed by two major tool vendors.

WRIGHT focuses strongly on connectors and is for this reason distinct from other ADLs. Descriptions of connectors are used to determine whether the connectors satisfy critical properties (e.g. internal consistency of the protocol). WRIGHT uses a CSP-like notation for specifying computation, port, role and glue. WRIGHT supports the formalization of an architectural style and can therefore describe a family of systems.

The comparison of ADLs was undertaken to identify an ADL that could support the description of the result of an architecture recovery process of a product line. All of the examined languages provide some concepts for modeling product families. However, none of the languages provides a set of concepts that support the description of a real-time product family. In addition, none of the languages provides features that support a commonality analysis of product family members. None of the languages provides a flexible concept for modeling architectural properties.

As a result of this comparison, it became clear that none of the four languages would be able to fulfill the requirements for modeling product families of real-time systems.

Other ADL surveys were analyzed (e.g. Medvidovic and Taylor [Med96], Kogut [KC95], Clements [Cle96a], [KC95], Vestal [Ves93]), but no other ADL could be identified that could fulfill the requirements for modeling product families of real-time systems.

It was therefore necessary to develop a specific ADL that could provide features to describe product families, properties of architectural elements and become an integral part of a commonality analysis. This language is called ASDL and is described in detail in chapter 4.

4. SARM (Software Architecture Recovery Methodology)

To effectively use the assets of legacy systems, it is important to develop a systematic strategy that supports the continued evolution of these systems. This is the basis of meeting changing technology and user requirements. However, architecture-related information (and other information) of legacy systems is often embedded in source code, design documentation and the minds of system experts, but is not explicitly known or documented. Architecture-related information evolves in an ad-hoc manner.

All medium and large-sized software systems have an inherent software-architecture, but the recovery of such information is difficult after many years of development, evolution, and changes in the project personnel.

In some cases, systems do have documented architectures but the architectural description is out of sync because of missing maintenance of the architectural description. One reason for this is the difficulty of integrating software architecture-related activities into the software development process.

4.1.1 Architectural recovery

Specific methods are necessary that support the process of software architecture recovery, which is the opposite process to the forward program development process.

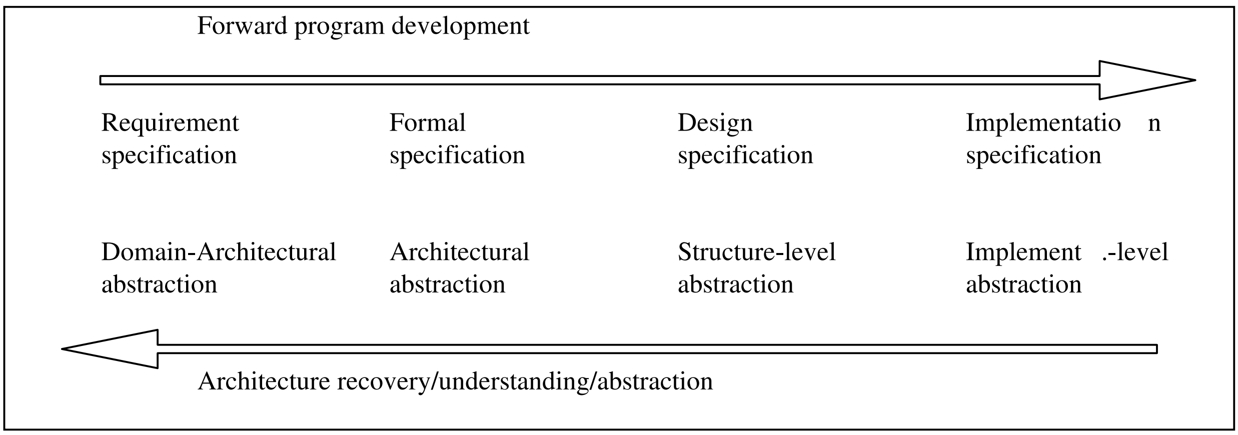


Figure 10 - Forward program development & architectural abstraction

Figure 10 presents the relation between forward engineering and reverse architecting [HN90]. Reverse engineering is reverse forward engineering [BM97].

SARM is a methodology that was specifically designed for reverse architecting [JRL00],[EOG+98],[Eix98]. SARM uses information of members of a family of legacy software systems and transforms it in a stepwise approach into a reference architectural representation. As explained in *2.4.2 Architectural Recovery* a number of approaches have been developed for meeting the goals of reverse architecting. SARM is one approach that differs from other approaches by providing *language integration*, *integration of input views* and *family architecture recovery*.

- Language integration - SARM is integrated with ASDL, the ADL introduced in this dissertation (see chapter 5). ASDL is used for description of architectural information recovered in each of the recovery steps of SARM. The results of each SARM step can directly be described by using ASDL. Especially in the commonalty analysis the ASDL description is input to process defined in SARM. Other recovery methodologies do not provide this integration of language and recovery process.

- Integration of input views - SARM is a generic approach and accepts any view of a legacy system as input to the recovery process. Many other architecture recovery methodologies accept only source code as an input medium, which is insufficient [WCK99]. Architectural information is then recovered by analyzing source code and extraction of recovery information. SARM accepts source code as input as well as any other view of information (as documents, models, notations and human knowledge).

- Family architecture recovery - SARM provides a mechanism that extracts general architectural information from family members. SARM may be used to recover the architecture of a single system but also provides commonality analysis concepts for the recovery of the reference architecture of a family of systems. SARM guides the reverse architect through the entire recovery process from input definition and selection to the description of the reference architecture.

4.1.2 Levels of abstraction

Information about software systems can be represented on different levels of abstraction. Figure 11 gives an overview of the different levels of abstraction.

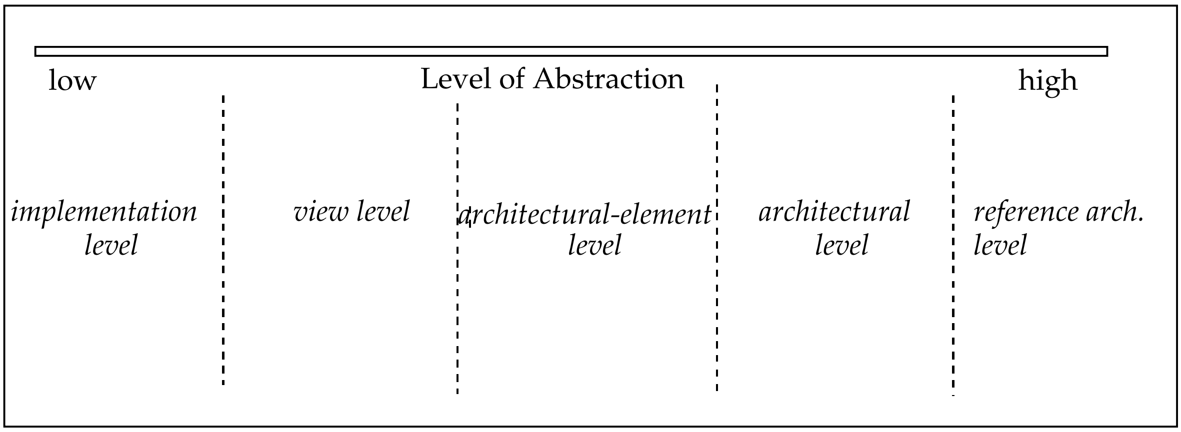


Figure 11: Abstraction levels

The lowest level of abstraction is on the *implementation level* and the highest level of abstraction on the *reference architecture level*.

- **Implementation level** – Detailed information about a legacy system can be found on this level of abstraction. Typical artifacts on this level are source code and detailed design documentation, e.g. function descriptions.

- **View level** – Design documentation used by programmers for implementation purposes can be found on this level. Call graphs, cross-reference tables, object models and data flow diagrams are typical representatives of such information. These artifacts are typically used as design views in the forward engineering process.

- **Architectural-element level** – Architectural information (architectural elements and architectural properties) can be found on this level of abstraction. The information is however not represented by a complete architectural description (e.g. block line diagrams or lists of components).

  Implementation and view level are input levels to architecture recovery and software comprehension. Architectural-element level is the first level of abstraction, where the directions of the processes are divided.
  - Architectural recovery focuses on the gross structure, the form and the content of the systems. On this level of abstraction, architectural information may be identified by grouping entities together and be searching for patterns. The result are candidates for architectural elements and properties that may be described in an Architecture Description Language.
  - Software comprehension (and software maintenance) is "to fully understand something, or in other words, to be fully aware of the meaning of

something"[Mit00]. Also for software comprehension the information on this level of abstraction may be grouped. However not to identify the gross structure but to establish a link between something currently under observation and a conceptual model already present in the observers mind [Mit00].

- **Architectural level** – This is the highest level of abstraction of a single system and provides complete and consistent information about the architecture of a system including architectural elements and architectural properties. Typically an ADL (Architecture Description Language) is used to represent the architecture.

- **Reference architectural level** - On this level, reference architecture information is represented. Repetitive aspects of a family systems as generic components are represented. ADLs (Architecture Description Language) are also typically used to represent the architectural information on this level.

4.2 The 7 steps of SARM

SARM provides 7 steps that lead from legacy system input to the reference architecture of a system:

Step 1:	Prepare input for recovery process
Step 2:	Extract structural architecture (components, interfaces and connectors), e.g. by extracting higher-level views (call graphs, data-flow diagrams) with tool support and/or manually
Step 3:	Extract architectural properties (safety, timing constraints)
Step 4:	Describe the architecture (structural architecture & properties) with an ADL

Steps 1 to 4 recover and describe the architecture of a single system and are repeated for every member of a product family.

Step 5:	Extract the structural reference architecture from two or more structural architectures of family members
Step 6:	Extract reference architecture properties
Step 7:	Describe the reference architecture (structural architecture & properties) with an ADL

Steps 5 to 7 recover the reference architecture of a family of products.

Step 1 is a preparation step that prepares information in the *implementation level* for the recovery process. Steps 2, 3 and 6 are transformation steps that transform information from the *implementation level* to the *view level,* the *architectural element level* and the *reference architectural level.* Steps 4 and 7 are description steps that describe architectural information on the *architectural* and the *reference architectural level*.

Figure 12 gives an overview of SARM for a single system that is input for the family system recovery process [EWK+96]. Source code and other input information is on a low level of abstraction. Architectural information cannot be directly extracted from such abstractions. The 7 steps of SARM raise the level of abstraction in a stepwise approach until architectural information is available.

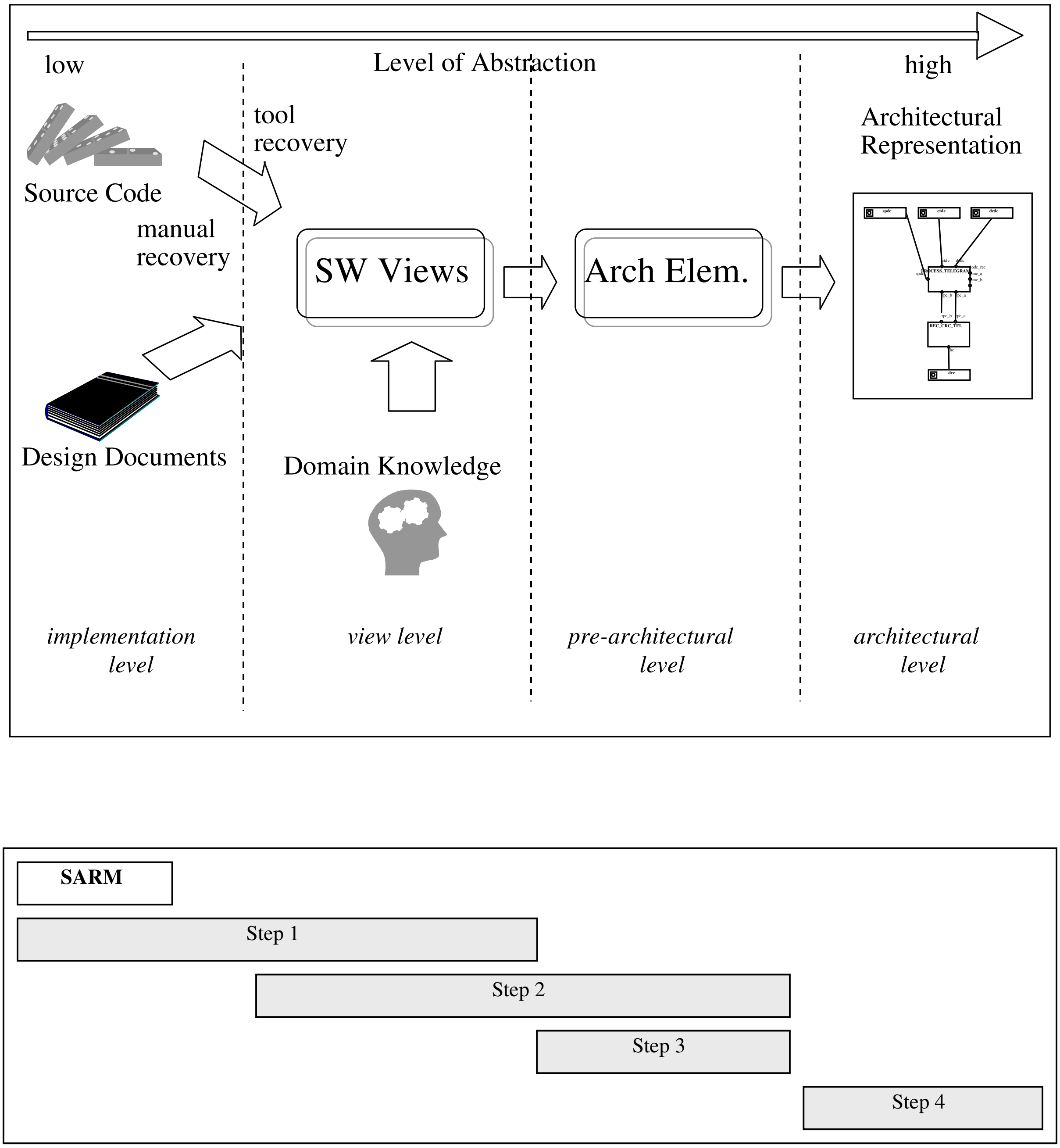


Figure 12: Outline of SARM for a single system

SARM provides a step-wise approach that raises the level of abstraction step-by-step until it reaches the level of architectural abstraction. By raising the level of abstraction, information that is not of interest to the architecture of the system is filtered out (e.g. implementation details, low-level design information).

The 7 steps are described in more detail in section 4.2.1 to 4.2.7 . The following section gives a short overview.

Step 1: Prepare Input

In step 1, all input information is gathered and prepared for the recovery process. The input for the architecture recovery process is in most cases source code (legacy code), design documentation and domain knowledge that must be prepared for the recovery process.

Non-relevant information such as information not directly related to the system under consideration is filtered out (e.g. outdated information). This step is a preparation step that provides input to the following steps. No architectural information is identified in this step.

Step 2: Extract structural architecture

In step 2, higher-level views such as call graphs are extracted from legacy code. In this step, information that is not related to the architecture of a system is filtered out (e.g. implementation details and low-level design information) and the level of abstraction is raised from source code to a higher level, the view level.

In case the legacy code is written in a widely used programming language, tool-support for view extraction is often available. Software reengineering tools perform typically static analysis on source code and allow no human interaction. The results of such analysis are different views on the source code, e.g. call diagrams, data flow diagrams and cross reference tables. In some cases (e.g. when the legacy code is written in more than one programming language), the tools reach their limits. Manually-generated views then provide information that can be recovered by tools.

Structural architecture elements (components, interfaces and connectors) are identified by examination of the design documentation and software views. A number of techniques supporting architectural element recovery are available.

Additional architectural information (architectural properties) are in most cases strongly connected to the structural architecture and are identified in step 3.

Step 3: Extract architectural properties

In step 3, additional information (architectural properties) of the software architecture is recovered. Depending on the system domain, different architectural properties are of interest. In the case of real-time systems, such properties are timing information or safety information. In the case of business systems, properties of interest may be security and response time.

After deciding on which architectural properties are to recover, the complying recovery technique must be chosen and applied on the input set. Architectural properties are in most cases hidden in source code and design information and may therefore be recovered manually.

Information recovered in steps 2 and 3 is now available in an unstructured manner. Step 4 solves this problem by transforming architectural information into a comprehensive representation.

Step 4: Describe the architecture

In step 4, the recovered information from steps 2 and 3 is described by using a formal description technique. Software architectures can for example be represented by ADLs (Architecture Description Languages), diagrammatic syntax [DC95] or specification languages as such Z [AG92]. The type of representation must be chosen depending upon specific requirements (e.g. the system domain) and is in principle independent of SARM. The chosen description technique must however be able to represent architectural elements recovered by SARM, otherwise information will be lost. In case the reference architecture is to be recovered, ASDL (as described in chapter *5 ASDL*) should be used for architectural description. ASDL provides concepts that are directly used for reference architecture recovery.

Step 4 provides architectural information of a single system in a consistent and complete way. System shareholders can use the descriptions for communication purposes and architectural assessment.

Repeat step 1 to 4:

Steps 1 to 4 are performed for each member of the system family. The result is a set of architectural descriptions of family members.

All further steps of SARM are used to identify the reference architecture of a system family. The entire SARM including steps (6) and (7) is designed for a rather small number of family members. The upper limit on family members is dependent on the overall number of components and connectors. The upper limit may be around five to eight family members. If more family members are involved in the recovery process, the complexity of the identification of the reference architecture would be too high. In that case, SARM should be redesigned and adapted.

Step 5: Extract the structural reference architecture

In step 5, the reference architecture of the system family is identified.

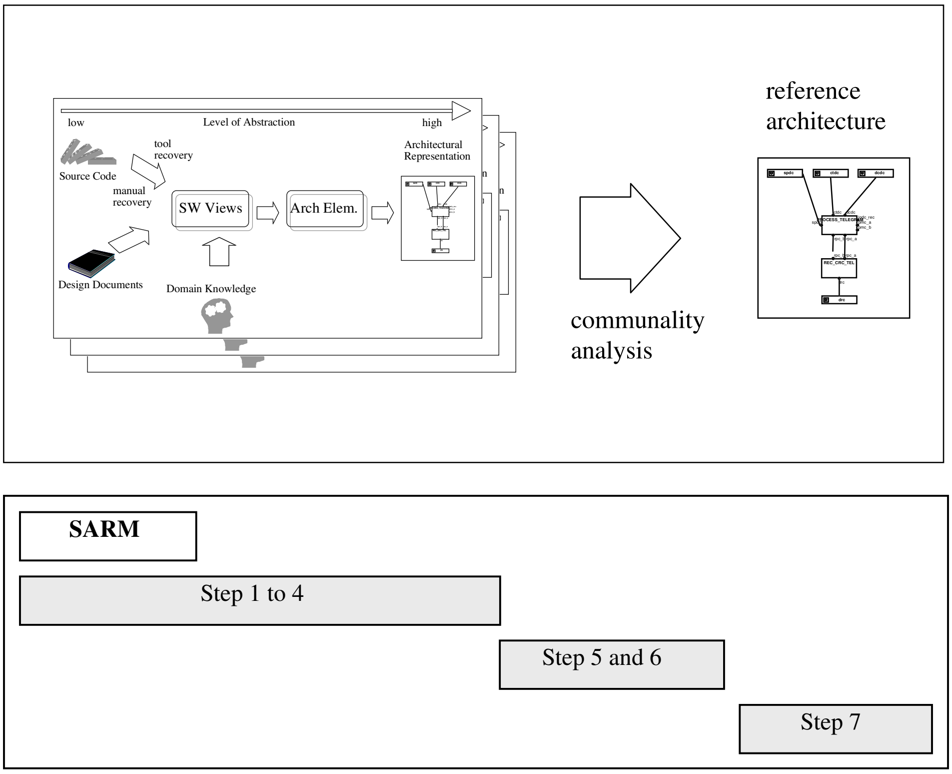


Figure 13 overview of entire SARM process

Figure 13 provides information about the entire SARM process. By analyzing the architectures of a number of family members, the general or reference architecture of the system is identified.

The reference architecture is extracted by using the SARM commonality analysis process. Input for this process comes from the recovered architectures of family members. Since the number of architectural elements may be rather high, a semi-formal method must be applied to handle the complexity. The SARM commonality analysis is such a semi-formal method that has been developed for identifying reference architectures. In a selection process, similar architectural elements are selected and grouped into sets of common elements. By analyzing the sets of common elements, the reference architecture is derived. Both the ASDL and the commonality analysis are flexible and may be adapted to specific domains.

The resulting reference architecture is described in an ADL, in most cases by using the same technique as in step 4.

The result of step 5 is the structural reference architecture of the system family under consideration. The structural view will be completed by step 6, in which system properties are identified.

Step 6: Extract reference architecture properties

In step 6, the information about the reference architecture is completed with system properties (e.g. overall safety system). System properties consist of single member properties identified in step 4 and family properties that are specifically identified in step 6.

As in Step 5, a selection process is used to identify similar architectural properties and group them into a set of common elements. By analyzing the sets of common properties, the reference architecture properties are derived.

The result of step 6 is a set of system properties.

Step 7: Describe the reference architecture

In step 7, the reference architecture is described. The result of step 7 is the complete description of the reference architecture (*structural reference architecture* and *reference architecture properties*) of the system family under consideration. The reference architecture is now available for system shareholders and system architects.

As in *Step 4 Describe the architecture,* the chosen description technique must be able to represent architectural elements recovered by SARM, otherwise information will be lost. The usage of specific languages such as ASDL (Architecture Structure Description Language) may be necessary because no other language is able to describe the system in a satisfying manner.

Step 7 concludes the processes initiated by SARM by providing the reference architecture in a consistent and complete manner.

4.2.1 Step 1: Prepare Input

Input	- Textual representations - Graphical representations - Source code - Human knowledge
Method	- Information selection and preparation
Output	- Filtered software views and human knowledge

4.2.1.1 Introduction

In step 1 of the recovery process, all input information is gathered and prepared for the architecture recovery process. Information that is irrelevant to the system under consideration is filtered out.

4.2.1.2 Input

Input to the recovery process may be available in different representations. Architectural information may be represented in different views such as source code, written documentation or information in the head of a system architect. “A view represents a partial aspect of a software architecture that shows specific properties of a software system.” [BMR+96]. For software architecture recovery activities, software views are documents, source code files or human knowledge containing information about a software system that can be used for reengineering and re-architecturing activities.

The following describes typical views in detail:

- **Textual representations -** All software development phases produce results in textual form. The result is a wide variety of documents - requirement specification, analysis documents, system specification, design documents, quality assurance plan, test specification and others. In some cases not all of these documents may exist or the documents may be outdated. Variance among different members of a product family at the code level and the documentation level is of special interest for recovery of the reference architecture. Documents that are not concentrating on one particular product but on a family of products are of great help because they express the commonalties between a family of programs. These commonalties reflect the architectural basis that has been established by the system designers.

- **Graphical representations -** Graphical representations, often found in combination with textual representation, are a tool for complexity management and for communication purposes. Examples of graphical representations are entity-relationship diagrams, domain models, uses-hierarchy charts, state charts, object models and component diagrams (block-line diagrams).

- **Source code -** In many cases, source code contains the only reliable, accurate, and current documentation of the software and the software architecture. In some systems, the ‘as-implemented’ architecture is not the same as the ‘as-designed’ architecture. The re-architect’s task is to identify such mismatches based on the source code. Source code contains information about design and implementation decisions. The information in source code is at a very detailed level and contains design and implementation-related information that is of no interest to the software architecture.

- **Human knowledge -** System shareholders, especially system architects, system designers and programmers are a valuable source of architectural information. Software developers build their mental model of the system that may or may not be the same as the 'official' architecture and design model. In many cases, developers have their own mental model in mind when talking about the system under consideration. The recovery of architectural information from human experts is therefore difficult.

4.2.1.3 Method

Different methods are used for the preparation of each of the views.

4.2.1.3.1 Step 1.1: Preparation of textual representations

- Use the development process description to identify what kinds of documents are typically produced. Collect the documentation and order it per topic.
- Select documents that are of help for the architecture recovery process. Documents covering the system specification and early design phase are probably the documents with the most architecture-related information.
- Identify state of documents. Remove outdated documents (documents that not longer represent the as-is system).
- Find documentation that describes the vocabulary used under system development. Such documentation may be of help in the component identification phase.

4.2.1.3.2 Step 1.2: Preparation of graphical representations

The method for the handling of graphical representations is similar to the method for textual representation. However, since the syntax and semantic of graphical representations may be outdated or incomplete specific actions are necessary.

- Use the development process description to identify what kinds of graphical representations are typically produced. Collect the representations.
- Select representations that are of help for the architecture recovery process. Representations covering the system specification and early design phase are probably the representations with the most architecture-related information.
- Identify the state of representations. Remove outdated representations (representations that not longer represent the as-is system).
- In case non-standard notations are used, identify:
    - What is the meaning of symbols (does a block mean that the element is a software element or a hardware element or both)?
    - What is the relation between diagrams? Are they conflicting or complementary?
    - Is there an underlying semantic in the diagrams? For example, SDL (Specification and Description Language [BH93]) is a language used for

specification of embedded software and is based on a specific semantic model. How did the semantic influence the design process?

4.2.1.3.3 Step 1.3: Preparation of source code

The information in source code is on a very detailed level. Source code contains design and implementation-related information that is of no interest for the software architecture recovery process. The most important factor for handling source code is to filter out irrelevant information.

Already at the level of medium-sized programs (100 kLOC or more), program comprehension is a problem for the re-architect. Tools and strategies that support program understanding and program comprehension are available for major programming languages. Program comprehension strategies are bottom-up, top-down, knowledge-based, systematic and as-needed, integrated approaches [SWM97]. SARM does not focus on program comprehension, but rather presupposes that extensive program documentation or system programmers is available.

- Identify the correct source code version, including all necessary files and directories.
- Identify the code architecture of the system. The distribution of source code over directories and files is the code architecture of a system. Code architectures also include relations between source files such as build-up with configuration files, make files and 'include' relationships.
- If reverse-engineering tools are used prepare, the source code such that tools can perform the source code parsing. Typical preparation actions are:
  - Conversion of the file format of source files from development platforms to the tool platform (e.g. DOS to UNIX).
  - Conversion of source file names (e.g. from upper-case character to lower-case character)
  - Acquisition of platform specific files (e.g. header files as dos.h)
  - Exclusion of files that are for test or simulation purposes
  - Usage of the correct compiler flags that represent the correct system version

4.2.1.3.4 Step 1.4: Preparation of human knowledge

System experts are indispensable for architecture model verification and for collection of architecture property information. Using a questionnaire is an appropriate way to structure the information collection process, which is most often an iterative process.

- Make certain that system experts for the recovery process are identified and available for supporting the recovery process
- Inform the system experts about the advances of software architecture recovery to gain a positive attitude for the recovery process
- Ensure that the system experts are familiar with the different recovery steps and the objectives of the recovery project

4.2.1.4 Output

Filtered views
Software views (textual representations, graphical representations, source code) of the system under consideration necessary for the architecture recovery process are identified and available. Outdated views and irrelevant views are filtered out.

Human knowledge
System experts for the system under consideration are identified, informed about the project and are available to support the architecture recovery process

4.2.2 Step 2: Extract structural architecture

Input	- Filtered software views and human knowledge
Method	- Tool-supported high-level view generation - Manually generated high-level views - Identification of structural elements
Output	- Structural architecture elements

4.2.2.1 Introduction

Step (2) raises the level of abstraction by producing higher-level views such as call graphs or data flow diagrams from legacy code or other input sources. The higher-level views are either produced manually or supported by a tool. From higher-level views, the structural architecture (components, connectors) are identified and non-architecture-related structural information (implementation details and low-level design information) is filtered out.

4.2.2.2 Input

The input comes from filtered software views and human knowledge identified in step 1.

4.2.2.3 Method

Different methods are used for the identification of the structural architecture:
- Tool-supported high-level view generation
- Manually generated high-level views
- Identification of structural elements

4.2.2.3.1 Step 2.1: Tool-supported high-level view generation

Tools are available that support the extraction of views from legacy code. Tool support is of high-importance because of the size and complexity of industrial software systems.

Reverse engineering tools typically perform three steps for recovering higher-level views ([Bel97]):

- Extraction - An internal representation is extracted from source code by a parser. The result is typically an abstract syntax tree (AST).
- Abstraction - The representation from the extraction step is stored in a repository. The representation can be manipulated and further abstracted. Most reverse engineering tools provide an application-programming interface (API) that allows the user to define applications for repository data manipulation.
- Presentation - Results of extraction and abstraction are presented to the user via high-level views generated by the reengineering tool. The views are either textual or graphical (2 or 3-dimensional).

Software reengineering tools typically perform static analysis on source code and do not allow human interaction. The results of such analysis are different views on the source code, e.g. call diagrams, data flow diagrams and cross reference tables. Tool generated diagrams are often confusingly large, unstructured and are difficult to read for human users. The views must therefore be manually edited by filtering, structuring and symbol rearranging.

4.2.2.3.2 Step 2.2: Manually generated high-level views

Reverse engineering of other software views than source code must be done by human experts and results in manually-generated high-level views. Manually-generated views contain additional information that cannot be recovered by tools. Human interaction is necessary for view creation, view transformation and view enrichment.

- View creation – Views that cannot be generated by tools may be generated by human experts. An example is a data flow diagram that represents the flow of specifically interesting data through the system where the data is manipulated by different languages and stored in shared memories. Current state-of-the-art reverse engineering tools are not able to create such views.

- View transformation – As explained above, views generated by tools may be difficult for human readers to understand. In a view transformation process, the tool-recovered views are transformed into views readable and editable by humans.

- View enrichment - Views generated by tools or views produced by software designers lack domain-specific information. In the following process, humans enrich these views with domain-specific information, e.g. safety information.

Manual view generation is a costly process. Therefore, the number of manually generated views will be rather low. Re-architects must identify missing concepts that are important for architecture recovery and that are not covered by tool-generated views.

As with tool-supported view generation, there are several steps for manual view generation:

- Selection - The re-architect has to decide what type of information has to be recovered and what views are the most promising for completion of higher-level views (selection criteria's). These may be additional views (view creation) or specialized views (view transformation and enrichment). In data-flow-oriented systems, it may be interesting to represent the most critical flow of data, in terms of performance, through the system. The view then contains a special perspective of an architecturally relevant aspect that was not recovered by tools.

- Extraction – The re-architect has to analyze the input views (source code, textual and graphical representation) for the selection criteria. Tools (e.g. source code browser or pattern recognition tools) may support the process of view extraction. The extraction process is time-intensive and costly.
- Presentation – Depending on type and amount of the recovered information, the re-architect has to present it in according views. Similar notations and symbols used for tool generated views should be utilized to produce matching diagrams. An important advantage of manually recovered views is the fact that the presentations are after recovery already in a form that is easy for humans to read. Manually recovered views are typically not overloaded. Additional filtering is therefore not necessary.

Views are an important set of information for architecture description and recovery. "Architectural views play the same role in dealing with architectural information as data views play in database management systems."[JRL00] section 1.9. Kruchten [Kru95] developed an architectural model, called 4+1 View Model. The different views organize architectural descriptions into following views: logical view (similar to an object model), process view (represents concurrency and synchronization), the physical view (describes the mapping of the software onto the hardware), the development view (static organization of the development environment) and a "collaboration" view (use-cases and scenarios that keep the other four view together.

4.2.2.3.3 Step 2.3: Identification of structural elements

The recovery of structural elements (components, component interfaces and connectors) is one of the most important steps in the SARM approach. Components, component interfaces and connectors form structural architecture information. The identification of components, component interfaces and connectors is central to the architecture recovery process.

Additional architectural information such as architecture properties is directly related to architectural elements. The non-structural information (properties, rationales) is directly related to structural information, but not the other way round. The recovery of architectural properties is described in step (4).

Component Identification

Components are independent units of computation that interact through interfaces with the environment. Detailed information about software architecture components can be found in section *2.3.1 Component* .

A typical architectural design activity is the partitioning of the overall system into smaller units (into composite and primitive components) rather than thinking about the details of modules [GJM91] p.6. The software architect makes an explicit or implicit decision about the partitioning algorithm. The re-architect on the other hand can group source elements into architectural elements in an enormous variety of ways [KC97]. For the recovery of components it is therefore essential to know the basics of the used decomposition algorithm to perform a correct recovery.

Decomposition technique - Wasserman [Was96] p.26 identified several partitioning approaches for architectural partitioning/decomposition:

- modular decomposition - based primarily on assigning functions to modules
- data-oriented decomposition - based on external data structures
- event-oriented decomposition - based on events that the system must handle
- outside-in design - based on user inputs to the system
- object-oriented design - based on classes of objects and their interrelationships

Such approaches are rarely used as stand-alone approaches but rather used together in the design phase of a system. For example, systems based on the SDL-92 [BH93] extended finite state machine approach incorporate event-oriented decomposition with object-oriented design.

The information about which approach was used for system partitioning is essential for recovery of components. If modular decomposition was used, the recovery process will be based on identifying functions and modules. If data-oriented decomposition was used, the recovery process will be based on the identification of data structures and so on.

The partitioning approach can be identified by
- asking system architects
- reading design documentation
- analyzing views (tool-generated views, manually generated views)

Recovery approach - A number of possible component identification approaches are considered available [EOG+98]. This list is far from being complete. The approaches are chosen because of their appropriateness to real-time systems.

- Software views – Views recovered in the sub-steps mentioned before. Typical views are data flow diagrams or cross-reference lists.

- File structure - Functions and data structures collected in files are natural candidates for components since the reason for collecting functions in a file is often functional dependency.

- Diversified software – Safety-critical applications often use software diversity to ensure fail-safeness. All safety-critical parts of the system are implemented in n-versions. A 2-version system is for example built by using the A,B,C style. A and B software is developed by different groups based on the same requirements; B, however, uses inverted data for safety reasons. C synchronizes, distributes and votes results from A and B. The A,B,C parts of the system are natural candidates for components.

- Coupling - Coupling is the measure of interconnection among functions in a software system. A high degree of coupling between functions could be an indication that the functions belong to the same component. A low degree of coupling could accordingly indicate that the functions belong to different components.

- Design documentation - Design documents contain block-line diagrams and also textual descriptions that give hints about the existing architecture. Indicators are also include the table of contents since components are often described in subchapters.

- Scheduling information - Software execution at run-time is in some real-time systems controlled by an in-house-developed scheduler. The scheduler tasks implement functionality that must be completely performed within a specific time-frame to ensure correct execution. The tasks contain strong functionl-related source code that are component candidates.

The following tables give an overview of the applicability of recovery approaches to the different decomposition techniques ((+) very helpful for architecture recovery, (±) helpful for architecture recovery, (-) not helpful for architecture recovery).

decomposition technique	**recovery approaches**		
	software views	**file structure**	**diversified software**
Modular decomposition	+	-	+
data-oriented decomposition	±	-	-
event-oriented decomposition	±	-	-
outside-in design	±	+	±
object-oriented design	+	+	+

Table 2: Connection decomposition techniques - recovery approaches (part 1)

decomposition technique	recovery approaches		
	coupling	**design docu.**	**scheduling**
modular decomposition	+	+	+
data-oriented decomposition	-	+	-
event-oriented decomposition	-	+	+
outside-in design	-	+	±
object-oriented design	+	+	+

Table 3: Connection decomposition techniques - recovery approaches (part 2)

In most cases, composite components are even more difficult to identify than primitive components. Primitive components can often be directly related to source code or source code views. Composite components are structuring mechanisms that collect a number of primitive or composite components together. For composite component identification it is essential to know the decomposition strategy and to use domain knowledge. Also, in the case of composite components, the re-architect can group source elements in a large variety of ways.

Systems may have different decompositions. Depending on the recovery criteria used and the system experts working on the system, different decompositions may be identified. Legacy system architects should therefore verify the component candidates for completeness and correctness.

The activities in this phase are related to *view integration* in database schema design. Frank [Fra96] described view integration as a tool for complexity reduction. The database schema is not developed in a single step, but by integrating different modeling views, which are created by a number of view-designers. Frank included behavioral aspects of objects in his view integration method and defined preconditions, that must be fulfilled to allow a complete view integration process.

Interface detection

An interface is a point of interaction between a component and its environment. Other components can change the state of the component by using the services of the component.

Input to the interface detection phase are the identified and recovered components from the preceding phase. The detection of interfaces consists typically of the following steps:

- identify primitive components (interfaces of composite components are interfaces from primitive components)
- identify source code implementations as functions or data structures that organize communication to other components
- identify the interface roles (e.g. server)
- continue until all interfaces are identified

Connector Identification

The re-architect has to identify the connecting interfaces and the kind of information that flows through the connector. If the connector is a simple function call or a shared variable access, the properties of connectors are relatively simple to identify. If a complex protocol is built around the connector, the re-architect has to use design documentation, source code and domain knowledge for connector property identification. Protocol specification defines the properties of connectors. The specifications include rules about the order of information exchange or the types of interfaces the connector is able to mediate for.

Connectors have protocol specifications that define its properties. Examples for connector properties are:
(1) synchronisation information (synchronous, asynchronous)
(2) control flow (direct, indirect, none)
(3) degree of coupling

(1) Synchronisation information describes the type of synchronisation between components. *Synchronous communication* means that the sending operation and the consuming operation occur at the same time. Synchronous communication is necessary to ensure sequence-correctness in communication and to control access to shared resources such as memory. In *asynchronous communication,* the sender may transmit output (e.g. signals) without waiting for the receiver to consume them. Asynchronous communication is a buffered communication scheme implemented, for example, as signal buffer or ringbuffer.

(2) The property ‘control flow‘ describes the flow of control between components. ‘Direct’ means that the process control moves directly from one component to the other via the connector. ‘Indirect’ control refers to control flow not directly between components but via an ‘external’ component such as the operating system or a database. The control flow is ‘none’ if the control does not flow via the connector to another component. An example for ‘none’ control flow is shared memory.

(3) Coupling is the measure of interconnection among components in a software architecture. Low coupling is often an explicit goal in the software engineering process. For simplification, three coupling degrees may be used: low (data coupling - data passed as argument list), moderate (control coupling - control passed between components) and high (common coupling - components refer to global data area).

4.2.2.4 Output

The output of step 2 is a collection of structural architecture elements (component, interface, connector). The elements are not described by an ADL. The architecture information is still incomplete since additional architectural information (architectural properties) is still missing.

4.2.3 Step 3: Extract architectural properties

Input	- Textual representations - Graphical representations - Source code - Human knowledge
Method	- Architecture property identification (*see 4.2.3.2 Architecture Property Identification Methods*)
Output	- Collection of architectural properties

4.2.3.1 Introduction

An *architectural property* (AP) represents a specific design decision related to a functional or non-functional requirement and may be described using different architectural descriptions [EOG+98]. Architectural properties are described in detail in section *2.3.4 Architectural Properties* .

In this step, architectural properties of architectural elements (components, interfaces, connectors) are recovered. In step 6, architectural properties of the reference architecture are recovered.

The following table shows the relation between recovery input sources and architectural properties (see section *2.3.4*):

(+) very helpful for architecture recovery

(±) helpful for architecture recovery

(-) not helpful for architecture recovery

architectural property	**recovery input**			
	requirement specification	**source code**	**design documentation**	**human knowledge**
Constraints	+	±	+	+
Rationale	+	-	+	+
safety properties	+	-	+	+
standardization properties	-	-	+	+
architectural style	±	+	+	+
architectural patterns	±	±	±	+

Table 4: Relation between recovery input sources and architecture properties

Before starting with the recovery of architectural properties, the re-architect has to decide which properties to recover. Since the recovery of properties is time-intensive and costly, only a small amount of properties will be recovered. The re-architect has to focus on properties that matter the most, depending on the objective of the recovery process. In the case of architectural understanding, mainly information about *rationale* will be of interest. If the reference architecture will be recovered and used for forward engineering purposes, *constraints* and *architectural pattern* will be the main issues.

4.2.3.2 Architecture Property Identification Methods

Architectural properties may be identified in any input source of the recovery process. This makes the identification and recovery challenging because a number of techniques are necessary for property recovery [EOG+98][EKO+98].

4.2.3.2.1 Step 3.1: Analyze documents (e.g. requirements specification, design documentation)

Documents should be analyzed for design information limiting the degree of freedom in the architectural design process. Properties are often expressed as statements in documents.

Example of statements in documents	Property type
the system must react to a damaged telegram in less than 12 ms	Timing constraint
the system is a fail-safe system	Safety property
a data store is not allowed to change values stored in it	Semantic constraints
the components are only allowed to communicate with components in the same layer or in adjacent layers	Topological constraints

Standardization documentation describing techniques and measures necessary to achieve certain standards (e.g. safety integrity level) are a specific input source for architectural properties. Standardization is almost always restricting the architectural design and is often related to system properties.

The recovered properties are assigned to one (or more) architectural elements or to the entire system. Some ADLs support the description of architectural properties. Otherwise the properties must be linked to architectural elements as comments.

4.2.3.2.2 Step 3.2: Recover information from system experts

System experts should be interviewed about

- requirements limiting the degree of freedom in the design process (constraints)
- reasons for architectural design decisions (rationale)
- rules that have to be followed (standardization) and
- design and architectural principles (pattern)

It is helpful in this process to present documents already collected which contain architectural properties of the system experts. The documents are a basis for discussion and support the definition of the scope.

Again, the recovered properties are described and assigned to one (or more) architectural elements or to the entire system.

4.2.3.2.3 Step 3.3: Analyze source code

Source code may be analyzed manually or by using tools. Since architectural information is principally reflected in the data and processing model, these source code-related views are an important source for identifying architectural properties. The recovery of these views is described in *4.2.2 Step 2: Extract structural architecture*. The architectural style, for example, may be identified by analyzing data models.

Comments containing rationale may also contain rationale which describe the justification for the particular choices of structure or other design decisions.

Timing properties may be identified by performing a Rate Monotonic Analysis (RMA). RMA is a set of methods that enables developers to understand, analyze, and predict the timing behavior of real-time systems [AGP98].

Architectural properties may also be found be using pattern matching. Source code artifacts may match the descriptions of patterns – e.g. safety-critical component (see *7.2.2.5 Safety-critical component / safety block*). Pattern matching may be supported by a tool. An example is *Dialect* [Rea90] which is a component of *Software Refinery* (a workbench for reverse engineering C programs) . *Dialect* is a parser-generator which supports the creation of language-specific pattern recognizers.

4.2.3.3 Output

The output of this process step, is a collection of architectural properties mostly described as plain text. The architectural properties are related to an architectural element or the entire system.

4.2.4 Step 4: Describe the architecture

Input	- Structural architecture elements (output from step 2) - Collection of architectural properties (output from step 3)
Method	- Architecture Description Language (e.g. ASDL)
Output	- Description of the architecture of a single system

4.2.4.1 Introduction

In step 4, the recovered information from steps 2 and 3 is described by using a formal description technique. Software architectures can for example be represented by ADLs (Architecture Description Languages), diagrammatic syntax [DC95] or specification languages such as Z [AG92]. The type of representation must be chosen depending upon specific requirements (e.g. the system domain) and is in principle independent of SARM.

Step 4 describes architectural information of a single system in a consistent and complete way. System shareholders can use the descriptions for communication purposes and architectural assessment.

4.2.4.2 Input

Input from step 2 is the collection of structural architecture elements (component, interface, connector) of the system under consideration.

Input from step 3 is the collection of architectural properties mostly described as plain text. The architectural properties are related to an architectural element or the entire system.

4.2.4.3 Method

4.2.4.3.1 Step 4.1: Choose a description technique

The architecture of a software system can be described using various formal models, architecture description languages or graphical notations. The re-architect has to decide which language should be used for representing the architectural information.

The type of representation should provide/support

- A formal syntax to support the identified requirements for architectural descriptions like support for composition, abstraction, configuration and heterogeneity analysis (see section *2.2* and *3.1.1*).
- Support for the description of specific requirements (e.g. the system domain or system type, e.g. real-time systems).
- Complete representation of the architectural information recovered by the first two steps of SARM; otherwise, information will be lost.
- The identification of the reference architecture. A structured architecture recovery process must be based on recovery data that is available in a structured manner.

ASDL is used as the description technique in this dissertation because it is a flexible architecture description language which supports the commonality process.

4.2.4.3.2 Step 4.2: Transform input information and describe the system

Architectural information recovered in steps 2 and 3 must be transformed into the chosen architectural representation.

Architectural elements can in most cases be directly transformed into the architectural representation. The ADLs typically provide a notation for description of components, connectors and interfaces.

The description of architectural properties is in some cases supported by a notation (e.g. ‘Constraints’ in WRIGHT or ‘behavior’ in Rapide). ASDL allows the re-architect to define specific notations for properties. In many cases the architectural presentation does not support a notation for architectural properties. In this case, comments may utilized to describe the architectural properties.

4.2.4.3.3 Step 4.3: Verify results by system experts

The process of verification of the architectural information is an iterative process with a number of processing loops and feedback rounds which involves the system experts. At the beginning of the verification process the notation of the architectural description technique should be presented. In personal meetings with experts of the system the architectural descriptions should be presented by the re-architect and discussed.

4.2.4.4 Output

The output is a complete and consistent description of the architecture of the system under consideration. The description may be used for architectural reasoning and architectural assessment.

4.2.5 Step 5: Extract reference architecture

Input	-	Description of a number of architectures of a product line
Method	-	Commonality analysis
Output	-	Set of reference architectural elements

4.2.5.1 Introduction

Reference architectures are designed to meet functional requirements shared by applications in the domain. A reference architecture should provide abstractions that are common to all family members and a design structure to accommodate change [Wei98][DG98]. The reference architecture will be input for further system development because it enables the derivation of new family members from the reference architecture.

The reference architecture is derived by discovering the common aspects of the family members (commonality analysis). The main input for the commonality analysis are the architecture structure descriptions of the family members.

Steps 5 and step 6 are no longer pure re-engineering processes, but rather a combination of forward and reverse engineering activities. The reference architecture not only represents historical information - it also contains information about possible future changes and developments for the system family.

4.2.5.2 Input

The input to step 6 is the ASDL-description of the family members. If the number of family members is relatively low (less than 5), the commonality analysis is done manually.

4.2.5.3 Method

The commonality analysis [EG98],[Eix98],[JRL00] identifies components from different family members that appear to be similar. The input into the process is gathered from all components of the family members to be analyzed.

The commonality analysis consists of 3 steps:

4.2.5.3.1 Step 5.1: Perform component selection

Perform component selection on each family member by defining selection criteria. Selection criteria are derived from ASDL language constructs and are typically: (1) component role, (2) component group, (3) component safety and (4) component type. The numbering of the commonality factors is identical to the selection sequence. After each selection step the number of valid components that fulfill the selection criteria is reduced. The order of the selection criteria has no influence on the resulting component sets, but it is very useful to choose the criteria with many components first. In this way the number of components is reduced on every level.

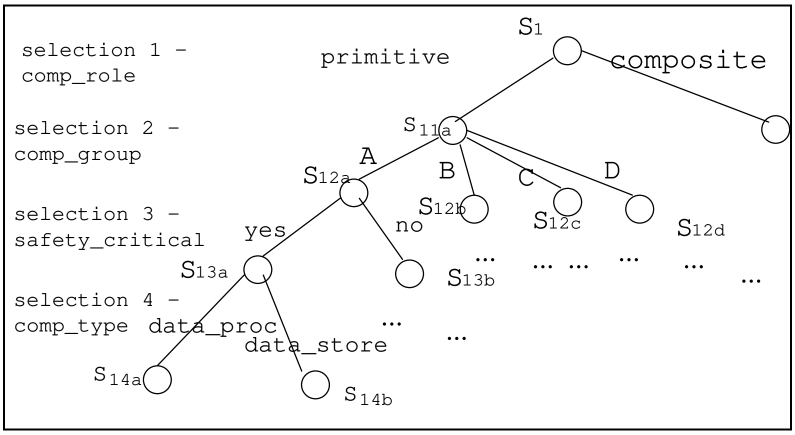


Figure 14 selection tree

Figure 14 shows the selection tree. The root of the tree is the set of components (S) that contains all components of the case study under consideration. After the first selection, two subsets of S are created (S_{1a} and S_{1b}). The name of the created component set indicates the original selection group (e.g. S1), the selection step (e.g. 1) and selection attribute value (e.g. A). After the last selection step, selection 4, the number of components in the selection group will be rather small (between 0 and 10).

For example, in the set S_{4a} all components will be gathered that are primitive components, that belong to group_A, that are safety-critical and that are data processing components. One component can therefore be only a member of one component set in one layer. For the commonality analysis only the leafs of the selection tree are of interest. The order of the selection criteria is therefore of no interest for the commonality analysis.

The selection criteria could be any ASDL language construct with enumeration type (as long as the construct is used in the description, e.g. for a non-safety-critical system, the construct *safety-critical* will not be used) . The depth of the selection tree is at least one step, but can be n steps deep.

4.2.5.3.2 Step 5.2: Produce matching table

A human expert analyzes the leaf component sets of the selection tree (e.g. S_{4a}) versus the same component set(s) of other family members (e.g. F_{4a} from case study Finland, K_{4a} from case study Korea). Each component is compared with each component of the other sets. Important indicators for similarity of components are the component description and the interface descriptions.

When comparing different family members, three outcomes are possible:

(a) Full-match: A component is identified to provide and require the same services in all product members. The similarities of the components are so strong that the components can directly be replaced by one reference component, which becomes part of the reference architecture (direct transformation of full-match components to a reference component).

(b) No-match: Components that have no similar properties to any other component of the other product families are called no-match components. No-match components provide services that are specific for one product family member. These components are candidates for transformation to reference components. The expert may, however, exclude no-match components because they do not conform to the reference architecture requirements, which are requirements developed in close cooperation with system engineers. The engineers try to anticipate future changes and extensions and document this information in the reference architecture requirements. Transformed no-match components are annotated with a constraint indicating under which conditions the component should be derived for new family members.

(c) Partial-match: Partial-match components are components from different family members with similar properties that are not full-match components. An example is a component in family member A that provides similar services as two components in family member B. The direct transformation to a reference architecture component is therefore not possible. The expert decides, in most cases together with the system architect, what should happen to these components. Partial-match components can be combined to form one component or they can be divided into a number of components. The resulting components have to conform to the reference architecture requirements.

Step 2 is documented in a matching-table:

Full match	components of the all family members that provide full match
partial match	components of family members that match with components of some family members (but not all of them)
no match	all other components that cannot be matched with any component

4.2.5.3.3 Step 5.3: Produce reference architecture

A human expert forms the reference architecture from the reference components. The architectures of the family members represent the implementation architectures of a system (as-is-architecture). The reference architecture represents however the conceptual architecture of a system (as-should-be-architecture). The transformation process has to take this fact into consideration.

A human expert uses the matching-table from commonality analysis step 2 and observes the following rules:
(a) Full-match components are replaced by one reference component. This means that there is a direct transformation from full-match components into a reference component.

(b) Partial-match components are transformed following as-should-be-architecture requirements. This means that the human experts (in this case reverse architect and forward architect) transform partial-match components so that the result conforms to the as-should-be-architecture requirements. The human experts can join partial-match components into one component or divide them into a number of components.

(c) No-match components provide specific services and are candidates for direct transformation into a reference component. The human experts can however exclude no-match components because they do not conform to the as-should-be-architecture requirements. Transformed no-match components are connected with a constraint indicating under which conditions the component should become a member of a new family member.

As-is versus as-should-be architecture

The recovered architectural descriptions represent the implementation architectures of the case study systems (as-is-architecture). The reference architecture represents a conceptual architecture of a system (as-should-be-architecture) that will not be instantiated but will be used for deriving the initial architecture of new family members.

The handling of the partial-match components indicates that the commonality analysis process is not a purely re-architecting process but is also influenced by forward engineering and architecting. The resulting reference architecture does not have a direct relation to the as-is-architectures of the existing family members because of the transformations of partial-match components. Therefore, before working on the reference architecture, the as-is architecture must be carefully validated.

For the development of the as-should-be architecture, the software architect has to anticipate changes. The process “requires special effort to anticipate how and where changes are likely to occur.“ [GJM91] p.52. Evolvability and reusability are software quality aspects that are strongly affected by the anticipation of change. Carefully designed and verified as-should-be architectures are an important contribution to *design for change* .

4.2.5.4 Output

The output of step 5 is a collection of reference architectural elements that are common to all family members. The reference architectural elements are not described by an ADL. The architecture information is still incomplete since additional reference architectural information (architectural properties) is still missing.

4.2.6 Step 6: Extract reference architecture properties

Input	- Architectural properties from step 3 - Collection of reference architectural elements from step 5 - Textual representations - Graphical representations - Source code - Human knowledge
Method	- Reference architecture property identification
Output	- Collection of reference architecture properties

4.2.6.1 Introduction

Reference architecture properties are common properties that represent properties of the overall system. Examples of reference architecture properties are throughput, latency, gross performance, component consistency, security and reliability [GMW97], [AAG95].

In step 3, the architectural properties of architectural elements (components, interfaces, connectors) are recovered. In step 6, architectural properties of the reference architecture are recovered.

4.2.6.2 Input

The main input to the process are architectural properties from step 3, the collection of reference architectural elements from step 5 and, if necessary, system documentation as textual representations, graphical representations, source code and human knowledge.

4.2.6.3 Reference Architecture Property Identification Methods

4.2.6.3.1 Step 6.1: Analyze reference architecture elements

The collection of reference architectural elements from step 5 is an important input. The comparison method for architectural elements for commonality (see 4.2.5.3) is also valid for architecture properties.

(a) Full-match components: The similarities of the components are so strong that the components can directly be replaced by one reference component that becomes part of the reference architecture (direct transformation of full-match components to a reference component). In the same way, the properties of the components can directly be included in the reference architecture.

(b) No-match components: If the re-architect includes the no-match components in the reference architecture requirements, the properties are candidates for reference architecture properties. The re-architect tries to anticipate future changes and extensions and decides about the significance of the property based on this information.

(c) Partial-match component: Depending on the decision if the partial-match components are combined to form one component or to be divided into a number of components, the re-architect has to decide how to manage the reference architecture property related to the component.

4.2.6.3.2 Step 6.2: Analyze other input sources

As in step 4, other input sources may be analyzed to identify reference architecture properties. The main question the re-architect has to ask: 'What properties are properties that are valid for more then one architectural element or for the entire system?'. The recovery techniques are the same as for architectural properties:

- Analyze documents (see 4.2.3.2.1)
- Recover information from system experts (see 4.2.3.2.2)
- Analyze source code (see 4.2.3.2.3)

4.2.6.4 Output

The output is a collection of architectural properties mostly described as plain text. The architectural properties are related to an architectural element or the entire system.

Like step 5, step 6 is also no longer a pure re-engineering process but rather a combination of forward and reverse engineering activities. The architect must decide which reference architecture properties are of interest for the reference architecture.

4.2.7 Step 7: Describe the reference architecture

Input	- Set of reference architectural elements (output from step 5) - Collection of reference architecture properties (output from step 6)
Method	- Architecture Description Language (e.g. ASDL)
Output	- Description of the reference architecture

4.2.7.1 Introduction

In step 4, the recovered information from steps 2 and 3 is described by using a formal description technique. The same formal description technique is used to describe the reference architecture.

4.2.7.2 Input

Input from step 5 is the set of reference architectural elements (component, interface, connector).

Input from step 6 is the collection of reference architecture properties mostly described as plain text.

4.2.7.3 Method

4.2.7.3.1 Step 7.1: Transform input information and describe the architecture

Architectural information recovered in steps 5 and 6 must be transformed into the chosen architectural representation.

As described in step 4, the architectural elements can in most cases be directly transformed into the architectural representation. The description of reference architectural properties is in some cases supported by a notation. ASDL allows the re-architect also in this case to define specific notations for properties.

4.2.7.3.2 Step 7.2: Verify results by system experts

The process of verification of the architectural information is in principle the same as described in step 4. Since the recovery of the reference architecture is not only a re-engineering process but also a forward engineering process, the involved reviewers should be system experts who are involved in both processes.

4.2.7.4 Output

The output is a complete and consistent description of the reference architecture of the system family under consideration. The description may be used for architectural reasoning, architectural assessment and to derive architectures for new system family members.

5. ASDL (Architecture Structure Description Language)

As identified in section 2.2, most of the ADLs are general purpose languages that can be used for the description of software architectures of different domains. ADLs for specific domains such as real-time systems do not exist at the moment. Properties of such systems such as timing requirements, safety aspects, product line relations and HW-SW relations cannot be represented in a satisfying manner by existing ADLs.

For these reasons, an ADL called ASDL (Architecture Structure Description Language) [JRL00],[EOG+98], [EKO+98], [Eix98] was developed. ASDL is specifically designed for the description of software architectures of product lines of real-time systems. The aim of the language is to represent the conceptual architecture of real-time systems and to provide input for the reference architecture recovery process. A conceptual architecture describes a system in terms of its major design elements and the relationships between these elements.

ASDL describes the main parts of a conceptual architecture: component, interface, connector and configuration. A component consists of a number of interfaces. Interfaces are connecting points that provide services of the component to its environment. Each component has at least one interface. Interfaces are part of the component description in the ASDL. Connectors relate two interfaces of two components. Connectors are the transport media between components. Configuration describes the instances of components, interfaces and connectors and how they are assembled to form a specific architecture.

Detailed information about the syntax and semantic of ASDL can be found in the appendix (section 7.2 ASDL).

5.1 ASDL Basic Concepts

In this section the basic concepts of ASDL will be described.

5.1.1 Characteristics

ASDL describes an architecture as a set of components that provide at least one interface to its environment and that are interrelated with connectors. ASDL describes not only the basic attributes of the architectural elements but also additional attributes called properties. Typical architectural properties are detailed typing information (e.g.

types of data communicated between components), scheduling constraints, interaction protocols, and resource consumption information [GMW97].

ASDL tries to find the balance between too much information that restricts commonality analysis and too little architecture-related information. ASDL represents important information, e.g. information about exchange of information between component interfaces in a compact form. Details, such as interface behavior, is not represented in the language. Such information overloads the system architect with unnecessary details. ASDL supports the architect in making typical architectural decisions such as the architectural partitioning or global error handling policy. ASDL is also meant as a communication medium for distributing architectural decisions to system stakeholders. Domain specific aspects such as safety or fault tolerance are described in a simple yet powerful way.

ASDL is a flexible language that can be adjusted to different domains. The language is divided into the *structure base* and the *property base*. The *structure base* describes the structural elements of an architecture and is usually not changed once it is defined. The *property base* describes additional architectural concepts and can be changed accordingly to the needs of the domain.

ASDL models components, component interfaces and connectors relatively independely. The result is an increase in scalability and extensibility. Components may be developed independently, in different languages and by different groups. Components may be adapted to heterogeneous, distributed, multi-user environments. Components may be redesigned to fit into architectures with different requirements on component interfaces.

5.1.2 Structure Base

The architectural elements are defined in the structure base. The structure base contains the basic attributes of the main architectural elements – components, interfaces and connectors.

Components are the building blocks of an architecture and encapsule the system functionality. Components are encapsulated and are black-boxes to other elements of the architecture. Users of components have no information on how the functionality is implemented in the component. The component base concept consists of the *component type*, the *component name* and the *component description* in plain text.

Interfaces are part of a component and provide services to the environment of the component. Other components can change the state of the component only by using the services of the component (like in object-orientation). The interface base concept consists of the *interface name* and the *interface description*.

Connectors are links between two components. Connectors define the relation between interfaces of components. The connector base concept consists of the *connector name*, the *connector description* and the *connecting interfaces* (represented by the corresponding identifiers).

5.1.3 Property Base

Additional information to architectural elements are defined in the property base. The property base supports the communication of architectural information to system stakeholders.

The property base is flexible and can be adapted to specific domains. Properties and property values can be added or removed. Property values can in most cases be selected from a predefined list of values. The number of property values will therefore be rather small and support the analysis of the architectures. A commonality analysis can be based on the property base.

In the following, the most important properties are described:

Component properties — Components are either composite or primitive. Composite components consist of at least one composite or primitive component. Primitive components are on the lowest level of the component hierarchy and do not contain further components, but are elementary entities. The property describing this information is called *structure property*.

In the case of safety-critical systems, n-version programming is used for failure detection. N-versions of software for the same functionality are developed by n teams. At run-time the n-versions are executed and the results are compared by a software voter. Components are either n-version software, voter or non-safety-critical components. This property is called *component group*.

Each component provides a set of services that can be summarized by the main characteristic or type of the component. Components can be characterized as data processing components, data store components or other types of components. The property describing this information is the *type property*.

Interface properties — Interfaces are part of components. Components play different roles depending on the relationship to other components. A component A can play the role ‘server’ to component B and at the same time play the role ‘producer’ to component C. The role of a component, therefore, cannot directly be related to a component, but rather to the component interface. The property *interface role* provides information about the interface role.

Connector properties —The property *communication type* provides information about how information is exchanged between two services. The property contains information about the data flow direction, the control flow (synchronous, asynchronous) and if the flow is continuous or not.

The transfer pattern can be either asynchronous or synchronous, which is defined in the property *transfer pattern*. Information about the type of information exchange (e.g. periodic) is defined in the *information exchange* property.

5.1.4 Components

A component represents an independent computation. On the architectural level of abstraction, components are described by the computation and the interfaces. Detailed design or implementation information is of no interest at the architectural level because it moves the focus away from high-level design and communication.

In ASDL, composite or primitive component types are constructed. In the ASDL configuration, instances of component types are declared. The resulting system is a hierarchical structure of composite and primitive component instances. The root component in the system is called *system component*. A number of component properties allow detailed specification of the component (see *5.1.3 Property Base*).

Computation describes the functionality of the component. The computation performs the functionality described by the interfaces. Detailed information about the computation is undesirable at the architectural level. An ASDL component represents a black-box and provides no information on how the computation is actually performed.

In ASDL, the interfaces are designed as required or provided interfaces, respectively requiring information from the environment or providing information to the environment.

5.1.5 Interfaces

Interfaces are defined points of connection between a component and its environment. An interface of a component indicates a specific aspect of the component's behavior; it describes a partial specification of the component. The collection of interfaces describes the entire specification of the component.

Current interface specification techniques do not adequately describe all of the properties exhibited by components that will determine the integrability of the component [BW96]. ASDL cannot solve that problem but it tries to focus on the most important aspects of component interfaces of typical real-time components.

ASDL interfaces provide and/or require services. Services represent any type of information that is provided to the environment by the component, or information that is required by the component from the environment. The interfaces in ASDL give no indication of how the information required/provided is exchanged with the environment. The connector describes this property.

Interface roles

Components in a system play different roles that influence not only the design of the component but also the connection to other components. Components can even play different roles depending on their relation to other components. The identification of such facts will be of high interest for development of new product family members. A 're-architecturing' of components may be necessary.

The roles of components greatly depend on the information exchange with other components and the frequency of information exchange. The component roles determine therefore also other architectural properties, e.g. the communication type.

The following roles can typically be found in legacy systems:

- Client/Server - Servers provide information that the client accesses using the communication type of the server. The client plays an active role and requests information from the server.

- Producer/Consumer - Consumers receive information from the producer. The information transfer is asynchronous and can happen at any time. The producer calls the API of the consumer and transfers the data. The control remains with the producer and will shift to the consumer later on, depending on the scheduler. The API is part of the consumer and not of the producer. This could be different in other systems but has been solved in this way because the time between inputs is usually very long and polling would therefore be a waste of CPU capacity.

- Library - Components that provide different functionalities (monitor, error handling, CPU-test, memory-test, clock) and are highly reusable are library components. The communication from other components to the library depends on the service that is used and can be based on API calls. The library components play therefore different roles in the system.

5.1.6 Connectors

On the architectural level, the semantic content of communication between components is much more important than design and implementation details. The concept of message passing in object-oriented analysis and design methodologies is an example of hiding detailed connector information. For the system architect, it is not necessary to know details about connectors, e.g. if the information is passed by a synchronous function call, by sending a bus message to another processor or via rendezvousing with semaphores.

Each connector has a protocol specification that defines its properties. Properties include rules about the types of interfaces it is able to mediate for, assurances about properties of the interaction, rules about the order in which things happen, and commitments about the interaction (e.g., ordering performance etc.) [SG96].

ASDL describes what happens between component interfaces without any further information about the communication technique. The communication may be based on messages, events, signals, function calls or any other communication concept. This implementation independence allows the architect to focus on communication pattern without thinking about design and implementation.

The ASDL connector properties include the following:

(1) Synchronization information (synchronous, asynchronous)
(2) Communication information / arrival pattern (periodic, irregular, bounded, bursty, unbounded)
(3) Control flow (direct, indirect, none)
(4) Time requirement

(1) Synchronization information describes the type of synchronization between components. Synchronous communication means that sending operations and consuming operations occur at the same time. Synchronous communication is necessary to ensure sequence-correctness in communication and to control access to shared resources such as memory. In asynchronous communication, the sender may transmit output (e.g. signals) without waiting for the receiver to consume them. Asynchronous communication is a buffered communication scheme implemented, for example as signal buffer or ringbuffer.

Of special interest is the synchronization connected to diversified software. The input and the output to diversified software is usually synchronized to ensure correct computations.

(2) The property communication information describes the arrival patterns of communication flowing over connectors.

The following patterns are identified (see also [AGP98]):

- Periodic - the arrival of communication is separated by equal timing intervals
- Irregular - the arrival of communication separated by unequal, but known, intervals
- Bounded - communication cannot occur before the elapse of a minimum separation time after the end of a previous event
- Bursty - communication can occur arbitrarily close to the previous one, the number of communication units over a specified interval is bounded
- Unbounded - communication can occur at any time, with no restrictions or bounds

Communication properties are an important input for performance analysis and HW-SW co-design on the architectural level.

(3) The property ‘control flow‘ describes the flow of control between components. ‘Direct’ means that the process control moves directly from one component to the

other via the connector. 'Indirect' control refers to control flow not directly between components but via an 'external' component such as the operating system or a database. The control flow is 'none' if the control does not flow via the connector to another component. An example for 'none' control flow is shared memory.

Direct control flow is typically found in between producers and consumers. Connectors based on indirect control flow exist between components that belong to different execution tasks controlled by the operating system. 'None' control flow exists per definition around connectors based on shared memory.

(4) Time requirement describes the maximum time interval between the arrival of a communication event and the termination of the according action of the component (e.g. providing a service to its environment).

5.1.7 Configuration

A configuration is a collection of constraints that wire objects into a specific architecture [MQ94]. The ASDL configuration describes the instances of components, the connection between super-component and sub-component and connectors and how they are assembled to form the specific architecture.

5.1.8 Example

The following example will be used for describing the ASDL in a concrete example of the case study (see also section *6* for more information about the case study).

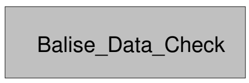


Figure 15: Component type *Balise_Data_Check*

Figure 15 shows the representation of a component type in ASDL. The component type acts as a template for instantiation of component instances. ASDL permits the declaration of more than one instance of a component (multiple instanciation).

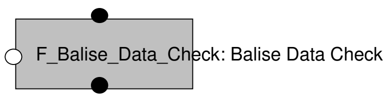


Figure 16: Component *F_Balise_Data_Check* with interfaces

Figure 16 represents an instance of the component type and also represents three component interfaces. The white circle represents a required service and the black circles provided services.

①
comp_begin Balise_Data_Check
comp_desc The Balise_Data_Check component checks incoming balise for correctness and completeness.
comp_group comp_D
comp_struc composite_comp
comp_type data_processing
safety_critical n.a.

②
intf_name Balise_Data
intf_desc The interface requests Tran balise data from an antenna.
intf_role consumer

intf_name ATC_Data
intf_desc The interface provides ATC_Data.
intf_role producer

intf_name Loco_Data
intf_desc The interface provides Loco_Data.
intf_role producer

comp_end Balise_Data_Check

③
config_begin TRAN_model
comp_inst F_Balise_Data_Check: Balise_Data_Check
…
config_end TRAN_model

Figure 17: Example of textual ASDL

Figure 17 shows the declaration of the component type *Balise_Data_Check* and the configuration of component *F_Balise_Data_Check*.

① Definition of the component type *Balise_Data_Check*.

comp_group represents information about the group the component type belongs to. In the case of 2-version programming, four different groups exist.

- comp_A – Safety-critical component, developed by development group A.
- comp_B – Safety-critical component, developed by development group B.
- comp_sync – Safety-critical component that is the synchronizer of n-version components
- comp_voter – Safety-critical component that is the voter of n-version components
- n_a - Primitive component that is neither safety-critical nor composite component, which has no safety-critical sub-component.

comp_struc represents structural information. A component may either be composite or primitive. Primitive components are not further decomposed; composite components consist of at least another composite or primitive component.

comp_type represents the main functional characteristic of the component.

- **data_processing** - transformation of input data and providing transformed data to environment
- **data_store** - storing data in specific form as a file, data structure (e.g. ring buffer) or database
- **communication_control** - controlling flow of information between entities (e.g. synchronization of data exchange between two interrelated components executed on different processors)
- **syncroniser** - synchronizing and providing data to safety-critical (n-version) components (e.g. providing inverted data for B component)
- **voter** - analyzing and verifies outcome of n-version components
- **monitor** - in-house developed scheduler
- **run_time_system** – organizing the execution of a system at run-time
- **user_interface** - organizing information exchange with human user
- **n_a -** functional characteristic cannot be identified (often the case for composite components)

safety_critical represents information about the safety-criticality of the component. A component is either safety-critical, not safety-critical, or in cases where the component is a composite component and the sub-component of the component are safety-critical and not safety-critical, the safety-critical component is not applicable (n.a.).

② The definition of the interfaces is embedded in the component definition block.

intf_role provides information about the role the interface represents to the environment. Components can play different roles depending on the relation to other components. The property role can therefore not be assigned to the component but must be assigned to the interface.

- **client -** requests information from a *server* component (represented through a *server* interface)
- **server** - provides information to the environment
- **producer** - produces information - is different to the server, since the information is provided to the environment without request from the environment
- **consumer** - consumes information - is different from the client, since information is consumed without a request to the environment

③ Configuration describes the instances of components, interfaces and connectors and how they are assembled to form a specific architecture.

comp_inst - represents the instantiation of the component type *F_Balise_Data_Check*

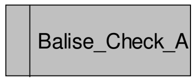


Figure 18: safety component *Balise_Check_A*

Figure 18 represents a safety-critical component (safety component). The safety component is represented by a rectangle including a vertical line on the left side of the rectangle. Specific rules apply for safety-critical components. For more information see section *7.2.2.5 Safety-critical component / safety block* .

A safety-critical component is represented in ASDL by the safety property (**safety_critical** -yes).

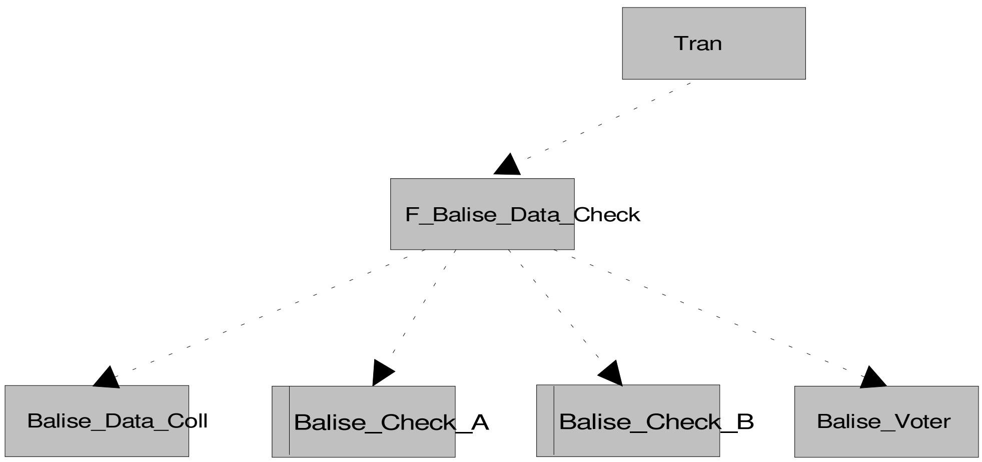


Figure 19: Relation between Component *F_Balise_Data_Check* and super/sub-components

Figure 19 shows the super-component/sub-component relationship between the component *F_Balise_Data_Check* and its sub-components and super-component. The hierarchical links between components are represented by dotted lines with arrows. The arrowhead points to the sub-component.

The super-component/sub-component relation is represented in textual ASDL by the *child_of* property (e.g. *F_Balise_Data_Check* is **child_of** Tran). Textual ASDL represents the relation from sub-component to super_component (child of). The relation of super_component to its sub_components (parent of) is not directly represented in textual ASDL (redundant information).

config_begin TRAN_model
…
comp_inst Tran: F_Tran
comp_inst Fin_Balise_Check: F_Balise_Data_Check
comp_inst Balise_Data_Coll: F_Balise_Data_Coll
comp_inst Balise_Check_A: F_ Balise_Check
comp_inst Balise_Check_B: F_ Balise_Check
comp_inst Balise_Voter: F_Balise_Voter

…
child_of_inst Fin_Balise_Check Tran
child_of_inst Balise_Check Tran
child_of_inst Balise_Check Tran
child_of_inst Fin_Balise_Check Tran
child_of_inst Fin_Balise_Check Tran
child_of_inst Fin_Balise_Check Tran

…
config_end TRAN_model

Figure 20: Configuration of Figure 19

Figure 20 shows the configuration of the architectural model described in Figure 19. **child_of_inst** relates the instance of a sub-component (e.g. Fin_Balise_Check) to an instance of a super_component (e.g. Tran). An instance of a sub-component is child_of one and only one instance of a super-component. An instance of a super-component may be 'parent_of' many sub_components.

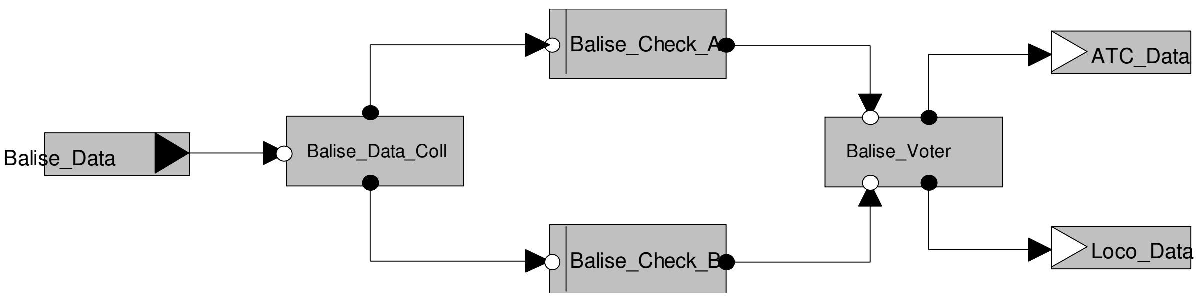


Figure 21: Sub-components from component *Balise_Data_Check*

Figure 21 represents the sub-components of component *Balise_Data_Check* and the relation to the interface of component *Balise_Data_Check*. For definition and detailed information see Figure 17 and Figure 19. *Balise_Data* is a component interface requesting information from the environment, represented by the black arrowhead.

ATC_Data and Loco_Data are interfaces providing information to the environment. Providing interfaces are represented by white arrowheads. Figure 21 also provides examples of connectors relating component interfaces. The arrowhead of the connector line indicates the flow of information. This example contains only unidirectional information flows. Figure 21 represents a typical safety block. For detailed information about safety blocks see *7.2.2.5 Safety-critical component / safety block*.

Figure 22 represents the textual ASDL of the connector from component *Balise_Data_Coll* to component *Balise_Check_B* and the definition of the connected interfaces.

① **comp_begin** Balise_Data_Coll
…
intf_name Coll_Data_B
intf_desc The interface provides ...
intf_role producer
…
comp_end Balise_Data_Coll

comp_begin Balise_Check_B
…
intf_name P_Coll_Data_B
intf_desc The interface ...
intf_role consumer
…
comp_end Balise_Check_B

② **conn_name** Balise_Coll_Data
conn_desc The connector …
data_flow **unidirectional** (**intf** Coll_Data_B **to intf** P_Coll_Data_B)
control_flow n.a.
transfer syn
info_exch irregular
ev_type external

③ **config_begin** TRAN_model
…
comp_inst Balise_Data_Coll: F_Balise_Data_Coll
comp_inst Balise_Check_B: F_ Balise_Check
…
conn_inst Balise_Coll_Data
    intf Balise_Data_Coll: Coll_Data_B; Balise_Check_B: P_Coll_Data_B
…
config_end TRAN_model

Figure 22: Textual ASDL - connector description

① Partial description of Figure 21 with the definition of interface *Coll_Data_B* of component *Balise_Data_Coll* and definition of interface *P_Coll_Data_B* of *component Balise_Check_B*.

② Definition of connector Balise_Coll_Data which connects the interfaces defined in ①.

- **data_flow** provides information about the direction of the information flow (unidirectional, bi-directional). The **unidirectional** property provides in addition information about the source and the sink of the information flow represented by the according interface.
- **control_flow** provides information about the direction of the control flow (unidirectional, bi-directional). In this example the property is not applicable (n.a.).
- **transfer** represents information about the synchrony of the information flow (synchronous, asynchronous).
- **info_exch** provides information about the exchange pattern. The exchange may either happen periodically, irregular, bounded, bursty or unbounded. Detailed information about these attributes can be found in section 5.1.6 .
- **ev_type** represents information about the event transported by the connector. Events may either be external, internal or timed. Detailed information about these attributes can be found in section 5.1.6 .

③ shows the configuration of the two involved components (Balise_Data_Coll, Balise_Check_B) and the definition of the connector.
conn_inst defines the connector *Balise_Coll_Data* that connects the interfaces *Balise_Data_Coll: Coll_Data_B* and *Balise_Check_B: P_Coll_Data_B* .

5.2 Relation to ACME

ACME (Architecture Description Interchange Language) [GW99],[GMW97] provides an (intermediate) interchange format that allows the transformation of separately-developed ADLs. ACME's intention is to make it possible to integrate a broad variety of existing ADLs. This makes ACME a candidate for an interchange format for ASDL specifications.

The designer of ACME found that there is a lot of commonality in the capabilities of ADLs. "Acme attempts to embody those commonalties while also allowing the incorporation of ADL-specific information, so that auxiliary information can be retained. This scheme permits subsets of ADL tools to share whatever architectural information is jointly understood by those tools, while tolerating the presence of information that falls outside their common vocabulary." [GMW97] p.1. The commonalties of the ADLs were especially found in the role of structure in architectural description (hierarchical collection of interacting components).

One key question is why it is beneficial to use ACME-based translation between two ADLs instead of direct translation. According Garlan et.al. [GMW97], the ACME-based translation simplified the handling of the structural aspects of the architectural description. Furthermore, ACME augments ADLs with a set of ACME-based tools for graphical browsing, conversion to web documents, and persistent storage.

ACME provides seven classes of design elements for architectural representation: components, connectors, systems, ports, roles, representations, and rep-maps.

- **Components** represent high-level computational elements and data stores of a system (e.g. clients, servers).
- **Connectors** mediate the communication and build interactions among components (e.g. pipes, procedure call).
- **Systems** represent the configuration of components and connectors.
- **Ports** represent points of interaction between a component and its environment (e.g. single procedure signature).
- **Roles** represent interfaces of connectors. In ACME, roles of a connector define a participant of the interaction (e.g. caller and callee).
- **Representation** represents the hierarchical description of architectures (e.g. a component represented by one or more detailed, lower-level descriptions).
- **Rep-map** ("representation map") represents the correspondence between the internal system representation and the external interface of the component or connector that is being represented (e.g. association between internal ports and external ports).

ACME not only provides seven classes for representing structural information but also concepts for auxiliary information (properties), by annotating architectural structure with lists of properties. Properties are represented by name, an optional type, and a value. ACME provides templates for the specification of recurring patterns. The patterns are instantiated by parameterization. Templates may be grouped together into a collection of architectural styles.

ASDL	**ACME**
components	components
connectors	connectors
configurations	systems
component interfaces	ports
component interfaces	roles
hierarchical decomposition	representations
-	rep-maps

Figure 23: Transformation of ACME concepts to ASDL

Figure 23 shows a transformation of ASDL concepts to ACME concepts.

6. The case study: The ITRIN product family

In chapter *4 SARM (Software Architecture Recovery Methodology)* a methodology was introduced for architectural recovery. The methodology was evaluated on a case study, the BTM (Balise Transmission Module) case study of ABB. The development of SARM and also the evaluation on the case study occurred as part of the ARES (Architectural Reasoning for Embedded Systems) [JRL00] project. ARES was an international research project supported by the European Commission under ESPRIT framework IV contract no. 20477. The project was pursued by Nokia RC Finland, ABB Corporate Research Norway, Philips RC Holland, Imperial College, the Technical University of Madrid, and the Technical University of Vienna. A work package in the ARES project was dedicated to architecture recovery. ABB and the Technical University of Vienna were working on architecture recovery of the ABB case study.

This chapter describes the members of the system family, the recovery process applied on each family member and the identification process of the reference architecture. System-related information has been made anonymous for reasons of confidentiality.

6.1 Introduction to the case study

The chosen case study system is the BTM (Balise Transmission Module) subsystem of a train control system (TCS). The TCS is an embedded real-time software system for controlling train movement. The TCS represents a product family of systems delivered to a number of railway companies around the world. The members of the product family are named after countries where the systems are in use – Spain, Finland, and Korea. In the future, the product family will constantly be evolved because of customer requirements, railway standards, hardware evolution and technological changes (e.g. GPS information for exact positioning of the train).

6.1.1 Basic functionality

Controlling train movements with software-based control systems emerged in the 90s. The most important reason for this concerns more demanding requirements for safety aspects of train systems. The BTM (Balise Transmission Module) handles the reception and decoding of intermittent balise information. The BTM subsystem can be used in systems for controlling high speed train movement and for precision stops in metros.

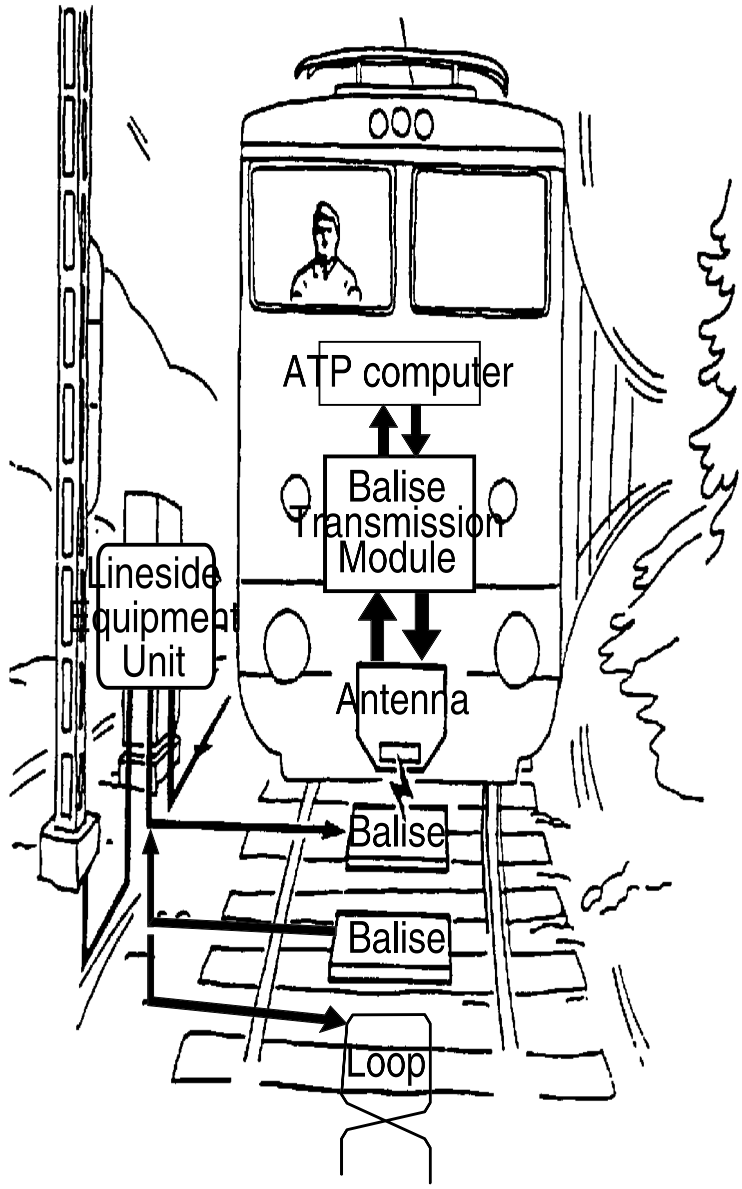


Figure 24 BTM System overview

Figure 24 gives an overview of the BTM system and its environment. The data flow is from the lineside equipment through the train antenna, the transmission equipment and BTM to the Automatic Train Protection (ATP) inside the train.

Line-side equipment provides the train driver with control information such as speed limitations or light-aspects, e.g. indicating ‘stop’ or ‘go’. Line-side Equipment Units digitalize the control information into balise telegrams (for simplicity they are called ‘telegram(s)’ in the rest of this work). Telegrams are stored in a ‘balise’, a device used to transmit information from the track to the TCS system of a passing train. On the train an antenna is mounted that continuously transmits a radio signal at a given frequency.

When the train passes a balise, the balise receives energy (through the radio signals) as well as a reference clock. When the balise has received enough energy, it starts to transmit telegrams. The telegram is repeated for as long as the balise receives energy. The telegrams are sent cyclically, either amplitude or frequency-modulated. The telegrams are modulated in a way that one bit is *energy* while zero is *no_energy*. Telegrams in use by current systems typically have a fixed length of 64 bits, 255 bits, or 1023 bits. In the case of a 255-bit telegram, 190 bits are reserved for user information. The remaining bits are used for control purposes such as error detection.

The telegram data flow is from the way-side equipment through the train antenna, the transmission equipment and BTM to the Train Control System (TCS) and Automatic Train Operation (ATO) systems inside the train.

The BTM subsystem handles the reception of balise information, filters out noise and disturbance, and decodes and forwards the information to other TCS subsystems. Blocks of the BTM system reside on different computers residing on two separate PCBs (Printed Circuit Board). BTM also tests the antenna, the transmission systems and the processor at regular intervals.

The BTM cyclically receives balise data protected by the CRC (Cyclic Redundancy Check), during the balise passage. Telegrams are for example protected by a CRC64 code; data transferred internally in the TCS are protected by a CRC16 code.

The safety concept of the BTM system is based on single hardware and multiple software (one processor unit with two software implementations of the same software specification). It is expected that the n-version programs will fail independently or nearly so. The data received by the BTM system is analyzed and decoded by the 2-version software (A and B safety software). A and B software is developed by two independent teams that are not allowed to exchange information about the development. The B programs use bit-inverted values and tables with the index in reversed order. At certain points in the execution the results from A and B are synchronized and compared.

After decoding, telegram information is sent to other subsystems in the TCS. One subsystem checks the current speed versus the permitted speed and supervises the passing of signals along the track. In case of failures (red signal passed, speed is too high), brakes will be applied. Balises are also used as a reference point for precision stopping of the train, e.g. at metro line stations.

The case study system is a medium-sized software program (80 - 100 kLOC) implemented in C and assembly language. The time-critical parts are implemented in assembly language, while the rest of the software is implemented in C. The development hardware platform is different from the execution hardware platform.

The case study is a family of systems with a high degree of commonality. Functional requirements are similar in all systems. It can be assumed that the architectures of the systems are similar and that architectural elements (components) exist that are used by all members of the application family. This opens a number of interesting questions. How can the 'common' architecture of a legacy system be extracted? How can the architecture of a family of systems be described and how can new family members be derived from the common architecture?

It is common to develop embedded software systems with the least powerful computers that meet the functional and timing requirements due to cost pressures. The case study is no exception.

6.1.2 Main differences between the case study systems

	Spain	**Finland**	**Korea**
Application field	metro (precision stop)	train	metro (precision stop)
Hardware	2 platforms connection: serial link	2 platforms connection: serial link	1 platform connection multi-function vehicle bus(MVB)
Programming Language	Assembler, C	Assembler, C, C++	Assembler, C, C++
Information Location (IL)	IL consists of 1 transponder contains fixed information	IL consists of 2 transponder fixed information and controllable information	IL consists of 1 transponder contains fixed information
Transponder Telegram Format	180 bits, 45 nibbles	180 bits, 45 nibbles	180 bits, 45 nibbles
Scheduling	TRAN: loop DECO: monitor 10 task	TRAN: loop DECO: monitor, SDL-RTS	BTIU: operating system with 3 vital-tasks, 3 +n non-vital-tasks

Table 5: Main differences between the case studies

6.1.3 Architectural issues related to the case studies

The four most important architectural factors of the case study systems:

- **Safety** : The main architectural issue for the case study is the ability to model safety concepts. Whether the safety concept is based on diversified software on the same platform or redundant hardware, it must be possible to describe and discuss safety issues at the architectural level. The safety inspectors can then be involved at an earlier stage of the development process of a new train control system. The current practice is to specify the system and block levels from a functionality point of view, without focusing on safety issues. This resulted in software changes in system composition late in the development process.

- **Hardware evolution** : The hardware of the BTM evolves over time with increased processing capacity. Future systems' processing power will enable higher data transmission speeds for performing CRC-checks and deciphering incoming telegrams. Due to hardware evolution, the software architecture must allow the exchange of hardware modules including the controlling software modules. Describing the processing requirements and timing limitations at an early stage will detect capacity limitations.

- **Railway standards** : The evolution of the BTM is mainly controlled by upcoming proposals for railway standards. The aim of the standards proposals is to enable higher data speeds and longer telegram formats. In addition, the evolution is driven by the need of the railway authorities to run a train through several countries, and therefore deal with different telegram formats and telegram data speeds without changing the locomotive or its ATP system. The current interface and component descriptions do not reflect the ability to develop composite BTMs capable of handling several telegram formats and speeds.

- **Upward compatibility** : It should be possible to do a software upgrade in the future to handle the future telegram format. The current software architecture does not easily handle upgradeability.

6.2 Status at the beginning of the re-architecting project

Source code and documentation about all case study systems were available. System experts were at hand to answer all questions related to the case study. Program comprehension was therefore only a minor issue.

A kind of architectural description of the case study members was available in the form of block-line diagrams representing components and connectors. Such diagrams give an intuitive picture of the system while leaving many open questions. Are blocks always representing components? If they are, are they representing a set of related functions or processes or files? Do lines representing connectors describe control flow or data flow or both? Are there time limitations associated with the exchange? Does software or hardware implement the represented functionality?

In addition, SDL (Specification and Description Language) [BH93] specifications made from the above descriptions, existed for each subsystem. The SDL specifications formed the basis for the software implementation (automatic code generation from specification).

The architectural descriptions of the case study were often incomplete, inconsistent and described with notations that changed from document to document. The descriptions could be checked by using predefined syntactic or semantic rules and no simulation of the systems' behavior was possible. A description of a reference architecture was not available at all.

The software architects were usually designers who were not aware of their architecture-related work. At the beginning of the project, a questionnaire was sent out to fourteen software architects of real-time systems. Some of the architects were also non-case-study-related architects. The questionnaire concerned the requirements of the software architects for architectural recovery. The chosen software architects were members of a software development team responsible for the specification of the architectural style (or the gross organization) of a software application. The software architect determines the style of the overall system. The individuals questioned were experienced software architects and who had participated in at least two large projects. Six architects filled in the questionnaires and five architects were interviewed.

One interesting observation is that the interest in using architectural recovery techniques is high among the architects. Most of the architects reuse design solutions from past development projects in later projects and recognize the need for support in describing high-level design components. The introduction of new team members in development and maintenance projects is a critical aspect and should be supported by a clear and precise documentation including architectural information.

The answers were wide-spread regarding the architectural style used in the current projects. No typical architectural style could be identified. The software architects currently use the following architectural styles: layered, sequential, client-server and pipeline. Timing constraints, safety (reliability) aspects and quality aspects in general are seen as the most critical aspects in the high-level design process for real-time systems. As the result of an architectural recovery process, the software architects want to obtain graphical representations of the legacy architecture. The graphical representation should be based on clear semantics and supported by a browser, viewers and with an integration of design documentation. The software architects indicated that documents (requirement and design documents) and source code (C/C++) should be the main input source for the architectural recovery process.

6.3 The application of SARM to the case studies

6.3.1 Introduction

In chapter 4 the SARM (Software Architecture Recovery Methodology) was explained in detail.
The first idea was to simply recover the structural architecture of the case studies. Because architectural information was missing, architectural properties were recovered and since none of the ADLs was able to describe the recovered information in a satisfying manner ASDL was developed. Finally, it was necessary to develop an approach for recovering the reference architecture (commonality analysis).

In this chapter, experiences of developing SARM and applying SARM to three case studies are described.

6.3.2 Step 1: Prepare input

6.3.2.1 Textual representations

Step 1.1: Preparation of textual representations :

- Use the development process description to identify what kinds of documents are typically produced. Collect the documentation and order it per topic.

In medium to large-sized projects, the number of different product documents may be surprisingly high. The textual representations consist of reports used by software developers in the analysis, design and implementation phases. Documents are either application-specific or of general use, e.g. programming conventions. In this phase the document types are collected and ordered, e.g. by project phase.

In the case study the following documents were available:

Phase	Product Document
Project- and Product Definition	Requirements Specification, Use-Case Description
System Analysis	Man-Machine Specification, Interface Specification
System Design	System Specification, Subsystem Description, Block Description, Architecture-Specification, Requirements Specification
Realisation / Integration	Block Specification, Architecture Specification, Requirements Specification, Technical Specification for external products, Interface Specification, Module Description, System-generation Information, Program Description, Interwork Description
Internal Acceptance	System Description, Product Description

The phases were defined in a document containing the development process description, which defines the phases and the resulting documents of any software-related project.

Step 1.1: Preparation of textual representations :

- Select documents that are of help for the architecture recovery process.

Not all of the document types mentioned above are of relevance to the architecture recovery process. Some of the types are not instantiated because they are not needed for the domain, e.g. man-machine specification, and others contain information that possesses no architectural information, e.g. program description.

Document types were selected if they typically contained information about

- the structure of the components of the program/system
- the interrelationships and interfaces of the components
- principles and guidelines governing the architectural design

The selected document types of the case studies are represented in Figure 25 and are treated as views for the recovery process.

View / Case Study	Spain	Finland	Korea
Subsystem Description	x	x	x
Block Description	x	x	x
Program Description	x		x
Interwork Description	x	x	x
Safety Standard Descriptions	x	x	x
Architecture Specification	x		x

Figure 25 - Relevant textual representations

Subsystem Descriptions are the top-level documents for a subsystem and are used as the main input for writing block descriptions and block test specifications.

The subsystem description of the case study mainly provides information about the subsystem itself, the blocks of the subsystem and the communication between the blocks. The document is based on the system specification of the case study. Some images can be found in the document, e.g. a data-flow oriented subsystem overview and a message sequence chart describing a start-up sequence. The document contains general information about the subsystem (also HW-specific information) and requirements for the application developer. The requirements are not numbered or highlighted in any other way.

The **Block Descriptions** of the experiment provide requirement and design information. Design information, for example, describes the definition of data fields and modules. Some documents contain a number of SDL diagrams in the annex describing the behavior of the block-modules.

A specific problem for this type of document is the level of detail. The software will be developed as A and B security-duplicated programs in which the block description is similar for both programs. If too much information is put in common documents, the A and B programs may not become sufficiently independent. The two development teams would develop similar software that would reduce the safety of the overall system. The block documents represent therefore a compromise between security, reliability and details.

Program Descriptions are documents describing how blocks will be implemented. For each block, there will be two program descriptions: one for the A software and one for the B software. The program descriptions (except in the Finland case study) describe the functionality in SDL. Additional text is used for clarification when necessary.

Interwork Descriptions are documents describing the interfaces between subsystems. Interwork Descriptions describe a subsystem's classes, functions and variables accessible from other subsystems. The case study's interwork description describes the public functions, including usage, parameters, and return values.

Safety Standard Descriptions are documents describing safety standards that are valid for all product development processes and for all kinds of safety-critical software concerning the business area of the case study.

Architecture Specification contains a short system description, interface and safety description and system decomposition information. The documents contain hardware and software-relevant information. The documents are, however, mostly hardware-oriented and therefore do not greatly impact the architecture recovery process. For the Finland case study, no architectural specification was available.

Step 1.1: Preparation of textual representations :

- Identify state of documents. Remove outdated documents (documents that no longer represent the as-is system).

The description of the documents also contained information about the connection to a specific version of the control system. For the case study, the version number of the system under consideration was available, so it was no problem to check if a document was relevant to the case study or not. The versioning algorithm of the documents also allowed to find out if a document was the current document or not.

Step 1.1: Preparation of textual representations :

- Find documentation that describes the vocabulary used under system development. Such documentation may be of help in the component identification phase.

Basic information (e.g. the subtasks) of the system was explained in different documents. Often, the information was given from different points of view that matched with the document (e.g. in the block specification the information about the most important components differed from the information in the architecture specification). No resource was available that described the most important elements and features of the system in one document. Work would have been easier at the beginning of the project if such a document had existed.

6.3.2.2 Graphical representations

The handling of graphical representations is similar to the handling of textual representation. Collect information about available graphical representation, select and prepare relevant information and use it for architectural recovery.

Step 1.2: Preparation of graphical representations

- Use the development process description to identify what kinds of graphical representations are typically produced. Collect the representations.

In the case study systems, graphical representations were often used to enhance information in the textual representation. By collecting textual representation, graphical representations were also collected. Besides SDL diagrams, no other formal notation was used to represent information about the system. Graphical representations are typically created in an ad-hoc manner without using a defined notation but are in most cases self-explanatory for the designers.

- Block-line diagram describes elements in the system (e.g. components, sub-systems, and hardware elements) and connections between the elements. In most cases the diagrams describe the static aspects of the system.
- Data flow diagram and similar diagrams (e.g. describing a specific flow of information through the system) describe dynamic aspects of the system.

- SDL specifications describe the dynamic behavior of a system and are based on the extended finite state-machine concept.

Step 1.2: Preparation of graphical representations

- Select representations that are of help for the architecture recovery process. Representations covering the system specification and early design phase are probably the representations with the most architecture-related information.

As explained in the previous step, graphical representations are part of the textual representation and by collecting relevant textual representation, the relevant graphical representation is also collected.

A special case concerns SDL diagrams that were developed as a specific part of the development process and are mainly independent from textual representations. SDL diagrams are definitely relevant for architectural recovery since they already model systems on the highest-level of abstraction.

Step 1.2: Preparation of graphical representations

- Identify the state of the representations. Remove outdated representations (representations that no longer represent the as-is system).

The textual representations identified in step 1.1 contain graphical representations that are up-to-date. Therefore, no specific activities were necessary to remove outdated graphical representations.

SDL diagrams of the case study did not have a highly sophisticated versioning system as the textual documents. It was therefore more difficult to identify the correct version of the diagrams. The correct versions were identified mainly by asking the designers and checking the cross references from documents to SDL diagrams

SDL diagrams

SDL (Specification and Description Language) [BH93] is a specification language with concepts for describing reactive and discrete systems. SDL representations are textual and graphical. SDL was originally specified as a language for telecommunication systems. The language is, however, not restricted to telecommunication but is also generally useful for specification of real-time and embedded software systems that follow the stimuli-response types of behavior.

The basic idea of SDL is to model independent behavior as concurrent processes. Processes influence each other and the environment by the exchange of signals. Each process consists of a finite state machine that is triggered by input signals. Signals carry the identity of the sender and perhaps additional data. The finite state machine consumes the signal and performs a transition. Actions may be executed in transitions. Assignments of values to attributes, sending output signals or calling procedures are typical actions in transitions.

In the case study, SDL was used for specification in the Spain and Korea case studies, but not for the Finland case study. SDL-views may be expanded by message sequence charts (MSC), block diagrams or object models used for decomposition of the domain into smaller pieces. In the case study, only message sequence charts were used to represent additional information.

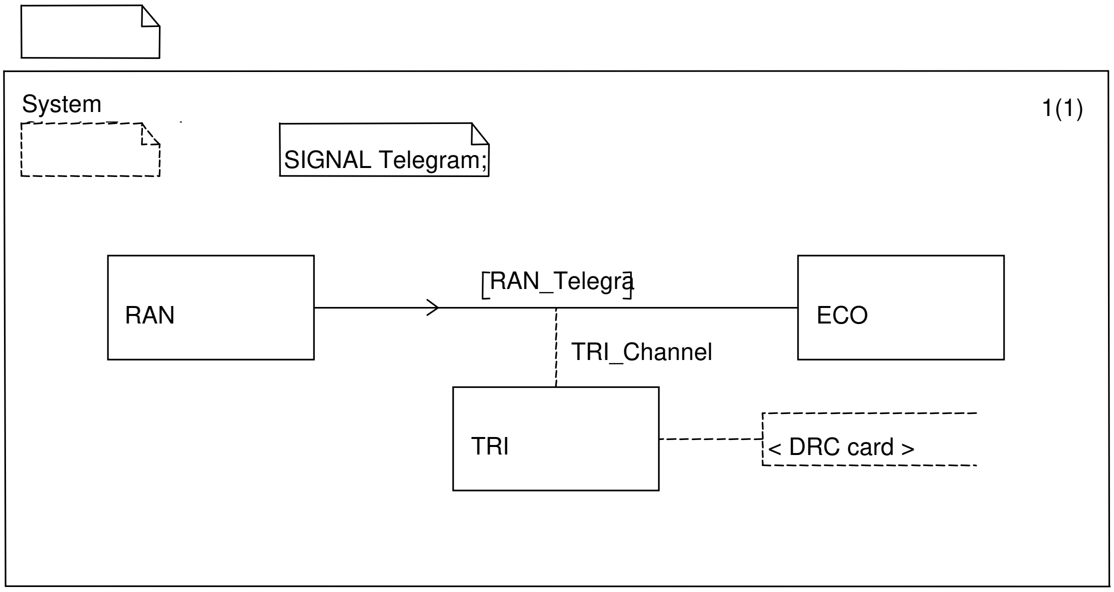


Figure 26: SDL system diagram

Figure 26 shows parts of a system in the case study. Blocks represent components and lines are signal lines. The informal textual attachment (<DRC_card>) comments that the TRI component is executed on the DRC hardware platform.

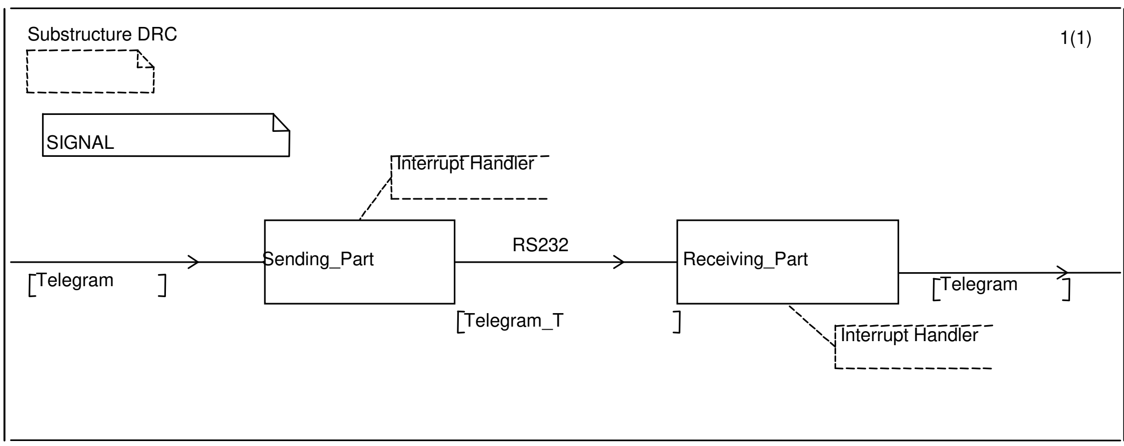


Figure 27: SDL sub-component

Figure 27 shows the sub-components of the TRI channel from above. The channel contains a sending component, a receiving component and the connection.

SDL is a formal specification language in which each notation has its specific syntax and semantic. SDL specifications may be verified for correctness with tool-support. SDL textual representations allow the transformation into programming languages or the automatic generation of source code.

Step 1.2: Preparation of graphical representations

- In case non-standard notations are used, identify:
  - What is the meaning of symbols (does a block mean that the element is a software element or a hardware element or both)?
  - What is the relation between diagrams? Are they conflicting or complementary?
  - Is there an underlying semantic in the diagrams?

In the graphical representations of the case study, non-standard notations are typically used.

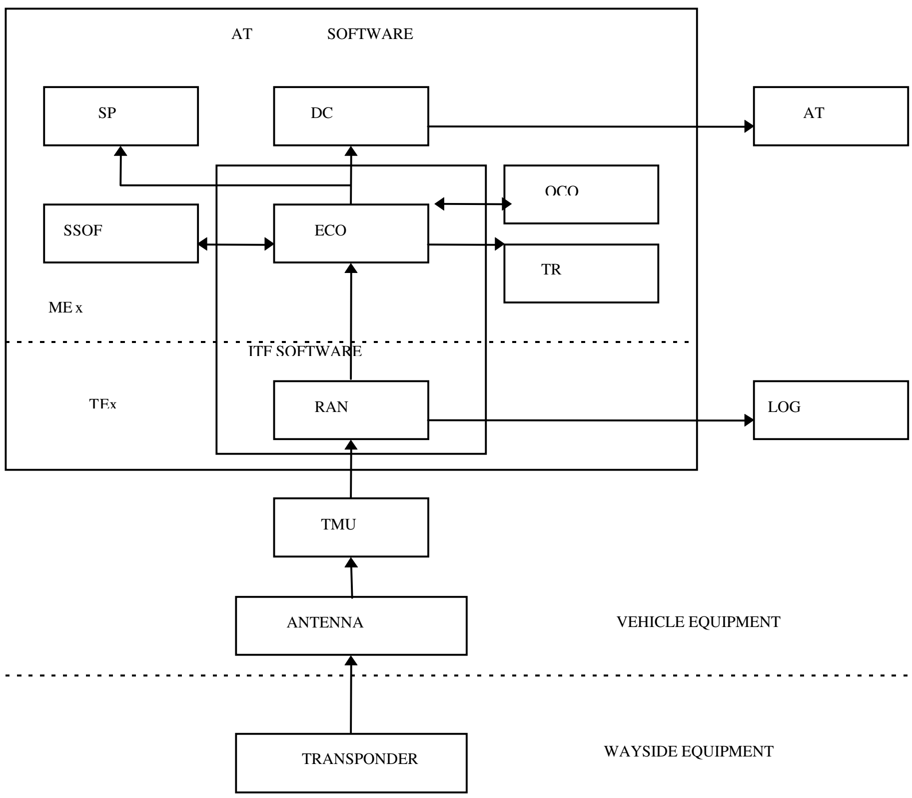


Figure 28: Component diagram example

Figure 28 shows a typical component diagram in the case study. The blocks represent composite and primitive components and, in addition, hardware platforms. The dotted line divides the MEx platform from the TEx hardware platform. The component *RAN* is executed on the TEx platform and the component *ECO* on the MEx platform. Both components are part of the composite component *ITF software*. In addition, the graphic contains information about equipment of the train which is a physical component without any logic (e.g. transponder). The lines with the arrowhead indicate the flow of information through the system.

The notation is not declared anywhere in the rest of the document where it was found. Many system shareholders are however used to such representations and correctly interpret the diagram without any need of further information.

To transform such a description into ASDL, it was necessary to discuss it with a system expert to find out the exact meaning of each of the symbols. It was also necessary to exclude information that was irrelevant for architectural recovery (e.g. hardware platform). If hardware platform is relevant information, a corresponding property in ASDL should be declared to allow the representation of this information.

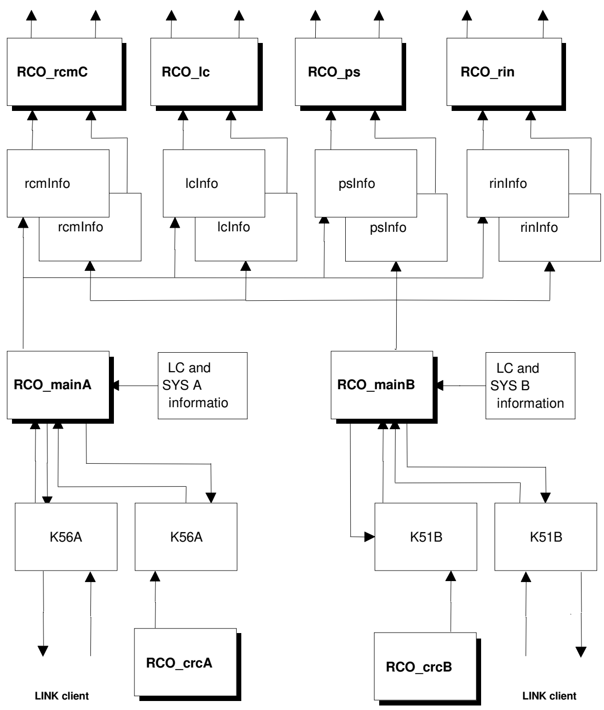


Figure 29: Example of data flow

Figure 29 shows a typical representation of data flow in the system without being a complete data flow diagram. The data that flows between the blocks is for example not presented. The blocks represent components and the lines connectors where data flows from one component to another. It was important to analyze the correspondence to the component diagrams. In some cases, components represented in the data flow diagram were not represented in the component diagram and the same is also valid for connectors. One of the diagrams was so incomplete that some information was missing. Often, different levels of abstraction were used in the diagrams. The sub-components of a composite component, which was presented in the component diagram, were presented in the data flow diagram. Since in some cases the relation between the sub-components and the super-component was never clearly represented, problems in interpreting the diagrams occurred.

6.3.2.3 Source Code

Step 1.3: Preparation of source code

- Identify the correct source code version, including all necessary files and directories.
- Identify the code architecture of the system. The distribution of source code over directories and files is the code architecture of a system.

The code size of the case study was approximately 150 kLOC. The software was implemented in C, C++ und assembler language. The time-critical parts of the systems were implemented in assembler language.

A clear relationship exists between the textual representations and the source code. Components seen in the block description documents are source files in C++ or C. Each source file has a corresponding header file. The classes, functions, and variables are described in the interwork description and in the related subsystem header file.

The source code was developed and implemented according to in-house programming standards. Source files are managed by a version control system, where the files are placed after an accepted module test has been performed. It was therefore no problem to identify the correct source code version including all source files, header files, make files and, additional configuration files.

Step 1.3: Preparation of source code

- If reverse-engineering tools are used, prepare the source code such that tools can perform the source code parsing.

Reverse-engineering tools were used as support in the architecture recovery process (see section 6.3.3.1 Tool and manually generated views). C and C++ code was analyzed by vendor-made tools, while the assembler code was analyzed by a custom-made tool developed at the Technical University of Vienna.

Most of the preparation activities were necessary because of the difference between the development platform (DOS) and the recovery platform (UNIX). The preparation included:

- Conversion of the file format of source files from development platforms to the recovery platform (DOS to UNIX).
- Conversion of source file names (e.g. from upper-case character to lower-case character)
- Acquisition of platform specific files (e.g. header files such as dos.h)
- Exclusion of files that were for test or simulation purposes
- Identification of the correct compiler flags that represent the correct system version.

It was important to see that the source code was compatible with standards (e.g. ANSI C). Otherwise, the recovery tools would not have been able to analyze the code.

6.3.2.4 Human knowledge

Step 1.4: Preparation of human knowledge

- Make certain that system experts for the recovery process are identified and available for supporting the recovery process
- Inform the system experts about the advances of software architecture recovery to gain a positive attitude for the recovery process
- Ensure that the system experts are familiar with the different recovery steps and the objectives of the recovery project

The identification of system experts was not difficult because some were directly available at work (in Oslo) and others in another ABB company in Stockholm. Most of the system experts were working on the case study systems for a number of years and continued working on new versions and/or new members of the product family.

The system experts were informed about the architecture recovery work and the different recovery steps in a number of presentations. At the beginning of the project, a questionnaire was sent out to the system architects (for more information see section *6.2 Status at the beginning of the re-architecting project*). The interest in using architectural recovery techniques was high among the architects and they recognized the need for architectural recovery. The basis for good cooperation for the work was laid.

It was important to have system experts at hand to understand the actual behavior of the system. In addition, the system experts are at the same time domain experts who know a lot about the domain behind the actual application. The case study belongs to the domain of railway systems and to the domain of real-time and safety-critical systems. Domain expertise allows one to understand why the system was designed in the way it was. Such information can rarely be found in system documentation.

6.3.2.5 Summary

The time frame between the beginning and the end of step 1 was rather long. The first activities of step 1 had begun before the official start of the ARES project. To give input for the definition of the project, it was necessary to identify a case study that would fit for the architecture recovery process. Since the ARES project was a research project and the SARM was developed as part of the project, some activities for step 1 were performed in the latest phases of the project (when input was needed for the reference architecture identification).

Although no architectural information is recovered in step 1, all other steps are directly depended on the results of this step of SARM. Since a number of people are usually working on the recovery, it is necessary to store results of step 1 in a specific place so that no system documentation will be changed on one side by the recovery team and, no confusion will come up on the other side, because the re-architects work on different documents from different sources. The effort for collecting and preparing the representations in step 1 should not be underestimated.

6.3.3 Step 2: Extract structural architecture

The direct recovery of architectural elements from the results of step 1 was possible. However, since architectural elements (components, connectors and interfaces) are not always described in a way that allows the direct recovery, additional work was necessary. By generating additional views, the input for recovery was qualitatively and quantitatively enhanced. The additional views were then important sources for the recovery process.

6.3.3.1 Tool and manually generated views

Step 2.1: Tool-supported high-level view generation
- Extraction
- Abstraction
- Presentation

The case study system was a medium-sized system (150 kLOC). To analyze source code of this size, tool support is necessary. Source code contains all essential information for architecture recovery (although additional information is necessary to fully understand and recover a system).

The case studies written in C and C++ allow for the application of re-engineering tools. Tool-supported high-level view generation of the case study was mostly done by colleagues from the Technical University of Vienna. Bellay and Gall [BG98] evaluated, as part of the ARES project, four reverse engineering tools (Refine/C, Imagix4D, Rigi and SniFF+). The main focus of the evaluation was on the ability of the tools to generate graphical reports (e.g. call trees, data and control flow graphs). The evaluation showed “that the performance and capabilities of a reverse engineering tool are dependent upon the character of the analysis purpose and upon the application domain "[BG98] p.328. Additional information about the results of this work can be found in [Bel97] and [BG97].

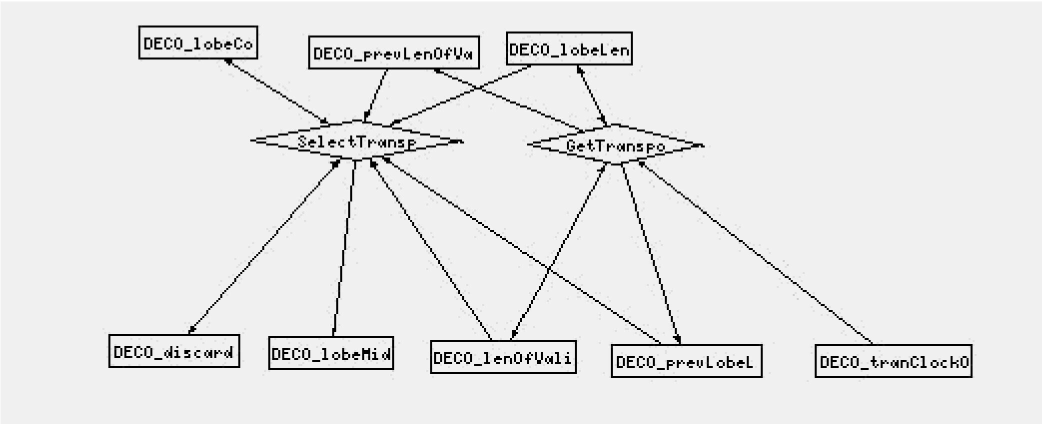


Figure 30: Data flow graph produced by Refine/C

Figure 30 shows a data flow diagram generated by the reengineering tool Refine/C. Data is represented by rectangles and functions by prisms. The lines indicate that the data structure is used by the function.

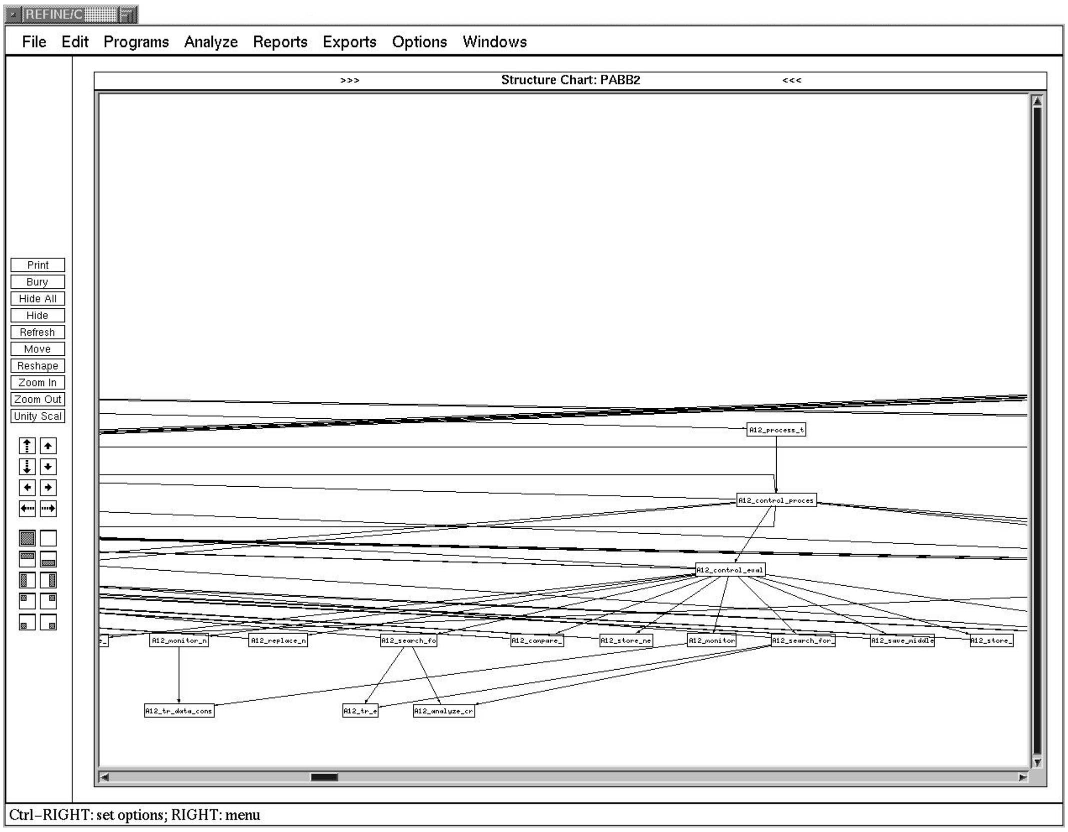


Figure 31: Call graph in Refine/C [BG98]

Figure 31 shows a call graph as generated by Refine/C. The call graph is so large that only a zoomed presentation (as in Figure 31) produces a readable format. Functions are represented by rectangles and lines indicate a call of a function. The caller is always presented in the level above the called function. The figure indicates that the case study consists of a relatively small number of central functions that call a relatively high number of other functions.

The reverse-engineering tools were helpful for architecture recovery because they generated higher level abstractions of the source code in a very short time. To develop a call graph, like the one in Figure 31 by a re-architect, is a time-consuming and sometimes frustrating task.

The problem with the generated views is that they are rather cluttered and unreadable. In addition, many of the generated views were incomplete because the tools were not able to analyze code written in assembler language. It was on the one hand necessary to filter out information from the views to identify an interesting aspect of the system (such as a component consisting of some functions). On the other hand, it was necessary to enhance the views to produce a complete picture of the functionality of the system. It was also not possible to introduce additional information such as domain or application knowledge into the generated views, which would have supported the re-architect in his/her work.

The reverse engineering tools did not directly support the identification of components and connectors by the application of some kind of pattern recognition. The interface for other subsystems of the case study were mainly implemented as shared memory and interrupt functions were written in assembler language. The reverse engineering tools failed to identify such information. It was therefore necessary to manually produce such views to represent the interface information.A hybrid approach, integrating tool-supported reverse engineering and knowledge acquisition from human experts, seems to be the appropriate solution to the identified limitations [GKM95].

Step 2.2: Manually generated high-level views

- Extraction
- Abstraction
- Presentation

Manually-recovered views are software views that are recovered by humans with or without the support of an reengineering tool. Manually-recovered views were extracted in parallel to tool-supported views in the case study.

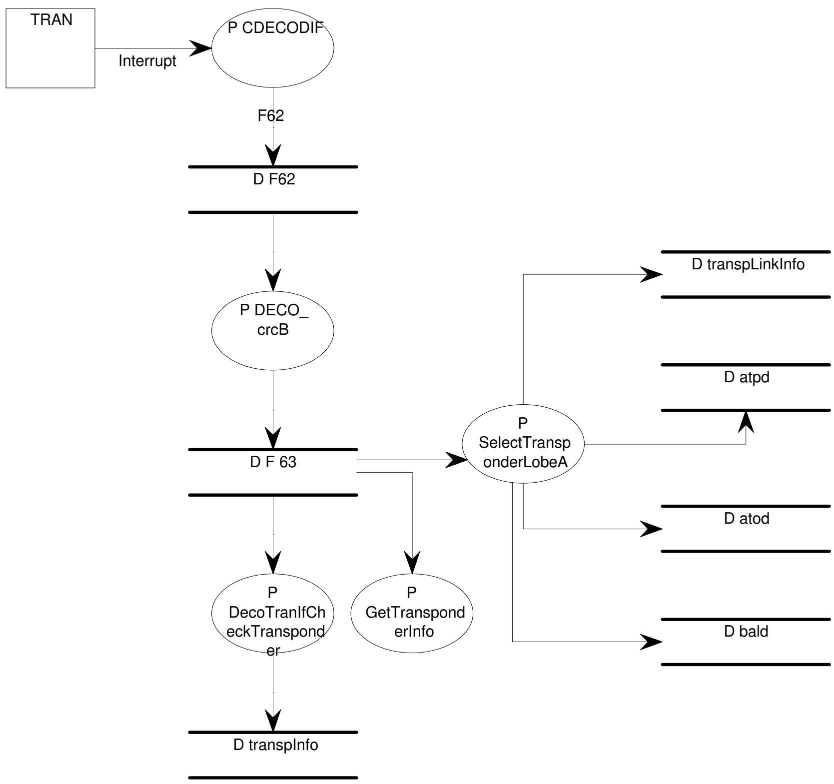


Figure 32: Manually-recovered data flow diagram (partially shown)

Figure 32 shows an example of a manually-recovered data flow diagram. The rectangles represent subsystems, the ellipses processes, the connecting arrows represent data flows and data stores are represented with two parallel lines.

The data flow diagram represents the flow of a specific data structure (a telegram) through the system. This view could not be generated by any of the examined reverse engineering tools. The manually-generated views present the information in a way that is easy for human readers to read (e.g. no zooming or filtering is necessary).

Manually-recovered views have the advantage that the re-engineer filters only the relevant information out of the available sources of information in the process. This requires knowledge of both the application and the domain, and sometimes additional tool support.

The major disadvantages of manually-recovered views are that the recovery work is very time-intensive and therefore costly. Manually-recovered view can sometimes not be reproduced because of a biased viewpoint of the re-engineer. The recovery process is error-prone because of the fallibility of humans.

Step 2.3: Identification of structural elements

- Component, interface and connector identification

Inherent module or component concepts are missing in the programming languages which were used to implement the case study (an exception are C++ classes). Component identification and recovery in the test-case can therefore not be purely based on language concepts.

A number of possible component identification approaches were considered:

- file structure - Functions and data structures collected in a source code file (including header files) are natural candidates for components. The reason for collecting functions in a file is often functional dependency, which is a strong reason to view such an artifact as a component.
- diversified software - All safety-critical parts of the case study are implemented by diversified software. The style is called A,B,C style. A and B are versions of the program developed by different development groups based on the same requirements. B uses inverted data for safety reasons. C is software that synchronizes, distributes and votes results from A and B. The A,B,C parts of the system are natural candidates for components because each part implements its one specific function that is strongly related, otherwise the designer would have designed it in another way.
- coupling - Coupling is the measure of interconnection among functions in a software system. A high degree of coupling between functions could be an indication that the related functions belong to the same component. A low degree of coupling could accordingly indicate that the functions belong to different components.

- documentation - Design documents and recovered views of the case study contain block-line diagrams and other information that give indication about the existing architecture. Indicators are also the table of contents of components, since components are often described in subchapters.
- monitor - The software execution of the case study is controlled by a software monitor. The scheduling approach is a modified round-robin algorithm. The monitor tasks implement functionality that must be completely performed within a specific time-frame to ensure correct execution. The monitor tasks encapsulate strongly related functionality and are therefore component candidates.

The component candidates were discussed with system experts to find out which would be the most promising approach. The outcome was that each of the component identification approaches had a distinct value for component identification and none were to be excluded.

The recovery work was started by analyzing the design documentation and the recovered views. Block and line diagrams in the documentation indicated that the overall system was built as a hierarchy of components. The block-line diagrams and also the structure of documents (e.g. sub-chapters) provided a list of component candidates on the highest level. The components at the next lower level were identified mainly by analyzing the monitor. The monitor schedules tasks that divide the components into non-overlapping sub-components. The monitor was also the reason for the centralized call structure of the systems (see Figure 31). Safety-critical sub-components were again divided into lower level components identified based on the diversified software principle. The recovery process was stopped at this level of abstraction after discussing it with the system experts. The idea was to stop before moving too deep into the implementation level and losing the overview of the entire system.

The component identification process was an iterative process with a number of processing loops and feedback rounds which included discussions with system experts.

Components in SDL

An architectural component can be expressed in SDL by using either blocks, processes or services.

- **Blocks** - Blocks are used in SDL to hide information from their surroundings. The blocks of a system or the sub-blocks of a block can be easily reused because their interfaces are specified in SDL. Components on the highest level of abstraction (system components) are usually represented by SDL blocks. The Korea case study is built that way. The system components of the system are represented as blocks (e.g. the TIF-software component in Figure 28).

- **Processes -** The further refinement of a block is again a block or a process. A processes has its own private data-space where no other process has access to its internal data. A process is an actor of the system; it can initiate signals and send them to other processes, blocks or the environment. A process is located on one single processor. Processes are typically used to express that components are located on different hardware platforms. The ITF-software component consists of two sub-components: ECO and RAN. ECO is executed on a processor, as is RAN. The two components share no memory with each other.

- **Service -** A process in SDL can either contain a behavioral description or a service interaction diagram. The service interaction diagram splits a process into further sub-components which interact through signals, but share data inside the process. Services are on a lower level of abstraction where the re-architect must decide if the service is treated as a component or not. Sub-components of RAN are modeled as services and were not modeled as a component in the case study.

The connector and interface identification process was an integral part of the component identification process. Components were analyzed for their interactions with the environment (i.e. other components). Services provided and required by the component became part of the component interface. Services exchanging data or control information were linked by connectors. The type of information exchange in the case study is mainly based on shared-memory and function calls.

Case study system	Number of identified components
Spain	41
Finland	43
Korea	19

The table shows the number of components that were identified for each of the case study systems. The case studies *Spain* and *Finland* belong to the same version group, whereas the *Korea* case study belongs to a version group that was developed after the other two case studies.

6.3.3.1.1 Examples of recovered components

The example shows part of the Finland case study. The architectural elements are described using the ADL Darwin. The components where recovered from documentation and manually-generated views.

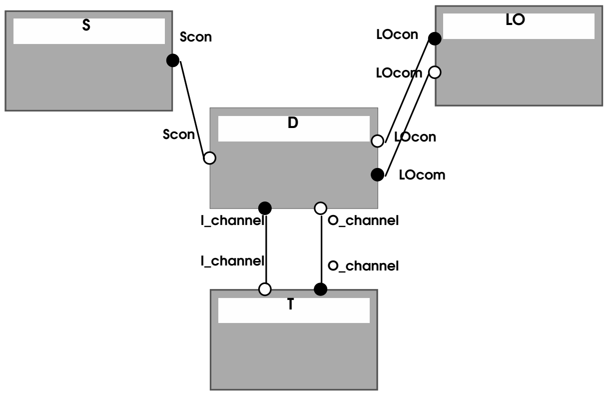


Figure 33: Components of the case study

Figure 33 shows the system level of the system. Components are executed on different hardware platforms (which can not be shown in the graphical representation of Darwin). The D, S and LO components are executed on the same hardware platform; the T component is executed on another hardware platform. The HW-platforms are connected by a serial link. The components play different roles. The T component works as a server for the D component. The D component, as a client, sends commands through the O_channel and receives data through the I_channel from the server T. Towards the S component, the D component plays the role of a producer. The connections between the components are based on shared memory and function calls. Towards the LO component, D plays the role of a server and LO is accordingly the client.

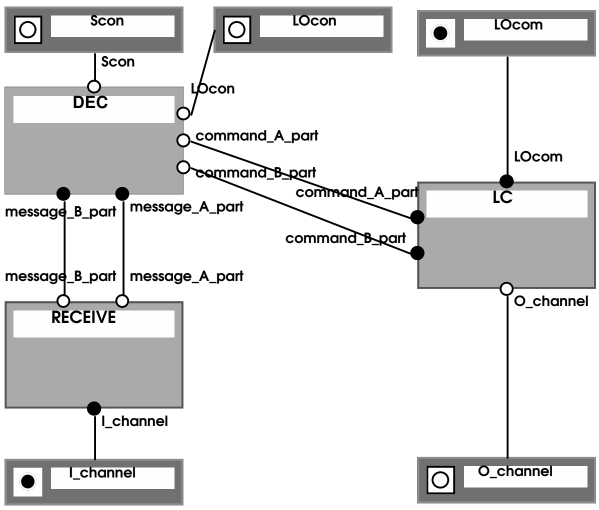


Figure 34: The sub-components of D.

Figure 34 describes the sub-components of the D component. The DEC and LC component are scheduled by the monitor (which is not visible on the diagram). The connections between the components are shared memories. The DEC component is a fail-safe component which consists of A, B and C components. The DEC component has therefore A and B inputs and outputs. The A and B parts of DEC controls the T component through the LC component.

6.3.3.2 Summary

The recovery of the structural architecture is a key-activity in the recovery process. All further recovery steps are directly dependent on the results of this step. In the recovery process, the component recovery was the most important part, since by recovering the components the interface and connectors were identified.

Different approaches were applied for the recovery of components. The most accurate input sources were the SDL diagrams. In many cases, a direct transformation from a SDL-block to a component was made. However, since only parts of the case study were modeled by SDL and in the case of Finland no SDL diagrams were available, other input sources were also used.

The documentation and the recovered views were, besides SDL diagrams, the best input for the recovery process. Both input sources delivered component candidates that were directly transformed into components in the architectural description of the case study. Tool and manually-recovered views were helpful in identifying components and as a verification mechanism for deciding if a candidate was finally transformed into a component or not.

6.3.4 Step 3: Extract architectural properties

The recovery of architectural properties described in this section was restricted to properties that are defined in *5.1.3 Property Base* (e.g. structure properties of components). The recovery of other properties of the case study was mainly done by the Technical University of Vienna (TUV) and is described in section *4.4. Common Architectural Properties* in the ARES book [JRL00]. The TUV viewpoint on architectural properties is system-wide-oriented and not as strongly architectural-element-oriented as presented in this work.

6.3.4.1 Steps

Step 3.1: Analyze documents (e.g. requirements specification, design documentation)
Step 3.2: Recover information from system experts
Step 3.3: Analyze source code

All three steps of the property recovery are described in one section, since most of the properties were identified by a combination of document and source code analysis and questioning system experts.

The following figure describes the case study as a layered system. The representation of the system in such layers was never documented in any of the input sources that were available for the recovery work. It is the result of a discussion with system experts. The system experts also explained that this would have been difficult (or impossible) to recover from source code.

Application layer (e.g.PS)

Datalink layer (RAN, ECO)

Physical layer (Airgap)

Figure 35: Case study as a layered system

The flow of information in the system is from bottom up. Telegrams sent by the balise are received by elements of the physical layer (e.g. antenna) and forwarded to the datalink layer. The task of the datalink layer is mainly to verify and decode the telegram and to forwards it to the application layer, where the train control and protection functionality will be performed.

As a result of the discussion with the system experts, it became clear that the datalink layer should be easy to replace with another implementation. In the future, the telegrams will probably not be received from Balises but rather through a different physical system (e.g. radio). The replacement of the datalink layer should not require changes in the application layer, which is not completely achieved by the current system.

Each of the component candidates (identified in the previous step) can be related to one of the layers of the system. Together with the knowledge about the flow of information, important indications for properties (e.g. the main characteristic of a component or the role of the interfaces) were derived.

Architectural properties that are derived by the recovery steps of SARM are directly related to architectural elements (components, interfaces, connectors). The fact that the case study can be represented as a layered system was an important input to the property recovery process.

Other important facts that were detected/recovered and that supported the identification of properties are:

- The execution of the system is controlled by a software monitor. The monitor schedules the implemented state automations and the basic hardware testing.
- The execution of the system is event-driven and information arrives asynchronously. The triggers are the telegrams. Parts of the system functions (mainly system test) are, however, performed cyclic (e.g. antenna test).
- The information exchange within one HW-platform is ‘polling’ based (and controlled by the monitor). The information flow between HW-platforms is performed through a serial communication channel.
- Many subsystems can be divided into three categories: Client/Server, Producer/Consumer and Library. Library was the category that was the most discussed about, since it was a container for components that did not belong to the two other categories. Other categories (e.g. communication control) were introduced to satisfyingly describe the main characteristic of these components.
- Connectors have protocol specifications that define their properties. Connector properties include the following: synchronization information (synchronous, asynchronous), control flow (direct, indirect, none) and degree of coupling. Coupling is the measure of interconnection among components: low (data coupling - data passed as argument list), moderate (control coupling - control passed between components) and high (common coupling - components refer global data area). Control and common coupling are the most used forms of coupling among components in the test case.

Each of these statements gave an input for property recovery because each statement contained important information about architectural elements. The example of the statement about the layered system is applicable to all other statements.

6.3.4.2 Summary

The recovery of properties was not as view-oriented as the recovery of components, but rather more fact-oriented. Examples of facts are the statements above. Each fact describes properties of architectural elements. Some facts were found in the documentation, others while discussing it with system experts. Architectural views (as recovered in step 2) were an important aid for the discussions with the system experts, however did not directly deliver usable facts. Architectural properties are harder to recover than architectural elements. Properties are ‘soft facts’ of a system. It was, for example, sometimes difficult to decide if the main characteristic of a component was *data processing* or *communication control*. In future projects, criteria should be defined that allow a clear assignment if an architectural element fulfills the properties criteria or not.

6.3.5 Step 4: Describe the architecture

In principle any architectural description technique could be used to describe the architecture. ASDL was finally chosen to describe the architectures of the case study systems.

6.3.5.1 Steps

Step 4.1: Choose a description technique

At the beginning of the project, the ADL DARWIN was used to describe architectural information (for examples see section *3.2 DARWIN*). Darwin provides concepts and tools to describe software architectures in a clear and consistent way. Darwin clearly defines input and output of each component, allows the grouping of components and connectors and the parameterization of components.

In addition to Darwin, other languages (SDL and UML) were evaluated as an ADL. SDL was used as a candidate for an ADL because it provides a complementary view to Darwin. SDL supports the description of behavioral aspects (Darwin only the static view). The combination of SDL with Darwin using transformation rules could be an interesting combination of languages. SDL is lacking in its capability to describe architectural properties. The language is standardized and therefore not extensible.

The UML class diagram was used to describe architectural elements of the case study. Information about operations of the class are described in UML, which can be called by any other class (in this case, a class is equivalent to a component). The class diagram does, however, not present the configuration of the components to a system, i.e. which services is required from which class. Darwin diagrams, on the other hand, show the most vital information, such as the components, the connectors and the services provided and required in one single diagram. UML class diagrams are developed for system analysis and design and stimulate the user to think about design and not about architectural design. Classes represent data structures and operations, which are clearly design information. In the architectural design phase, the architect is not interested in data structures and operations but in the overall structure of the system.

Darwin, SDL, UML and other languages (see section *3.6 ADL comparison*) did not fulfill the requirements for an ADL (see section *3.1.1 ADL requirements*) that should be used to describe the architecture of the case study system. Therefore, ASDL was developed and chosen as the description technique for the case study system (see section *5.1.8 Example* for a case study related example).

Step 4.2: Transform input information and describe the system

The input to step 4 are the results from step 2 (collection of structural architectural elements) and step 3 (collection of architectural properties). The collection of structural architectural elements is in principal a list of components (including the component interfaces) and connectors (including configuration information). The collection of architectural properties is a set of architecture-element-specific properties and properties of (sub-)systems.

In the case study, additional information packages in the form of DARWIN and SDL diagrams were available. These collections of information had to be checked for consistency and to be transformed into ASDL. For the case study, the ASDL was designed to describe the necessary properties. For other systems it could be necessary to adapt ASDL such that all relevant architectural information of the system can be described.

Step 4.3: Verify results by system experts

The architectural representation of the case studies was analyzed and evaluated by system experts. There are many ways to describe the architecture of a system. Depending on the re-architect, the recovered architecture will be slightly different. To formally prove that a recovered architecture is complete and correct will be difficult or even impossible. It is, however, important to check if an architecture is “appropriate” in the sense that a derived system from this architecture will fulfill the requirements of the system.

The software architects were asked to check the architectural description for completeness, correctness and for the architectural representation itself. No formal approach was developed for the verification. The system experts simply checked the architectural description and presented their findings in meetings.

- Completeness – The system experts found that the architecture description represented the structural view of the system. It was found complete by the system experts when looking for the components, interfaces and connectors of the system. The system experts missed information about the connection to the hardware platforms and about process control and scheduling. Additional views are necessary, especially to support hardware-software co-design. This was, however, out the of scope of the project.

- Correctness – The system experts found that the architecture was presented correctly. The existence of components, interfaces, connectors and the configuration was verified and accepted. The system experts may have modeled the system slightly differently but accepted the representation as correct.

- Architectural representation – The representations in Darwin and in ASDL were easy to read and to understand for the system experts. The system experts were asked about the match between their mental model and the architectural representation. Darwin and ASDL seemed to match the model, which is not surprising, since both languages present architectural information in a way the system experts are used to. The system experts demanded the existence of a professional tool and professional tool and language support.

6.3.5.2 Summary

An important phase at the beginning of the project was the identification of an ADL that fulfilled the defined requirements. Since none of the available languages fulfilled the requirements, ASDL was developed. The case study was described by using Darwin (only parts of the case study) and ASDL. The architectural representations were checked by system experts for completeness, correctness and appropriateness of the description technique. The architectural representations were accepted by the system experts.

6.3.6 Step 5: Extract reference architecture

To identify the reference architecture, the commonality analysis was developed (see section *4.2.5 Step 5: Extract reference architecture*). In the commonality analysis, components from different family members are identified that appear to be similar.

6.3.6.1 Steps

Step 5.1: Perform component selection

The selection criteria for the case study are: (1) component role, (2) component group, (3) component safety and (4) component type. The selection criteria are derived from the ASDL property base. By applying these selection criteria, it is ensured that appropriate input for the extraction of the reference architecture is prepared.

The selection trees of the case study systems are presented in the following figures.

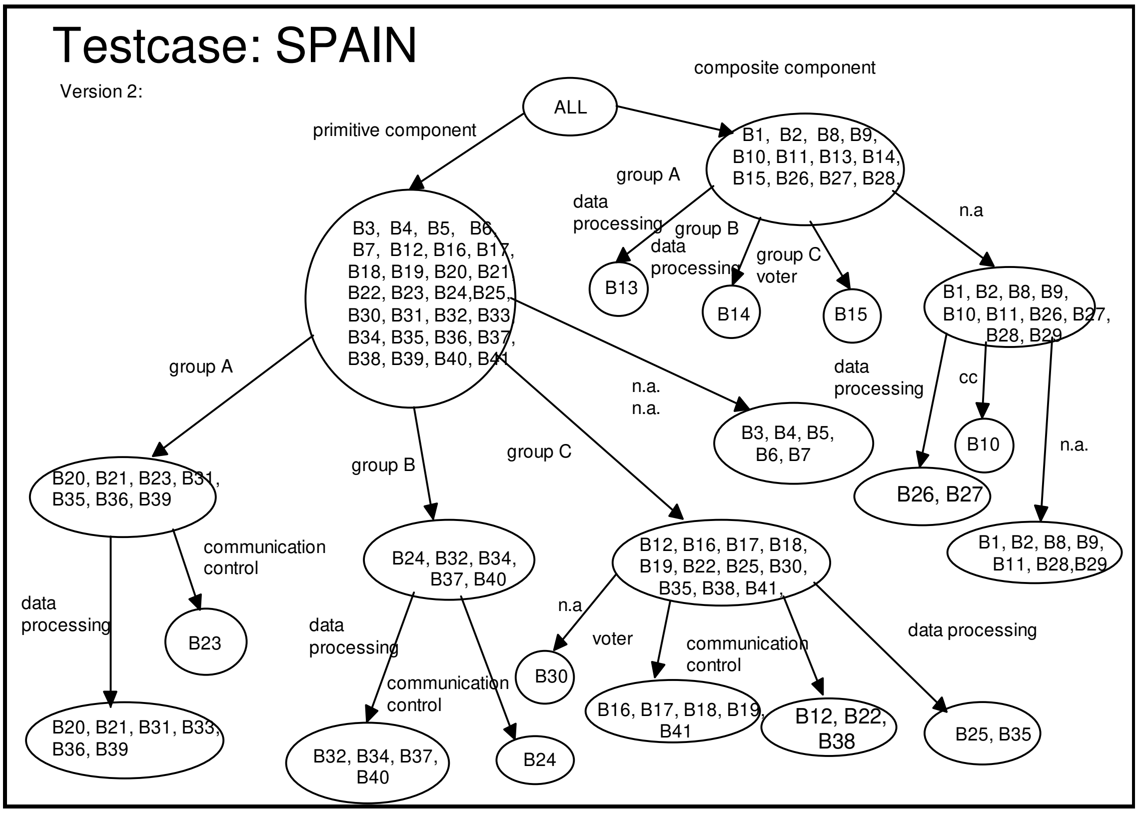


Figure 36: Selection tree of *Spain*

The numbers in the ellipses (representing the set) are the component identifiers. For example, the component B20 is described in ASDL:

comp_begin
comp_id B20
comp_name decotran_a
comp_group comp_A
comp_desc The decotran_a component ...
comp_struc primitive_comp
comp_type data processing
safety_critical yes

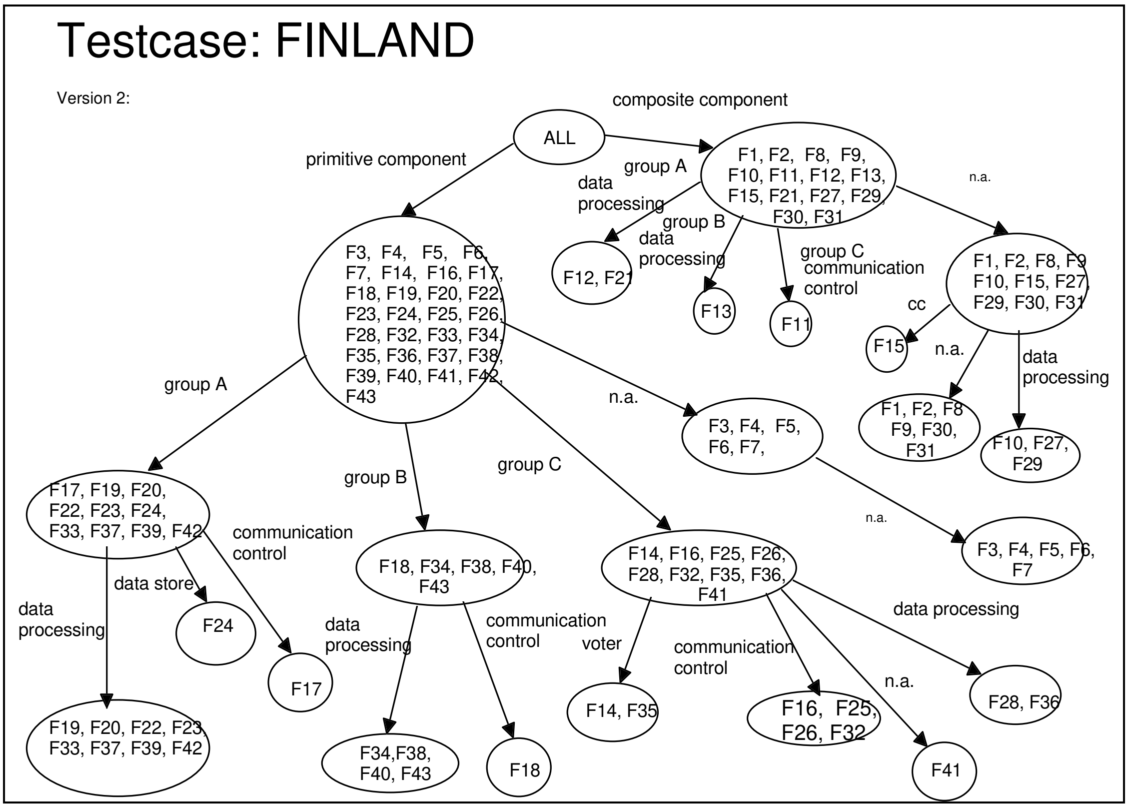


Figure 37: Selection tree of *Finland*

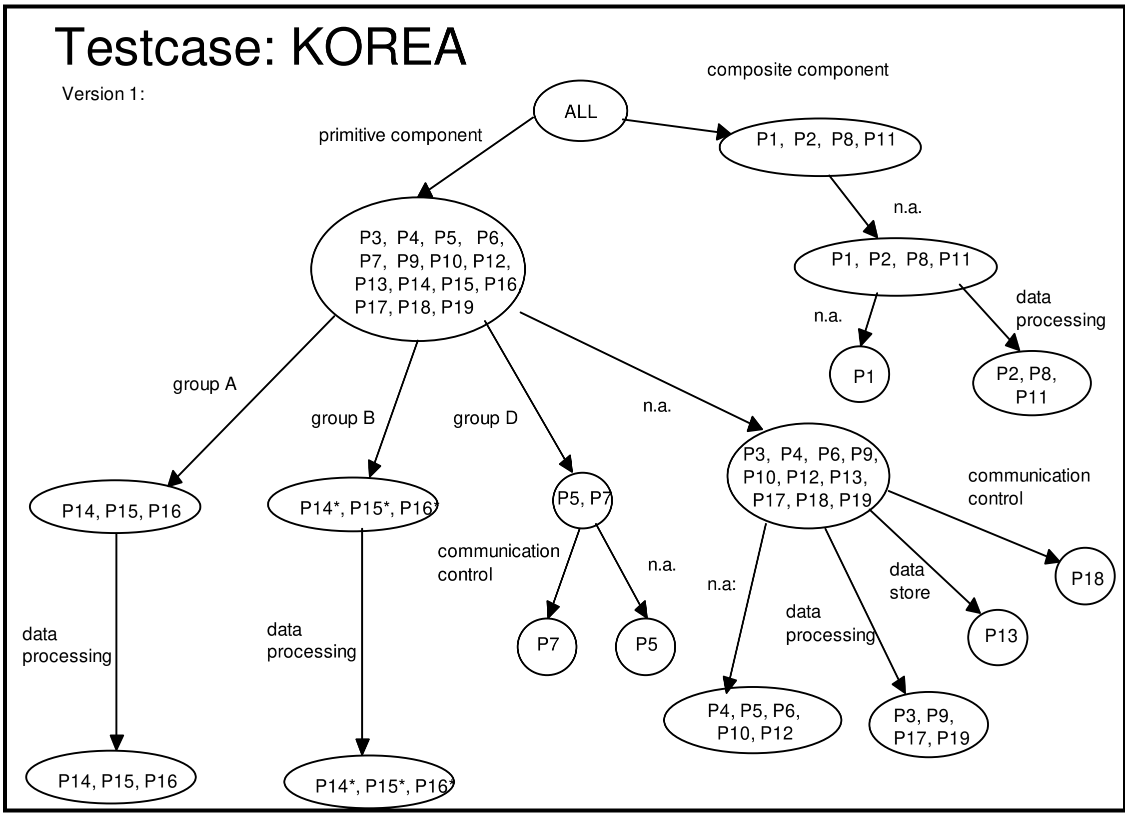


Figure 38: Selection tree of *Korea*

The task of performing the component selection is straightforward and can be supported by a tool.

Step 5.2: Produce matching table

System experts analyzed the leaf component sets of the selection tree. Each component is compared with each component of the other sets.

As an example, the members of the set of *Spain*, that are *primitive components*, that belong to *group_A*, that are *safety critical* and that are *data processing* components, must be compared with the members of the *Finland* and *Korea*:

Spain	B20, B21, B31, B33, B36, B39
Finland	F19, F20, F22, F23, F33, F37, F39, F42
Korea	P14, P15, P16

Figure 39: Set x14a of Spain, Finland and Korea

Figure 39 shows as an example the components of the case study that are *primitive components*, that belong to *group_A*, that are *safety critical* and that are *data processing* components. In this example, each component of *Spain* must be compared with each component of *Finland* and *Korea*. As explained in section *4.2.5.3.2 Step 5.2: Produce matching table*, three outcomes are possible when matching the components: full-match, partial-match or no-match. Important indicators for similarity of components are the component description and the interface descriptions. The result of the matching activity is presented in a matching table.

Full match	F42 – B33 – P14 F20 – B20 – P16
Partial match	F33 – B39 F37 – B36 F39 – B31 F19 – B21
No match	F22, F23, P15

Figure 40: Example of a matching table

Figure 40 shows the corresponding matching table to Figure 39. The system expert found that the components F42, B33 and P14 for example have the same characteristic and provide/require the same services (full match). The components F33 and B39 are similar; however, no matching component of the case study Korea was found. These two components are therefore only partially matching. For the components F22, F23 and P15, no matching component could be found.

Step 5.3: Produce reference architecture

The system experts now have the task of building a reference architecture.

(a) Full-match components are usually replaced by one reference component. In the case study system, the system experts decided that this was the case for both of the full-match results. The pairs of components (F42 – B33 – P14 and F20 – B20 – P16) were each transformed to a component in the reference architecture. The reference architecture component has the same characteristics and provides/requires the same services.

(b) Partial-match components are transformed following as-should-be-architecture requirements. The system experts had a look of each set of partial-match-components and transformed partial-match components into one single component or into another solution (e.g. divide them into a number of components). Especially partial-match components can be transformed into reference architecture elements in many different ways. There are a number of arguments for each of the results and it is not clear which one is the best. It was therefore necessary to develop a direction for the transformation process. The reference architecture should not only represent the architecture of a legacy system but also represent architectural information for future systems. The architecture of future systems of the product line should be derivable from the reference architecture. Finally, it was agreed that in case of difficulties in deciding how legacy architectural information should be transformed into reference architecture information, the transformation should focus on future systems.

This was the first step in the recovery process, in which the re-architects stopped with pure re-architecting and started with forward engineering activities. This also means that the reference architecture is no longer directly related to the legacy systems. If documented properly, the recovered information from steps 2 and 3 can be traced back to information from the case study systems. Most of the results of step 5 may also be traced back to the case study systems, but not everything.

(c) No-match components do not match with any of the components of any other system. It is often hard to decide if the reason for this is a specific requirement for only one system, or if the functionality is hidden in other components in the other systems. For the case study system, the system experts could decide the reason for the existence of no-match components. In some cases, the no-match components were already implemented by components in the reference architecture and could therefore be excluded in the transformation process. In other cases, they provided specific services and were directly transformed into a reference component. Other no-match components were simply excluded since they did not conform to forward engineering requirements.

6.3.6.2 Summary

The most important factors for extracting and building the reference architecture are human experts. Some guidelines are defined on how to transform the components into reference architecture components, but in most cases the final decision is up to the re-architects (and forward engineering architects). About 75% of all components of the case study were partial-match components. The transformation problem needs therefore special attention from both the expert doing the re-architecting and the system architect. Steps 1 to 4 must be performed and verified seriously because, in step 5, the re-engineering process gets intermixed with the forward engineering process and that the ability to trace back the results will be reduced.

6.3.7 Step 6: Complete the reference architecture properties

6.3.7.1 Steps

Step 6.1: Analyze reference architecture elements

The transformation of the properties followed the transformation of the components described in the previous section. If full-match components were transformed to one reference architecture component, the properties of the components were directly transformed to properties of the reference architecture component. If the properties were different, the re-architect tried to anticipate future changes and decided about the actual representation of the property. This was valid for all three component groups (full-match, partial-match, no-match).

Step 6.2: Analyze other input sources

In this task, the re-architects tried to answer the following question: ‘What properties are properties that are valid for more than one architectural element or for the entire system?’.

An example is a standardization property. For electrical/electronic/programmable electronic systems, there is the IEC draft standard 1508 (1) that provides information for all safety lifecycle activities. An important goal of this standard is to consider not only elements of an individual system, but also all safety-related systems making the overall system of a safety-related product. The IEC standard 1508 refers explicitly to software architecture: ‘From a safety viewpoint the Software Architecture is where the basic safety strategy is developed for the software’ (International Standard IEC 1508-1 : Functional Safety : Safety Related Systems Part 1: General Requirements (65A/179/CDV)). In the case study, a specific set of techniques for achieving the safety integrity level has been developed and used. These techniques are data protection techniques (e.g. Cyclic Redundancy Check), fail-safe principles and diversified software principles.

Additional standards exist that are specific for railway control systems. An example is EN 50128 ‘Railway Applications: Software for Railway Control and Protection Systems’. EN 50128 specifies procedures and technical requirements for the development of programmable electronic systems for use in railway control and protection applications. It is applicable exclusively to software and the interaction between software and the ‘wider system’.

Information about techniques for achieving the safety integrity level were represented in ASDL as system properties. Some other safety-relevant information is declared for the sub-set of a component. An example is the safety-integrity level since components may belong to different safety-integrity levels. The way to declare such information in ASDL is via component properties. However, specific concepts are necessary to describe properties of sub-systems.

6.3.7.2 Summary

The transformation of properties of architectural elements into reference architecture element properties follows the same rules as the transformation of components.

The detection and description of system and sub-system architectural properties were evaluated on safety properties. Improvement of both the detection process and the description technique for such properties are necessary.

6.3.8 Step 7: Describe the reference architecture

Also the reference architecture was described using ASDL.

6.3.8.1 Steps

Step 7.1: Transform input information and describe the architecture

The activities in this step are similar to that in step 4. The difference was that the input was purely available in ASDL and that additional information was brought in by *Step 6.2: Analyze other input sources*. The additional properties were, as much as possible, described with ASDL (as described in section *6.3.7.1*, some concepts were missing).

Step 7.2: Verify results by system experts

As in the case of a single system, it is also valid for the reference architecture that there are many ways to describe the architecture. The situation for reference architectures is even more complex since the reference architecture no longer describes the as-is-architecture but moves in the direction of an as-should-be architecture. The reference architectural representation was therefore analyzed and evaluated by system experts and forward engineers.

Again, the reviewers verifying the architecture were asked to check the architectural description for completeness, correctness and for the architectural representation itself. Also for this verification, no formal approach was developed. The reviewers checked the architectural description and presented their findings in meetings.

- Completeness – The system experts and forward engineers found that the reference architecture description was complete.

- Correctness – After some discussion, the system experts and forward engineers found that the reference architecture was presented correctly. Most of the discussion were about the appropriateness of the architecture for future systems and product families. This is once again a topic that could not be decided properly at the end of the project. Only the future will show if the anticipations of future systems in the reference architectures was correct or not.

- Architectural representation – ASDL was accepted by the system experts and forward engineers. Professional tool support and concepts for describing sub-system wide-properties was found missing.

6.3.8.2 Summary

The architectural representations were transformed into ASDL and checked by system experts and forward engineers. The architectural representations were found complete and correct and the architectural representation was found appropriate.

6.3.9 Evaluation

The sections before described the experiences of applying SARM and ASDL to the case study. The key activities were the identification and description of the structural architecture of the case study systems and the reference architecture. In this section an evaluation of SARM and ASDL is presented.

6.3.9.1 Evaluation of SARM

The 7 steps of SARM cover the entire recovery process. No additional steps are necessary. Step 4 (Describe the architecture) and step 7 (Describe the reference architecture) are activities that can be performed in parallel to the according recovery steps, a fact that was not totally clear at the beginning of the project.

Step 1 (Prepare input) is an important activity because all other steps directly depend on the results of this step. The focus on input sources (e.g. source code, human knowledge) will be different depending on the system under consideration. The methods of step 1 will however remain unchanged.

The methods and tools used in step 2 (Extract structural architecture) depend much on the concepts and languages used to produce the legacy system. SARM provides a general method for analyzing legacy systems. SARM must be adapted if detailed information about the system and/or system experts are not available. In this case, specific program comprehension methods must be included which also will influence the way architectural views are produced and analyzed. If the legacy system was produced using object-oriented or component-oriented principles, the identification of structural architectural elements may be simplified because the elements may be recovered directly from the input source. Step 2 covers the recovery of the structural architecture and gives the methodology experts the possibility for adaptation to specific cases.

Step 3 (Extract architectural properties) and step 6 (Complete the reference architecture properties) are the steps of SARM with the most potential for further improvement. SARM focuses on architecture-element-related properties. System or sub-system properties, which are properties that represent the entire system or large parts of it, need in addition specific attention.

The first challenge is the identification of the most important properties that should be recovered. Since properties represent design decisions the number of different properties is rather large. Properties may be domain-specific or domain-independent, functional or non-functional, element-specific or system-specific. A defined process is therefore needed for selecting the appropriate properties for the system under consideration.

The second challenge is the recovery of the properties. As opposite to architectural elements, architectural properties may not be directly recovered from input sources. Recovery tools, which are currently available, do not directly support the recovery of properties but provide only filtered information from which the human expert must recovery the specific information. In addition, each property needs its specific recovery method, which opens a wide field of techniques. The recovery of properties is the part in the recovery process that is most difficult to describe as a generic method. A framework is needed that provides an overview of properties with related proven recovery methods.

Step 4 (Describe the architecture) and step 7 (Describe the reference architecture) depend strongly on the chosen description technique. The problem of finding an appropriate description technique is not specific for architectural recovery but is of general interest for all architecture related activities. The most challenging aspect in the area of architectural description is the transformation of the recovered information to the architecture description technique. SARM is specifically designed for representing architectural information with ASDL. The transformation of architectural elements is straightforward; the transformation of architectural properties is more difficult. ASDL is adapted to properties of the case study and since it is a flexible language it may be adapted to other domains. However, in case other ADL's are used which are not that flexible the transformation of architectural information to an architectural description may be even impossible. Transformation rules, which are generic for a broad range of domains and description techniques, are difficult to identify. However, additional work could be done in this step of SARM to improve the general applicability of the method. Specific information about the evaluation of ASDL can be found in *6.3.9.2 Evaluation of ASDL* .

Step 5 (Extract reference architecture) is the central step for identifying the reference architecture. The extraction is build around the commonality analysis. The selection tree is the structure that is used in the process to perform a component selection. The selection criteria's are derived from the properties. The re-architect has to decide which properties to use. In case a relatively high number of properties are used, rules on how to identify the most relevant properties should be defined. This was not necessary for the case study but would be necessary for larger recovery processes.

For the case studies, the recovery process was done manually and was supported by systems experts. The reference architecture was recovered from three family members. For this rather small number of members the process was manageable as it is defined in SARM. However, for a larger number of families or more complex structures tool support and modification of the methods are necessary. Necessary modifications are for example the definition of rules for handling the 'partial match' problem in the analysis process. In addition to the recovery process, also tools must support the maintenance process of large family systems. Components may for example belong to more than one software architectures and the reference architecture may be adapted to new requirements.

6.3.9.2 Evaluation of ASDL

ASDL provides concepts for a comprehensive architectural description of the architecture of a single system and of the reference architecture of a family of systems. A mayor advantage of ASDL is its flexibility. The property base may be expanded to represent most properties of a family of systems. This solves however not the general problem, that it is difficult to represent and manage architectural properties. Although ASDL was used on an embedded real-time system, it should be adapted to other domains by adapting the property base.

Since an ADL supports not only the software architecture recovery process but all forward engineering process, ADL's must be integrated into the company-specific software development process. No experiences were made on that topic, however it can be expected, that adaptations to the software development process and ASDL will be necessary.

The ASDL-description of the case studies were documented in a word processing tool and therefore changing of numbers or names required much effort because of change propagation. Working in hyper-text documents could simplify the definition and maintenance activities. There is also no tool available for ASDL that could support the editing process and could perform simple checks.

6.3.9.3 Resume

SARM and ASDL were applied to the case study presented in this dissertation. Systems and process experts of the case studies evaluated both techniques. The feedback was positive and the results were satisfying. SARM was an effective method for the recovery of the case study systems. With minor adoptions SARM should produce satisfying results for other system domains and organizations. ASDL was a language that provided constructs to represent the architecture in a complete and consistent way. The integration of ASDL in the software engineering process of an organization is a challenging task as it is for any ADL. The positive effects of representing architectural information in a structured way makes it worthwhile to integrate the language in the process.

Recovery of architectural information is a promising approach for gaining control over the evolution of existing software systems. The usage of architectural descriptions will result in better handling of early design decisions, increased communication between system shareholders, transferable system abstractions and increased application maintainability. Although, as Brooks identified already in 1982, "the tar pit of software engineering will continue to be sticky for a long period of time." [Bro82] p. 177

7. Appendix

7.1 Abbreviations

ABB	Asea Brown Boveri
ARES	Architectural Reasoning for Embedded Systems
ASDL	Architecture Structure Description Language
AST	Abstract Syntax Tree
CORBA	Common Object Request Broker Architecture
CSP	Communicating Sequential Processes
ESPRIT	European Strategic Program for Research in Information Technology
FAMOOS	Fram ework-based Approach for Mastering Object-Oriented Software Evolution
HW	Hardware
IDL	Interface Description Language
MPI	Message Passing Interface
OMA	Object Management
OMG	Object Management Group
OMT	Object Modeling Technique
ORB	Object Request Broker
ROOM	Real-Time Object-Oriented Modeling language
SARM	Software Architecture Recovery Methodology
SDL	Specification and Description Language
STL	Standard Template Library
SW	Software

7.2 ASDL

This chapter contains the ASDL language definition. ASDL constructs are generally described, followed by the graphical representation, the concrete textual grammar and the semantic definition.

The concrete textual grammar follows Backus-Naur Form (BNF) semantics. In the following, a convention is used that ASDL keywords are typeset in **boldface** and non-terminals are typeset in *italic* . In the grammar rules square brackets indicate 0 or 1 instances, curly braces indicate 1 or more instances and a vertical bar separates alternative choices.

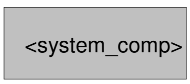
7.2.1 System

7.2.1.1 Description

The purpose of the ASDL system construct is to allow system architects to define the building block of the conceptual architecture of a system. The resulting description of the system is a hierarchically structured collection of architectural elements (components, interfaces, connectors).

The system component (root component) is representing the system to its environment. The root component is ‘parent’ to at least one component.

7.2.1.2 Graphical representation



7.2.1.3 Concrete textual grammar

<system > ::= **system_begin** < system_name **>;** <system_desc **>;** <system_ratio **>;** <system_component **>;** <system_constraint>; { <component> | <connector>}; <configuration> **system_end** <system_name>;
<system_name> ::= **system_name** **-** <text>
<system_desc> ::= **system_desc** **-** <text>
<system_ratio> ::= **system_ratio** **-** <text>
<system_component> ::= **system_comp** <c_name>
<system_constraint> ::= **system_const** <text>

7.2.1.3.1 Description

<system_desc> - general description of the system in plain text
<system_ratio> - description of system rationale (justification for the particular choices of system structure and style) in plain text
<system_component> - identifier of the component that is the root component of the system
<system_constraint> - description of the system constraints (e.g. semantic constraints that enforce safety requirements)

7.2.1.4 Semantic

For the <system_component> a corresponding <component> must exist.

The number of <connector> must be at least the number of <component> (c) minus one (c-1). The <system_component> is excluded.

7.2.2 Component

7.2.2.1 Description

ASDL distinguishes between component types and component instances. Component types in ASDL are described by a number of properties, e.g. safety-critical aspects. The description of a component type also includes the description of the component interfaces. As in Darwin, the names of the interfaces are local to the component type specification. ASDL components are therefore independent constructs and may be specified and tested as stand-alone constructs. The instantiation of a component type is described in section 7.2.5 .

If the commonality analysis described in this work should be performed, all component properties are mandatory because they are input to the commonality analysis. The parent - child relationship is non-associative (to avoid loops, this is also valid for grandparents etc.).

7.2.2.2 Graphical representation

7.2.2.3 Concrete textual grammar

<component_type> ::= **comp_begin** <c_name> **;** <c_desc> **;** <c_group> **;** <c_struc> **;** <c_type> **;** <c_safety> {<interface>} **comp_end** <c_name>;
<c_name> ::= **comp_name -** <text>
<c_desc> ::= **comp_desc -** <text>
<c_group **>** ::= **comp_group -** <c_group_enum **>**
<c_group_enum **>** ::= **comp_A | comp_B | comp_sync | comp_voter | n_a**
<c_struc> ::= **comp_struc -** <c_struc_enum>
<c_struc_enum> ::= **primitive_comp | composite_comp**
<c_type> ::= **comp_type** - <c_type_enum>
<c_type_enum> ::= **data_processing | data_store | communication_control | syncroniser | voter | monitor | run_time_system | user_interface | n_a**
<c_safety> ::= **safety_critical -** <c_safety_enum>
<c_safety_enum> ::= **Yes | No**

Description:

<c_name> - unique description of the component type
<c_desc> - general description of the component in plain text
<c_group **>** - specific description of a safety-critical component (see section *7.2.2.5 Safety-critical component / safety block*)
<c_struc> - description of the structural role of a component (either primitive or composite component)
<c_type> - description of the main characteristic of the component (e.g. data_store)
<c_safety> - definition if a component is a safety-critical component or not (n.a. is used in case of a composite component which consists of safety-critical and non-safety-critical components)

7.2.2.4 Semantic

The <comp_name> must be unique.

The following relations are allowed between <c_group_enum> and <c_safety_enum>:

c_group_enum	**c_safety_enum** yes	no
comp_A	x	
comp_B	x	
comp_sync	x	
comp_voter	x	
n_a	x	x

The following relations are allowed between <c_type_enum> and <c_group_enum>:

c_type_enum	**c_group_enum** comp_A	comp_B	comp_ sync	comp_ voter
data_processing	x	x		
data_store	x	x		
communication_control	x	x		
syncroniser			x	
Voter				x
Monitor	x	x		
run_time_system	x	x		
user_interface				
n_a	x	x		

<interface> must consist of a least one component interface (cross element restriction)

7.2.2.5 Safety-critical component / safety block

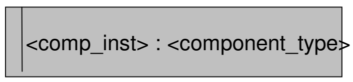
Safety-critical components collected in a safety block are a specific pattern that is often used in real-time systems and is therefore introduced in this chapter about ASDL components. This pattern is represented by a safety block.

7.2.2.5.1 Description

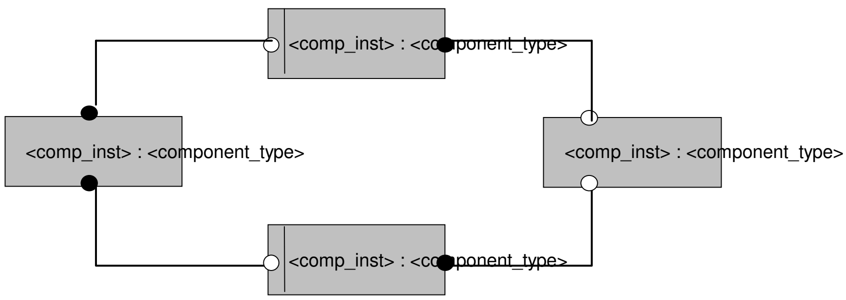
A safety block implements n-version programming and consists of a number of safety-critical components, a syncroniser and a voter. The safety-critical components are developed by different teams. The syncroniser synchronizes the input with the safety-critical components. At run-time the safety-critical components are executed (semi)parallel and the results are compared by a software voter. If all critical components deliver the same results, the execution continues. Otherwise some form of an exception-handling procedure will be executed, bringing the system into a safe state.

7.2.2.5.2 Graphical representation

Safety-critical components are specifically marked by a vertical line.



A safety block can be built by combining safety-critical components with non-safety-critical components (voter, syncroniser).



7.2.2.5.3 Concrete textual grammar

The grammar of a safety-critical component is the same as the grammar of a component, except that the safety attribute is default **Yes**.

<component_type> ::= **comp_begin** <c_name> **;** <c_desc> **;** <c_group> **;** <c_struc> **;** <c_type> **; safety_critical Yes** {<interface>} **comp_end** <c_name>;
<c_name> ::= **comp_name** **-** <text>
<c_desc> ::= **comp_desc** **-** <text>
<c_group **>** ::= **comp_group -** **<**c_group_enum **>**
<c_group_enum **>** ::= **comp_A** | **comp_B** | **comp_sync** | **comp_voter** | **n_a**
<c_struc> ::= **comp_struc** **-** <c_struc_enum>
<c_struc_enum> ::= **primitive_comp** | **composite_comp**

<c_type> ::= **comp_type** - <c_type_enum>
<c_type_enum> ::= **data_processing** | **data_store** | **communication_control** | **syncroniser** | **voter** | **monitor** | **run_time_system** | **user_interface** | **n_a**

7.2.2.5.4 Semantic

Semantic rules for safety-critical components:

- All 'childs' of a safety-critical component must also be safety-critical components.
- Safety-critical components may only be connected with other safety-critical components or a component with *comp_type syncroniser* or *comp_type voter*

Specific semantic rules apply for a safety block:

- consists of one component with comp_type syncroniser
- consists of one component with comp_type voter
- consists of at least two components with the property safety_critical yes
- input from external components (not safety block components) is restricted to the syncroniser component
- output to external components (not safety block components) is only allowed from the voter component
- the information flow is from the *syncroniser* component to the *safety-critical* components and further on to the *voter* component (safety path)
- no communication is allowed between the *safety-critical* components that belong to different safety paths

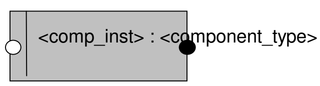
7.2.3 Interface

7.2.3.1 Description

Interfaces provide and/or require services to/from the environment.

7.2.3.2 Graphical representation

The diagrammatic convention is that circles not filled-in represent interfaces (services) required by a component and filled-in circles represent interfaces (services) provided by a component.



In the sub-system view of a component the interfaces are filled with triangles (required/provided):

7.2.3.3 Concrete textual grammar

<interface> ::= <i_name>; <i_desc>; <i_role>; <i_role_desc>
<i_name> ::= **intf_name** - <text>
<i_desc> ::= **intf_desc** -<text>
<i_role> ::= **intf_role -** <i_role_enum>
<i_role_enum> ::= **client** <text> | **server** <text> | **producer** <text> | **consumer** <text>

Description:

<i_name> - unique description of the interface
<i_desc> - general description of the interface in plain text
<i_role> - description of the role an interfaces plays (either client, server, producer or consumer)

7.2.3.4 Semantic

Following associations are allowed between <i_role_enum> and <c_type_enum>:

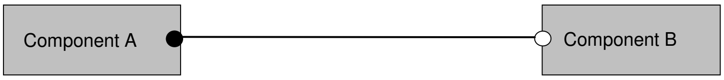
	interface_role			
component_role	**client**	**server**	**producer**	**consumer**
data_processing	x	x	x	x
data_store	x		x	x
communication_control	x	x		
voter	x	x	x	x
monitor	x	x	x	x
run_time_system	x	x	x	x
user_interface	x	x		x
n_a	x	x	x	x

7.2.4 Connector

7.2.4.1 Description

Connectors connect two components via the component interfaces. Connector properties include the synchronization information, communication information, control flow information and timing requirements.

7.2.4.2 Graphical representation



7.2.4.3 Concrete textual grammar

<connector > ::= <cr_name>; <cr_desc>; <cr_type>; <connecting_intf>;
<communication_type>;<cr_transfer_pattern >; <cr_infex>
<cr_name> ::= **conn_name** - <text>
<cr_desc> ::= **conn_desc** - <text>
<connecting_intf> ::= **conn_intf - (** <i_name> **,** <i_name> **)**
<communication_type> ::= <data_flow_dir>; <control_flow_dir>;
<data_flow_dir> ::= **data_flow** - <flow_enum>
<control_flow_dir> ::= **control_flow** - <flow_enum>
< flow_enum> ::= **unidirectional(intf** <i_name> **to intf** <i_name>) |
bi-directional | **n_a**
<cr_transfer_pattern > ::= **transfer** - <cr_transfer_enum>
<cr_transfer_enum> ::= **asyn** | **syn**
<cr_infex> ::= **info_exch -** <cr_infex_enum>
<cr_infex_enum> ::= **periodic <** cr_time **>|** **irregular** | **bounded <** sep_time **>** | **bursty** | **unbounded**
<cr_time> ::= **<**number **>** **<**time unit **>|** **n_a**
<sep_time> ::= **<**time unit **>**
<time_unit> ::= **ms** | **µs**

Description:

<cr_name> - unique description of the connector
<cr_desc> - general description of the connector in plain text
<connecting_intf> - declares the two interfaces that are connected by the connector
<communication_type> - defines a number of predefined transfer patterns between interfaces (either synchronous or asynchronous)
<cr_infex> - declares the type of information exchange(see also [AGP98]):

- Periodic - the arrival of communication is separated by equal timing intervals
- Irregular - the arrival of communication separated by unequal, but known, intervals
- Bounded - communication cannot occur before the elapse of a minimum separation time after the end of a previous event

- Bursty - communication can occur arbitrarily close to the previous one; the number of communication units over a specified interval is bounded
- Unbounded - communication can occur at any time, with no restrictions or bounds

7.2.4.4 Semantic

The connector name must be unique.

The connecting interfaces must be existing valid interfaces of two different components.

Valid interface connections (Component A provides a service which is required by component B):

	Component B			
Component A	client	server	producer	consumer
Client		x		
Server	x			
Producer				x
Consumer			x	

The number of connectors (c) in the system is at least the number of interfaces (I) divided by 2. Interfaces of the system component are excluded.

7.2.5 Configuration

7.2.5.1 Description

The configuration describes the instances of components, the child_of relation between components, connectors and how connectors connect the component interfaces. The configuration assembles architectural elements to form the specific architecture.

7.2.5.2 Graphical representation

The graphical representation of components, interfaces and connectors represents the configuration of an architecture.

7.2.5.3 Concrete Textual Grammar

<configuration> ::= **config_begin** <config_name> {<comp_inst> | <conn_inst> | <child_of_inst> } **config_end** <config_name>

<config_name> ::= **config_name** - <text>
<comp_inst> ::= **comp_inst** - <comp_id_text>
<child_of_inst> ::= **child_desc** <child_of>
<child_of> ::= **child_of_inst** <c_name> **|** **<**system_name **>**
<conn_inst> ::= **conn_inst** <conn_name> **intf** <comp_intf> **;** <comp_intf>
<comp_intf> ::= <comp_name> **:** <intf_name>

Description:

<config_name> - unique description of the configuration
<child_of_inst> - identifier of the composite component which is the parent of the component under consideration

7.2.5.4 Semantic

The components, interfaces and connectors must be existing valid elements.

The <child_of> (parent component) must be a defined valid composite component.

7.3 References

[AAG95] Abowd G., Allen R., and Garlan D., "Formalizing Style to Understand Descriptions of Software Architecture," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 4, no. 4, October 1995, pp. 319-364.

[AF97] Allen P. and Frost S., "Leaping the Obstacles of Component Technology," Object Magazine, vol. 7, no. 8, October, 1997.

[ADG98] Allen R., Douence R., and Garlan D., "Specifying and Analysing Dynamic Software Architectures," Proceedings of 1998 Conference on Fundamental Approaches to Software Engineering, Lisbon, Portugal, March, 1998.

[AG92] Allen R. and Garlan D., "Towards Formalized Software Architectures", CMU-CS-92-163, July, 1992.

[AG94] Allen R. and Garlan D., "Formalizing Architectural Connection," Proceedings of the 16th International Conference on Software Engineering, Sorento, Italy, May 1994.

[AGI98] Allen R., Garlan D., and Ivers J., "Formal Modeling and Analysis of the HLA Component Integration Standard," In proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6), November 1998.

[All97] Allen R.J., "A Formal Approach to Software Architecture" Ph.D. Dissertation. Carnegie Mellon University, May, 1997.

[AGP98] Alonso A, García-Valls M., and de la Puente J.A., "Assessment of timing properties of family products," Proceedings of the Second International Workshop on Development and Evolution of Software Architectures for Product Families. Las Palmas de Gran Canaria, Spain, 1998.

[Arm93] Armitage J.W., Process Guide for the Domain-Specific Software Architectures (DSSA) Process Life Cycle, CMU/SEI-93-SR-21, 1993.

[BCC+97] Bass L., Clements P., Cohen S., Northrop L., and Withey J., Product Line Practice Workshop Report, CMU/SEI-97-TR-003, 1997.

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

[BG97] Bellay B. and Gall H., "A Comparison of Four Reverse Engineering Tools," Proceedings of Fourth Working Conference on Reverse Engineering, October 6-8,1997.

[BG98] Bellay B., Gall H. “An Evaluation of Reverse Engineering Tool Capabilities. “ Journal of Software Maintenance: Research and Practice, 10(5):305-332. John Wiley & Sons, Ltd., September-October 1998

[BM97] Baxter I. and Mehlich M., "Reverse Engineering is Reverse Forward Engineering," Proceedings of the Forth Working Conference on Reverse Engineering. Amsterdam, The Nederland’s, October 6-8, 1997.

[Bel97] Bellay B.J., Evalu ation of Reverse Engineering Tools for Architecture Recovery, Master's Thesis, Technical University of Vienna, Information Systems Institute, Distributed Systems Department, 1997.

[Bha93] Bhansali S., Architecture-driven Reuse of Code in KASE. SEKE'93, Sa n Francisco, 1993.

[Big89] Biggerstaff T.J., "Design Recovery for Maintenance and Reuse," IEEE Computer, vol. 22, no. 7, pp. 36-49, 1989.

[BMW94] Biggerstaff T.J., Mitbander B.G., and Webster D.E., "Program Understanding and the Concept Assignment Problem," Communications of the ACM, vol. 37, no. 5, pp. 72 – 83, May 1994.

[BS92] Boehm B. and Scherlis W., "Megaprogramming," in Proceedings of the DARPA Software Technology Conference, Los Angeles, California, pp. 63 - 82, 1992.

[BH98] Bowman I.T. and Holt R.C., "Software Architecture Recovery Using Conway's Law," Procedings of CASCON'98, November 30-December 3, 1998.

[BH93] Braek R. and Haugen O., *Engineering Real Time Systems*. Prentice Hall International, 1993.

[BW96] Brown A.W. and Wallnau K.C., "A Fram ework for Systematic Evaluation of Software Technologies," Proceedings of the 2nd IEEE International Conference on Engineering of Complex Systems, 1996.

[BH97] Buchheit M. and Hollunder B., "Building and Assembling Components," Object Magazine, vol. 7, no. 9, November, pp. 62-64, 1997.

[BMR+96] Buschmann F., Meunier R., Rohnert H., Sommerlad P., and Stal M., Pattern-Oriented Software Architecture. Chichester: John Wiley & Sons, 1996.

[Bro82] Brooks F. P., *The Mythical Man Month*. Addison-Wesley, reprinted with permission, 1982.

[Cat98] Cattaneo F., "ADLs," Dipartimento di Elettronica e Informazione Politecnico di Milano, http://www.elet.polimi.it/~fuggetta/WebBook/adls.htm, 1998.

[CH98] Chase M.P. and Harries D.R., "Software Architecture in Reverse," http://www.mitre.org/pubs/edge/january_98/first.htm, The MITRE Advanced Technology Newsletter, January, 1998.

[CHR+96] Chase M.P., Harris D.R., Roberts S.N., and Yeh A.S., "Analysis and Presentation of Recovered Software Architectures," Proceedings of the Third Working Conference on Reverse Engineering, pp. 153-162, November 8-10, 1996.

[Cle96a] Clements P.C., "A Survey of Architecture Description Languages," Eighth International Workshop on Software Specification and Design. Paderborn, Germany, March 1996.

[Cle96] Clements P.C., Coming attractions in Software Architecture, CMU/SEI-96-TR-008, January 1996.

[CN96] Clements P.C. and Northrop L.M., Software Architecture: An Executive Overview, CMU/SEI-96-TR-003, 1996.

[Cok96] Cockburn A., The interaction of social issues and software architecture, Communications of the ACM Vol. 39, No. 10 (Oct. 1996), Pages 40-46, 1996.

[Col92] Collins COBUILD English Language Dictionary. Stuttgard: Klett 1987, reprinted 1992.

[DC95] Dean T.R. and Cordy J.R., "A Syntactic Theory of Software Architecture," IEEE Transactions on Software Engineering, vol. 21, no. 4, pp. 302-313, April/1995.

[DG98] DeBaud J.M. and Girard J., "The Relation Between the Product Line Development Entry Points and Reengineering," in Proceedings of the 2nd International Workshop on Development and Evolution of Software Architectures for Product Families, Springer, Berlin Heidelberg, pp. 132-139., 1998.

[DKO+97] Dikel D., Kane D., Ornburne S., Loftus W., and Wilson J., "Applying Software Product-Line Architecture," IEEE Computer, vol. 30, no. 8, August 1997.

[Dou98] Douglass B.P., Real-Time UML. Object Technology Series, Addison Wesley, 1998.

[EM99] Egyed A. and Medvidovic N., "Extending Architectual Representation in UML with View Integration," Proceedings of the Second International Conference on the Unified Modeling Language (UML'99), Fort Collins, CO, October 1999, pp. 2-16.

[EG99] Egyed A. and Gacek C., "Automatically Detecting Mismatches during Component-Based and Model-Based Development," Proceedings of 14 th International Conference on Automated Software Engineering, Cocoa Beach, FL, 1999.

[EG98] Eixelsberger W. and Gall H., "Describing Software Architectures by System Structure and Properties," in Proceedings of the twenty-second annual International Computer Software and Application Conference (CompSac98). Vienna, Austria, August, 1998.

[Eix98] Eixelsberger W., "Recovery of a Reference Architecture: A case study," Proceedings of the 3rd International Software Architecture Workshop (ISAW3), Orlando, USA, November 1998.

[EKO+98] Eixelsberger W., Kalan M., Ogris M., Beckman H., Bellay B., and Gall H., "Recovery of Architectural Structure: A Case Study," in Proceedings of the Second International Workshop on Development and Evolution of Software Architectures for Product Families. Las Palmas de Gran Canaria, Spain, 1998.

[EOG+98] Eixelsberger W., Ogris M., Gall H., and Bellay B., "Software Architecture Recovery of a Program Family," Proceedings of International Conference on Software Engineering (ICSE '98), Kyoto, Japan, April 1998.

[EWK+96] Eixelsberger W., Warholm L., Klösch R., Gall H., and Bellay B., "A framework for software architecture recovery," International Workshop on Development and Evolution of Software Architectures for Product Families in Las Navas del Marques, Avila, Spain, November 1996.

[EWK+97] Eixelsberger W., Warholm L., Klösch R., and Gall H., "A Framework for Software Architecture," Proceedings of International Conference on Software Engineering (ICSE '97), Boston, USA, ACM Press, May 1997.

[ERP+96] Ellis W.J., Richard F.H., Poon P.T., Rayford D., Saunders T.F., Sherlund B., and Wade R.L., Toward a Recommended Practice for Architectural Description, Proceedings 2nd IEEE International Conference on Engineering of Complex Computer Systems, Montreal, Quebec, Canada, October 21-25, 1996.

[Fra96] Frank H., "View Integration für objektorientierte Datenbanken," Ph.D. Dissertation. University of Klagenfurt, October, 1996.

[FS97] Fowler M. with Scott K., UML Distilled - Applying the Standard Object Modeling Language. Menlo Park, California: Addison Wesley, 1997.

[GAC+95] Gacek C., Abd-Allah A., Clark B., and Boehm B., "On the definition of Software System Architecture," ICSE 17 Software Architecture WorkshopApril, 1995.

[GJM91] Ghezzi C., Jazayeri M., and Mandrioli D., *Fundamentals of Software Engineering* . Prentice-Hall, 1991.

[GKM95] Gall H., Klösch R., and Mittermeir R., "Object-Oriented Re-Architecturing," LNCS 989, Springer-Verlag, pp. 499-519, Fifth European Software Engineering Conference (ESEC '95), Barcelona, Spain, September, 1995.

[GHJ+95] Gamma E., Helm R., Johnson R., and Vlissides J., *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Gar95] Garlan D., "Introductory Remarks," Position Paper, Dagstuhl Seminar 9508. Schloss Dagstuhl, Germany, 20.-24.02.95, 1995.

[GMW97] Garlan D., Monroe R., and Wile D., "Acme: An Architecture Description Interchange Language," Proceedings of CASCON'97.

[GP95] Garlan D. and Perry D.E., "Intro duction to the Special Issue on Software Architecture," IEEE Transactions on Software Engineering, vol. 21, no. 4, April 1995.

[GAO94] Garlan D, Allen R., and Ockerbloom J., "Exploiting Style in Architectural Design Environments," Proceedings of SIGSOFT'94, New Orleans, Louisiana, USA, December 1994, pp. 175-188.

[GAK99] Guo Y., Atlee J.M., and Kazman R., "A Software Architecture Reconstruction Method," Proc. First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX, USA, February 1999.

[GW99] Garlan D. and Wang Z., " Acme-Based Software Architecture Interchange," in *Paolo Ciancarini, Alexander L. Wolf (Eds.): Coordination Languages and Models, Lecture Notes in Computer Science, Vol. 1594, Springer,* 1999, pp. 340-354.

[HN90] Harand i M.T. and Ning J.Q., "Knowledge-Based Program Analysis," IEEE Software, vol. 7, no. 1, January 1990, pp. 74-81.

[HPL+95] Hayes-Roth B., Pfleger K., Lalanda P., Morignot P., and Balabanovic M., "A Domain-Specific Software Architecture for Adaptive Intelligent Systems," IEEE Transactions on Software Engineering, vol. 21, no. 4, April, 1995.

[Hon96] Honeywell, "What are the Benefits of Using a DSSA?" http://www.htc.honeywell.com/projects/dssa/dssa_benefits.html, 1996.

[IW95] Inverardi P. and Wolf A.L., "Formal Specification and Analysis of System Architecture Using the Chemical Abstract Machine Model," IEEE Transactions on Software Engineering, vol. 21, no. 4, April/1995, pp. 373-386.

[JCJ+94] Jacobson I., Christerson M., Jonsson P., and Oevergaard G., Object-Oriented Software Engineering. ACM Press, Reprinted 1994.

[JGJ97] Jacobson I., Griss M., and Jonsson P., "Making the Reuse Business Work," IEEE Computer, vol. 30, no. 10, October, pp. 36-42, 1997.

[Jos96] Joseph M., Real-time Systems Specification, Verification and Analysis. Hertsfordshire: Prentice Hall International (UK), 1996.

[JRL00] Jazayeri M., Ran A., and van der Linden F. (Eds.), *Software Architecture for Product Families: Principles and Practice*. Addison Wesley, 2000.

[KC99] Kazman R. and Carrière S.J., "Playing Detective: Reconstructing Software Architecture from Available Evidence," Journal of Automated Software Engineering, 6(2) (April 1999), p.107-138.

[KC97] Kazman R. and Carrière S.J., Playing Detective: Reconstructing Software Architecture From Available Evidence, CMU/SEI-97-TR-010. Software Engineering Institute, 1997.

[Kie98] Kiely D., "Are Components the Future of Software," IEEE Computer, vol. 31, no. 2, February, pp. 10-11, 1998.

[KC94] Kogut P. and Clements P., "The Software Architecture Renaissance," Crosstalk - The Journal of Defense Software Engineering, vol. 7, no. 11, November / 1994.

[KC95] Kogut P. and Clements P., "Feature Analysis of Architecture Description Languages," In Proceedings of the Software Technology Conference (STC'95), Salt Lake City, April 1995.

[Kru95] Kruchten P. B., "The 4+1 View Model of Architecture," IEEE Transactions on Software Engineering, vol. 12, no. 6, November, 1995, pp. 42-50.

[Luc96] Luckham D.C., "Rapide: A Language and Toolset for Simu lation of Distributed Systems by Partial Orderings of Events," DIMACS Partial Order Methods Workshop IV, Princeton University, July 1996.

[LKA+95] Luckham D.C., Kennay J.J., Augustin L.M., Vera J., Bryan D., and Mann W., "Specification and Analysis of System Architecture Using Rapide," IEEE Transactions on Software Engineering, vol. 21, no. 4, April, pp. 336-355, 1995.

[LV95] Luckham D.C. and Vera J., "An Event-Based Architecture Definition Language," IEEE Transactions on Software Engineering, vol. 21, no. 9, September/1995.

[Lyo98] Lyons A., "UML for Real-Time Overview." http://www.objectime.com/otl/technical, April 1998.

[MDE+95] Magee J., Dulay N., and Eisenbach S. and Kramer J., "Specifying Distributed Software Architectures," Proceedings of 5th European Software Engineering Conference (ESEC 95), Sitges, Spain, September 1995.

[MDK94] Magee J., Dulay N., and Kramer J., "Regis: A Constructive Development Environment for Distributed Programs," IEE/IOP/BCS Distributed Systems Engineering, vol. 1, no. 5, September 1994, pp. 304-312.

[Med96] Medvidovic N., A Classification and Comparison Framework for Software Architecture Description Languages, Technical Report UCI-ICS-97-02. University of California, Irvine, 1996.

[Mit00] Mittermeir R.T., "Comprehendin g by varying Focal Distance" Proceedings of 8th International Workshop on Program Comprehension, Limerick, Ireland, June 10-11, 2000.

[MOR+96] Medvidovic N., Oreizy P., Robbins J.E., and Taylor R.N., "Using Object-Oriented Typing to Support Architectural Design in the C2 Style," Proceedings of SIGSOFT'96: The Fourth Symposium on the Foundations of Software Engineering (FSE4), San Fransisco, CA, October 18-18, 1996.

[MT00] Medvidovic N. and Taylor R.N., "A Classification and Comparison Framework for Software Architecture Description Languages," IEEE Transactions on Software Engineering, vol. 26, no. 1, pp. 70-93, January 2000

[MT97] Medvidovic N and Taylor R.N., "A Framework for Classifying and Comparing Architecture Description Languages," Proceedings of the Sixth European Software Engineering Conference (ESEC 97). Zurich, Switzerland, September 22-25, 1997, pp. 60-76.

[MK96] Mendonça N. and Kramer J., "Requirements for an Effective Architecture Recovery Framework," Proceedings of the 2nd ACM SIGSOFT International Software Architecture Workshop (ISAW-2), pp 101-105. San Francisco, October 1996.

[MK98] Mendonça N. and Kramer J., "An Experiment in Distributed Software Architecture Recovery," in Proceedings of the Second International Workshop on Development and Evolution of Software Architectures for Product Families. Las Palmas de Gran Canaria, Spain, 1998.

[MQ94] Moriconi M. and Qian X., "Correctness and Composition of Software Architectures," Proceedings of ACM SIGSOFT'94: Symposium on Foundations of Software Engineering, New Orleans, Louisiana, December,1994, pp. pp. 164-174.

[NKM96] Ng K., Kramer J., and Magee J., "A CASE Tool for Software Architecture Design," Journal of Automated Software Engineering, vol. 3, no. 3/4, Kluwer Academic Publishers, pp. 261-284, August, 1996.

[ORT98] Oreizy P., Rosenblum D.S., and Taylor R.N., On the Role of Connectors in Modeling and Implementing Software Architectures, Department of Information and Computer Science, University of California, Irvine, Technical Report UCI-ICS-98-04, Feb. 1998.

[Pac98] Peter S. Pacheco "A User's Guide to MPI" Department of Mathematics, University of San Francisico,http://www-unix.mcs.anl.gov/mpi/tutorial/ 1998

[Par76] Parnas D.L., "On the Design and Development of Program Families," IEEE Transactions on Software Engineering (TSE), vol. 2, no. 1, March 1976.

[PW92] Perry D.E. and Wolf A.L., "Foundations for the Study of Software Architecture," ACM SIGSOFT Software Engineering Notes, vol. 17, no. 4, Oct/1992.

[Pre95] Pree W., Design Pat terns for Object-Oriented Software Development. ACM Press; Addison-Wesley, 1995.

[Pre94] Pressman R.S., Software Engineering A Practitioner's Approach. Third Edition - European Adaption: McGRAW-HILL, 1994.

[Qui95] Quilici A., "Reverse Engineering of Lega cy Systems: A Path Toward Success," In Proceedings of the 17th Annual International Conference on Software Engineering. IEEE Press. Seattle, WA, pp. 331-336, April, 1995.

[RS96] Rajlich V. and Silva J.H., "Evolution and Reuse of Orthogonal Architecture," IEEE Transactions on Software Engineering, vol. 22, no. 2, February/1996, pp. 153 - 157.

[Ran96] Random House Webster's Dictionary. Ballantine Books, 1996.

[RTD97] Rapide Design Team, Guide to the Rapide 1.0 Language Reference Manuals, http://pavg.stanford.edu/rapide/rapide-pubs.html. Standford University, July 17, 1997.

[RO94] Ravene R. and Obenza R., Rate Monotonic Analysis for Real-Time Systems: Instructor's Guide, CMU/SEI-94-EM-11. Technical report - Carnigie Mellon Software Engineering Institute, April 1994.

[Rea90] Reasoning Systems, “Dialect – User’s Guide” Palo Alto, California, 1990

[SR98] Selic B. and Rumbaugh J., Using UML for Modeling Complex Real-Time Systems. http://www.objectime.com/, March 11, 1998.

[SDK+95] Shaw M., DeLine R., Klein D.V., Ross T.L., Young D.M., and Zelesnik G., "Abstractions for Software Architectures and Tools to Support Them," IEEE Transactions on Software Engineering, vol. 21, no. 4, April, 1995, pp. 314-335.

[SG95] Shaw M. and Garlan D., " Formulations and Forma lisms in Software Architecture," in Computer Science Today: Recent Trends and Developments. Lecture Notes in Computer Science, Vol. 1000, Jan van Leeuwen, Ed.1995, pp. 307-323.

[SG96] Shaw M. and Garlan D., Software architecture: perspectives on an emerging discipline. Upper Saddle River, New Jersey: Prentice Hall, 1996.

[SG94] Shaw M. and Garlan D., Characteristics of Higher-level Languages for Software Architectures, CMU-CS-94-210, December 1994.

[Smi97] Smith K.W., "Real-Time Object-Oriented Modeling, " Dr. Dobb's Journal #280, December 1997.

[Sta98] Stal M., "Worldwide CORBA: Distributed Objects and the Net," Object Magazine, vol. 8, no. 1, March 1998, pp. 34-43, 1998.

[SWM97] Storey M.A.D., Wong K., and Mueller H.A., "How do Program Understanding Tools Affect How Programmers Understand Programs," Proceedings of the Forth Working Conference on Reverse Engineering. Amsterdam, The Nederlands, October 6-8, 1997.

[TTC95] Taylor R.N., Tracz W., and Coglianese L., "Software Development Using Domain-Specific Software Architectures," ACM SIGSOFT Software Engineering Notes, vol. 20, no. 5, pp. 27-38, 1995.

[TDM97] Tichelaar S., Ducasse S., and Meijler T.D., "Architectural Extraction in Reverse Engineering by Prototyping: An Experiment," Proceedings ESEC/FSE'97 Workshop on Object-Oriented Reengineering, S. Demeyer and H. Gall (eds), August 1997.

[Tra97] Tracz W., "Domain-Specific Software Architecture," http://www.owego.com/dssa/faq/faq.html, 1995.

[Ves96] Vestal S., MetaH Programmer's Manual, Version 1.09. Honeywell Technology Center, April, 1996.

[Ves93] Vestal S., A Cursory Overview and Comparison of Four Architecture Description Languages. Honeywell Technology Center, Minneapolis, February 1993.

[Vli95] Vlissides J., "Reverse Architecture," Position Paper, Dagstuhl Seminar 9508. Schloss Dagstuhl, Germany, 20.-24.02.95, 1995.

[Was96] Wasserman A.I., "Toward a Discipline of Software Engineering," IEEE Software, November 1996.

[Wei98] Weiss D., "Commonality Analysis: A Systematic Process for Defining Families," in Proceedings of the 2nd International Workshop on Development and Evolution of Software Architectures for Product Families, Springer, Berlin Heidelberg, pp. 214-222., 1998.

[WRM95] Whitehead Jr E.J., Robbins J.E., Medvidovic N., and Taylor R.N., "Software Architecture: Foundation of a Software Component Marketplace," Proceedings of the ICSE17 Workshop on Architectures for Software Systems, 24-25 April, 1995.

[Wil99] Wile D., "AML: An Architecture Meta-Language," 14th International Conference on Automated Software Engineering (ASE'99), Cocoa Beach, FL, October 1999.

[WCK99] Woods S.G, Carrière S.J., and Kazman R., " The Perils and Joys of Reconstructing Architectures," The Architect . Volume 2 . Issue 3 . September 1999.

[YHC07] Yeh A. S., Harris D. R., and Chase M. P., "Manipulating Recovered Software Architeture Views," Proceedings 19th International Conference on Software Engineering, 1997.

[Zho98] Zhao J., "Applying Slicing Technique to Software Architectures," Proc. Fourth IEEE International Conference on Engineering of Complex Computer Systems, pp.87-98, August 1998.

[Zho99] Zhao J., "Extracting Reusable Software Architectures: A Slicing-Based Approach," Proc. ESEC/FSE'99 Workshop on Object-Oriented Reengineering, Toulouse, France, September 1999.