Linköping Studies in Science and Technology

Dissertation No. 867

AN APPROACH TO SYSTEMS ENGINEERING TOOL DATA REPRESENTATION AND EXCHANGE

by

Erik Herzog

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköping 2004

Abstract

Over the last decades computer based tools have been introduced to facilitate systems engineering processes. There are computer based tools for assisting engineers in virtually every aspect of the systems engineering process from requirement elicitation and analysis, over functional analysis, synthesis, implementation and verification. It is not uncommon for a tool to provide many services covering more than one aspect of systems engineering. There exist numerous situations where information exchanges across tool boundaries are valuable, e.g., exchange of specifications between organisations using heterogeneous tool sets, exchange of specifications from legacy to modern tools, exchange of specifications to tools that provide more advanced modelling or analysis capabilities than the originating tool or storage of specification data in a neutral format such that multiple tools can operate on the data.

The focus in this thesis is on the analysis, design and implementation of a method and tool neutral information model for enabling systems engineering tool data exchange. The information model includes support for representation of requirements, system functional architecture and physical architecture, and verification and validation data. There is also support for definition of multiple system viewpoints, representation of system architecture, traceability information and version and configuration management. The applicability of the information model for data exchange has been validated through implementation of tool interfaces to COTS and proprietary systems engineering tools, and exchange of real specifications in different scenarios. The results obtained from the validation activities indicate that systems engineering tool data exchange may decrease the time spent for exchanging specifications between partners developing complex systems and that the information model approach described in the thesis is a compelling alternative to tool specific interfaces.

Foreword and Acknowledgements

A lot of time has been allocated to the completion of this thesis. Still I have come to the realisation that it will never be quite complete. There will always be some aspect of the text that will be in urgent need for improvement, or some detail of the work which is not adequately presented in the last detail. Hence the quote below by Winston Churchill is very much applicable to this thesis:

> "Writing a book is like an adventure. To begin with it is a toy and an amusement. Then it becomes a mistress, then it becomes a master, then it becomes a tyrant. The last phase is that just as you are about to be reconciled to your servitude, you kill the monster and fling him to the public."

In this case, "the public" is the people and organisations interested in tool data exchange between systems engineering tools and systems engineering information models.

This thesis is concerning tool integration in general and systems engineering tool data exchange through the use of a tool neutral information model in particular. The information model is presented in detail as one part objective with the thesis is that it shall document information model as the positive and negative experience generated by using it for tool data exchange, such that those experiences made can be profited on in future systems engineering data exchange standardisation projects.

The information model presented in the thesis has been developed in the European SEDRES and SEDRES2 projects and the NUTEK founded projects Cohsy and Sedex. The model has also been extensively discussed and reviewed within ISO 10303-233 working group and in INCOSE. Initially the information model was very limited in scope and also, as stated very clearly by one distinguished project reviewer, of limited quality. However, the scope was extended over each model version and become more suitable for systems engineering tool data exchange. This positive development would not have been possible without the colleagues and friends participating in the projects. I would like to thank the following people for their comments, support and criticism: Roland Almgren, Sylvain Barbeau, Carsten Düsing, Roland Eckert, Harry Frisch, Michael Giblin, Klaus Heimannsfeld, Manfred Inderst, Julian Johnson, Jean-Louis Loueillet, Federica Luise, Wolfgang Mansel, Benny Nilsson, David Oliver, Phil Spiby and Mike Ward.

At the department of Computer and Information Science at Linköping University my thanks goes to Professor Anders Törne for his patience in supervising my studies and for reading the material I produced over the years. Professor Simin Nadjm-Therani provided a lot of valuable motivation at times when the thesis appeared to be in a state where it was highly questionable whether it would ever be completed. My friends Dr. Paul Scerri and Dr. Asmus Pandikow provided a great help and guidance in the deed that they did complete their thesis projects in the stipulated time — an admirable achievement! Of course my thanks goes also to past and present members of the Real-Time Systems laboratory with whom I shared a lot of fun. Likewise my thanks goes to the management and staff at Syntell AB and SAAB Training Systems for their generosity in allowing me to complete this thesis despite a pressed project schedule.

Finally I would like to thank my family, Almut, Alexander, Leonard and Christoph who have supported the completion of this thesis all in their own ways. The extra days the family life has added to the completion of the thesis has been rewarded with extra insight both in the subject presented in this thesis and in those more important aspects of real life.

Summary of publications

The following papers reporting on aspects of the information model presented herein and lessons learned from using the information model for data exchange have been published:

- Erik Herzog and Anders Törne. Using STEP to Integrate Systems Engineering Design Tools - Experiences from the SEDRES Project. In *Produktmodeller 98*, pages 479–491, November 1998.
- Erik Herzog and Anders Törne. A Seed for a STEP Application Protocol for Systems Engineering. In *Proceedings of the 1999 IEEE Conference and Workshop on Engineering of Computer-Based Systems*, pages 174–180. IEEE Computer Society Press, 1999.
- Erik Herzog and Anders Törne. Towards a Standardised Systems Engineering Information Model. In Allen Fairbairn, Cass Jones, Christine Kowlaski, and Peter Robson, editors, *Proceeding sof the 9th annual international symposium of the international council of systems engineering*, volume 2, pages 909–916. INCOSE, INCOSE, 1999.
- Erik Herzog and Anders Törne. AP-233 Architecture. In *Proceedings of the 10th Annual International Symposium of the International Council on Systems Engineering*, pages 815–822. INCOSE, 2000.
- Erik Herzog and Anders Törne. Support for Exchange of Functional Behaviour Specifications in AP-233. In *Proceedings 7th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 351–358, 2000.
- Klaus Heimannsfeld, Erik Herzog, Carsten Dusing, and Julian Johnson. Beyond Tool Exchanges - the Current Status and Future Implications of the Emerging ISO Standard AP-233 for the Exchange of System Engineering Data. In *Proceeding the 2nd European Conference on Systems Engineering*, September 2000.
- Erik Herzog, Asmus Pandikow, and Anders Törne. Integrating Systems and Software Engineering Concepts in AP-233. In *Proceedings*

of the 10th Annual International Symposium of the International Council on Systems Engineering, pages 831–837. INCOSE, 2000.

- Julian Johnson, Erik Herzog, Sylvain Barbeau, and Michael Giblin. The Maturing Systems Engineering Data Exchange Standard and Your Role. In *Proceedings of the 10th Annual International Symposium of the International Council on Systems Engineering*, pages 823–830, 2000.
- Erik Herzog and Anders Törne. Information Modelling for System Specification Representation and Data Exchange. In Frances Titsworth, editor, *Proceedings of the 8th IEEE International Conference and Workshop on the Engineering of Computer-based Systems*, pages 136 – 143. IEEE Computer Society, 2001.
- Erik Herzog and Anders Törne. Investigating Risks in Systems Engineering Tool Data Exchange. In *Proceedings of the 11th annual International Symposium of the International Council on Systems Engineering*. INCOSE, 2001.
- Julian Johnson and Erik Herzog. The Data Standard AP-233: An Invigorator for Global Systems Engineering. In *Proceedings of the 11th International Symposium of the International Council on Systems Engineering*, 2001.
- Julian Johnson, Erik Herzog, and Michael Giblin. The Technical Data Coverage of the Emerging AP-233 STEP Standard and its use in Virtual Enterprises. In *proceedings of PDT Europe 2001*, pages 203 – 212, April 2001.
- Erik Herzog. A PS covering Functional Architecture Support in SEDRES Developed Drafts of AP-233. Accepted for publication at INCOSE 2004.

CONTENTS

PART I

Preliminaries

Chapter 1
Introduction

Over the last centuries the complexity of and expectations in terms of, e.g., quality and availability, on human made systems have increased enormously. At the same time the prime engineering tool — the brain — has evolved minimally or not at all. To cope with complexity the engineering community have formalised methods and processes for describing systems in formats that facilitate unambiguous communication. One important step in this process was the introduction of modern engineering drawing principles by Gaspard Monge in 1801 [51]. Consequently, engineering drawings became the accepted means for communication between the design and manufacturing phase in the development process.

As system complexity increase new process phases has been added early in the process. Today it is common to describe a system in terms of its life-cycles as outlined in Figure 1.1 [24]. The motivation for this is to promote consideration of issues like system updates and phase out in the early phases in the system development process. The objective is to develop life-cycle-balanced systems. Factors beyond production cost such as usability, upgradeability, maintainability, procedures for phase-out and

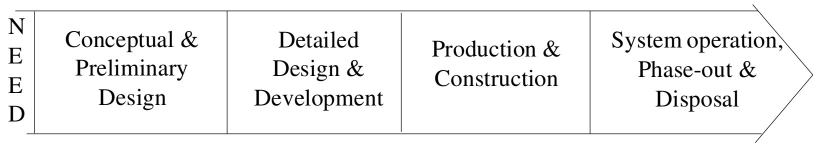

N
E
E
D
Conceptual & Preliminary Design
Detailed Design & Development
Production & Construction
System operation, Phase-out & Disposal

Figure 1.1: The system life-cycle

disposability are important in this perspective and should be addressed throughout the development process.

A number of process standards for the development of complex systems have been proposed, e.g., IEEE-1220 [67] and EIA-632 [102]. The standards define a number of activities that shall be undertaken to ensure that all aspects of the system life-cycle is considered for a system. The activities performed in a design or development life-cycle phase depend on the characteristics of the system, the development organisation and the phase in the cycle. At a very high level of abstraction all design oriented phases share the structure proposed by Patterson [119] and presented in Figure 1.2. Any number of requirements define the problem space for a system, i.e., required capabilities for a system and the constraints identified. The format and structure of requirements for a phase depend on, e.g., life-cycle, system complexity, process — it may be textual definitions, engineering drawing or formal specifications.

The first task in each generic phase is to recognise and understand the problem space. Once the problem space is understood analyses are performed to investigate alternate solutions. Candidate solutions are synthesized and evaluated against the original requirements. The selected solution, a specification, will likely serve as input to the next phase in the life-cycle and thus form part of the requirements for that phase. The process is inherently iterative and communication between the owner of the requirements can be expected to be intense. Analyses within a phase may

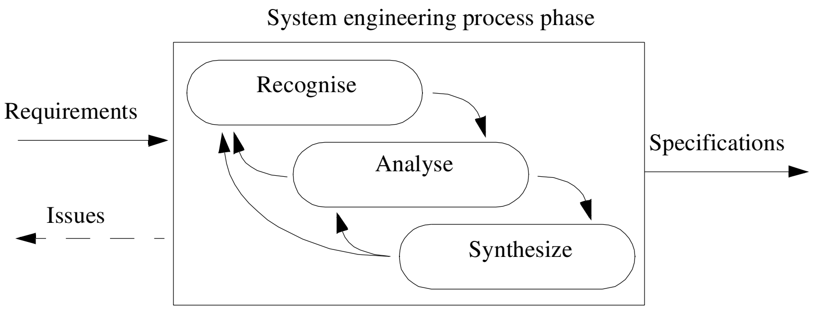


Figure 1.2: Generic systems engineering process phase

reveal that the requirements as stated are too strict. In such cases it is necessary to adjust requirements to match what is possible to realise under the constraints imposed. It can be expected that a specification produced in a phase is in a subset of the solution space defined by the requirements. The specification is usually more specific and detailed than the set of requirements that were guiding the work within a phase. It is not uncommon that the output of a phase is in the form of multiple specifications for a number of identified subsystems.

1.1 Product Data in the Systems Engineering Process

Large amount of design data is created, generated, referenced and maintained over the complete life-cycle for any complex system. As noted above, different engineering methods are used in the different phases in the development process. Early in the process, in the conceptual and preliminary design phases, the requirements captured are typically expressed at a high level of abstraction and usually do not prescribe any specific realisation technology. As the process proceeds engineers interpret the original requirements, and partition complex systems into more manageable components. Design and implementation decisions are made until component specifications reach a level of detail such that detailed engineering domain analyses can be performed and it is a suitable foundation for production. For mechanical parts this could be in the form of specifications including manufacturing drawings and NC programs. For computer hardware it could be a specification expressed in VHDL and for software it could be source code expressed in a high level programming language.

The information generated is not only restricted to the system parts to be manufactured. System integration, verification and validation plans and operational and maintenance specifications are vital for the engineering of complex systems. The process is not linear. Several specification versions and variants are usually considered in each phase. Moreover for traceability it is important that trade-off, design decision and change information is maintained throughout the system life-cycle.

1.1.1 ENGINEERING SUPPORT TOOLS

The factors described above make efficient management of product data a major challenge for industry despite the introduction of PDM (Product Data Management) systems and the development of ever more advanced computer based tools for engineering design, analysis and manufacturing support. Today there are a large number of excellent engineering tools available to support engineers in different domains and who are applicable to specific tasks encountered in the development process. Without going into details it can be assumed that the service offered by state of the art tools are more than adequate for the domain of service they are designed to cover. However, the situation is such that there are neither a single tool which can effectively support all design activities for a complex system, nor is the general situation such that a single tool has achieved total market domination. Consequently, it can be expected that development information for a complex systems is distributed over multiple engineering tools, or more generally databases, e.g., requirements, manufacturing, engineering analysis, and maintenance databases.

Complex systems are often developed in a multi-organisation context, either in partner relationships or in contractor - subcontractor relationships. It is common that the cooperating organisations use different sets of engineering support databases. The prevailing situation has led to a demand for mechanisms for enabling tool and database interoperability with the objective to support data exchange across tool boundaries and also for linking product data produced in different phases of the system life-cycle.

1.1.2 DATA REPRESENTATION AND EXCHANGE

Data representation and exchange problems are not new. A number of domain specific data exchange standards have been developed to improve database interoperability for different engineering domains, see [116] for an overview. The preferred approach in these standards is to define an information model that captures the design elements of interest for the domain of the standard and their logical interrelationships. One of the earliest standards was IGES [51][69] that was initiated in the early 1980'ies with support for geometrical CAD data. Later IGES was succeeded by the STEP (ISO 10303) standard framework [6]. Individual STEP standards (application protocols) provide data representation specifications for, e.g., Configuration Controlled Design 3D designs of mechanical parts and assemblies (AP-203) [7], for Design-analysis of Composite Structures (AP-209), Electronic Printed Circuit Assembly, Design and Manufacture (AP-210), Electrotechnical Design and Installation (AP-212) and Core data for automotive mechanical design process (AP-214) [105]. The mentioned standards are extensive and provide the means for enabling tool data exchange capabilities in their respective domains. Related to the system life-cycle presented in Figure 1.1 the reviewed standards provide data coverage for the detail design and development, and construction and production phases in their respective domain. But there is limited or no support for representation of system specification and system design data in the conceptual and preliminary design phases. As a consequence there are neither any standard means for data exchange for the engineering tools used in conceptual and preliminary design nor does there exist a standard framework that supports traceability through the phases in the systems development process.

1.2 Research Problem

The aim of the work reported is to investigate and propose an information model for reliable Systems Engineering tool data exchange and capabilities.

More specifically the following items were investigated:

- What data representations are used in tools and methods used by systems engineers?
- How shall a tool neutral information model be structured to accommodate data from multiple stakeholders, and captured in multiple tools?
- How do Systems Engineering data relate to data representations used in later phases of the life-cycle?

The main objectives were:

- To enable data exchange between engineering tools used in the conceptual and preliminary design phases of the system life-cycle. The explicit objective is to support data representation requirements for existing methods and tools rather than to define new methods.
- To provide for constructs for integration and traceability between conceptual and preliminary design data created in multiple tool environments.
- To provide the structure for enabling traceability between engineering data represented in the conceptual and preliminary design phases and the detail design and development phases.

The information model shall be seen as a complement to existing STEP application protocols. The scope of the information model has been selected to avoid areas where there is a significant overlap with existing STEP application protocols.

1.3 Research Method

The results presented in this thesis originate from work performed in the EU funded SEDRES[1] (Esprit 20496, 1996 - 1999) and SEDRES-2 (IST 11953, 2000 - 2001) projects where the department of Computer and

Information Science at Linköpings Universitet cooperated with Systems Engineering experts from the European aerospace industry: Alenia[ab] (Italy), BAE SYSTEMS[ab] (UK), EADS Germany[ab] (Germany), EADS Launch Vehicles[ab] (France), SAAB[ab] (Sweden) and SIA[b] (Italy) together with the Institut für Maschinenwesen of Technische Universität Clausthal[b] (Germany), the Australian Centre for Test and Evaluation[a] (Australia), the department of Computer Science at Loughborough University[ab] (UK) and EuroSTEP[b] (UK). In the later stages of the project there was also significant interactions with the ISO working group TC184/SC4/WG3/T8/AP-233 and from the International Council on Systems Engineering (INCOSE). In these projects and activities the author has been responsible for modelling architecture and information model development of the AP-233 standard. Part of the research activities has also been performed with support from NUTEK under the COHSY and SEDEX projects. These projects were also performed in cooperation with SAAB.

At the set out of the project the understanding of requirements on data representation for were relatively limited both in academia and in industry. Consequently an “Industry-as-laboratory” [122] approach was selected to allow for frequent exchange of information from the problem domain (industry) to the academic domain and back.

1.3.1 ROLES AND RESPONSIBILITIES

The author’s primary role in the projects was to harmonise industrial requirements, develop and document the AP-233 information model based on data exchange requirements identified in industry. The industrial partners in the projects have used the information model for tool interface development and there have been validation activities in the form of real data exchanges. The effectiveness of the data exchanges was evaluated by representatives from academia (LUCHI and the Australian Centre for Test and Evaluation).

1. SEDRES is an acronym for Systems Engineering Data Representation and Exchange Standardisation.

[a] indicates that the organisation was participating in the SEDRES project.

[b] indicates that the organisation was participating in the SEDRES-2 project.

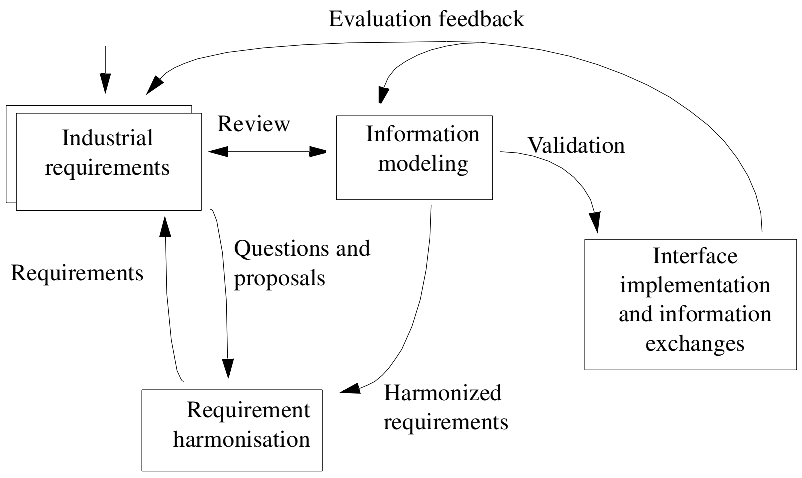


Figure 1.3: Research process

The roles identified do not imply that the author was responsible for all aspects of the information model implementation in the projects. Substantial parts of the AP-233 information models, i.e., the areas covering data types and object oriented Systems Engineering support were developed by Michael Giblin at BAE SYSTEMS and Asmus Pandikow at Linköping University respectively. The contributions made by these very skilled colleagues are not included in the information model documented in this thesis.

1.3.2 RESEARCH PROCESS

Five information model revisions had been implemented to meet gradually more extensive industrial requirements. For each revision industrial feedback has been collected, analysed and where appropriate included in the succeeding revision. For the first, second, fourth and fifth revision there have been extra validation activities in the form of tool interface implementation and real data exchanges to ensure that the concepts modelled were relevant in the problem domain. The process is illustrated in Figure 1.3 and outlined below.

The first activity in the cycle was to allow industrial experts to express their data exchange requirements. This was followed by a requirements analysis and harmonisation activity performed jointly by industry and the information modellers. This activity was necessary to make sure the harmonised requirements were well understood and acceptable to all partners. The harmonised requirements were then used for information model developed. Reviews with industrial specialists were held during the development phase. This early review step provided industry representatives with an opportunity to comment on the solutions proposed in the information model and also provided an opportunity for motivating decisions taken. The review cycle was iterated multiple times for each information model revision. For the validation activities data exchange interfaces was developed for a set of tools in use in the Systems Engineering process in industry. Data exchange of real system specifications was then used to validate the interfaces and the concepts included in the information model. The feedback captured in this activity have had an impact on both the information model and on requirements as it provided new insights in the data exchange problem valid for the succeeding revision of the information model.

1.4 Contributions

The main contributions in this thesis are:

- A set of general guidelines for information modelling for data exchange. The guidelines have been applied consistently to data exchange information model.
- A tool independent information model for Systems Engineering data exchange.

The contributions in each area are outlined further below:

1.4.1 MODELLING GUIDELINES

The identified set of modelling guidelines describes philosophy applied for the development of the Systems Engineering data exchange information model. The rules emphasize, e.g., the importance of a data exchange information model to be process and method independent. The guidelines have been applied consistently to the information model and have motivated many important design decisions.

1.4.2 SYSTEMS ENGINEERING DATA EXCHANGE INFORMATION MODEL

An information model for Systems Engineering data exchange has been developed. The main purpose of the information model is to enable data exchange and design data traceability for data stored and manipulated in multiple tools. The information model contains structures for defining what a system shall perform and other non-functional characteristics, for how the specification evolves over time and for capturing the process the system was developed within.

The main parts on the information model allow representation of:

1. A system from multiple viewpoints and in the context of a system composition hierarchy.
2. Requirements on a system stated in text or models
3. Functional architecture of a system. There is support for representing data in accordance with the modelling methods commonly used within Systems Engineering.
4. A representation of high level architecture of the physical or logical components of a system
5. Information for verifying the correctness of a system
6. Activities carried out in the engineering process and the relationship to data referenced and produced in the process.

Substantial parts of the information model have been validated through tool interface development and real data exchanges in the SEDRES and SEDRES-2 projects.

It may be argued that there is little new in the information model scope presented above. This is true, but the objective with the research is not to define new methods but to support the integration of data created using existing methods. The novel aspects with the information models presented is precisely this integration or tool independent aspect.

1.5 Disclaimer

The work presented herein has in part been performed in the European research projects SEDRES-1 and SEDRES-2, in the ISO working group TC184/SC4/WG3/T8/AP-233 and in the Swedish research projects COHSY and SEDEX. Although the thesis is based on material produced for standardisation purposes its content does neither completely reflect the contents of any standard document nor does the information model fragments presented herein completely represent the structure of past or future versions of the ISO 10303-233 standard.

1.6 Thesis Overview

This thesis is divided into four parts.

Part I - Preliminaries

Contains an introduction to product data modelling and Systems Engineering. The scope of the work presented in the thesis is defined and constraints are justified. Part I consists of chapter 1 to 3 of the thesis.

Part II - Information modelling

In this part the information modelling guidelines applied in the implementation of the information model is presented in chapter 4, followed by an overview of the information model in chapter 5. These chapter present the principles that guide and constrain the implementation of the information model.

Part III - Information model presentation

This part constituting chapters 6 to 11 contain detailed presentations of the information model capabilities including examples indicating how the information model is intended to be used. Each chapter starts with a section presenting identified requirements and constraints governing the scope of the model, followed by a presentation of the information model and sample instantiations. The information model presentations are necessary very detailed. The thesis as a whole can be read and understood without reading and understanding of every detail of the information model.

Part IV - Evaluation

Chapter 12 presents the evaluation activities undertaken to verify the appropriateness of the information model. Evaluation has been performed by peer reviews with participants from INCOSE and ISO 10303, by tool interface development and tool data exchange in small controlled evaluation scenarios as well as in an industrial context. Evaluation results from tool interface implementation and data exchange activities has mainly been obtained from the SEDRES-2 project, but non SEDRES-2 evaluation results are also presented.

Finally the thesis is concluded with chapter 13 containing a summary, overall conclusions prompted by the work presented and an outline of potential future work.

Chapter 2 Framework

Efficient data exchange between computer based engineering tools, or more generally — databases, require an agreement on format and semantics of the data exchanged. This chapter introduces basic terminology and methods for information modelling and reviews background information on methods for the integration of multiple heterogeneous databases. Two database integration architectures, *tightly* and *loosely coupled schema integration* is introduced and compared for the their applicability for tool data exchange.

Finally the ISO 10303 (STEP) standard framework is introduced and compared with other existing data exchange frameworks. The special focus on STEP is due to the fact that it is the framework used for the work presented in this thesis.

2.1 Product Data

A large amount of information is generated in the development process for any non trivial product. This information covers multiple aspects of a product in all its life-cycles [1]. For complex products it is common that substantial parts of the generated information is captured and maintained using computer based tools. The range of tools employed vary and may include word processors, CAD and CAM systems, project management or requirement management tools. With the exceptions of word processors the type of tools mentioned above are all specialised for a particular set of tasks. In this thesis we refer to this set of tools as *engineering tools*. The data captured in an engineering tool is a representation of information related to a facet of a product at a certain level of abstraction suitable for communication, interpretation or processing. We use the term *product data* [6] to refer to this kind of data. The scope of product data is very wide. It can be in the range from a set of high-level requirements on a complex systems to very detailed specifications of discrete components including geometry data.

There are multiple factors that make management and exchange of product data non-trivial [115] [145] [149]:

- Data heterogeneity: There is a large number of data types used for capturing product data ranging from textual documents, over requirements management data over CAD and CAM models to product maintenance data.
- Product complexity: For a single product there may be multiple views that define the product from different perspectives or in different life-cycle phases.
- Product structure complexity: A product may be included in multiple product structures and for each structure there may be temporal constraints applied such that the product may only be valid for inclusion a limited time period.
- Design process complexity: This kind of complexity is due to the iterative nature of the product development process. A high frequency of updates and changes can be expected. It is crucial that all members of a product development project have a coherent view of the product under development. Moreover, for projects with stakeholders from multiple domains there may be cases where different terminology are used to refer to shared product properties.

The listed factors have contributed to the development of dedicated support systems for product data management (PDM). More information on PDM systems are presented in [1] [23] [144].

2.2 Product Data Representation

This section presents some basic assumptions on how product data is represented, managed and organised. We assume that product data is managed in a *database* or *repository* in a logical structure defined by a *schema*. In this thesis no assumptions is made on the complexity and the services offered by a database system. It may be a simple tool operating on a sequential file structure or an advanced database management system. A schema may be defined with the intention to be implemented in a particular database system, in this case it is called a *data model*. An *information model* or *conceptual model* is a schema that is independent of any particular implementation [79] [133].

The structure of a schema or data model is defined in a set of rules defined using a *data modelling* or *information modelling language*. A large number of graphical modelling methods and textual modelling languages have been proposed.

Graphical methods include the Entity-Relationship method proposed by Chen [31], the OMT method [129] and the UML static structure diagram [130]. In these methods the domain of the world of interest is described in terms of entities or objects, relationships between entities and attributes of individual entities. Some of the more recent methods allow for declaration of specialisation relationships between entities.

Textual modelling languages generally allow for representation of more detail than graphical one, but interpretation of a textual model is perceived to be more time consuming compared to interpretation of a graphical model. Textual modelling languages proposed include the functional language Daplex [138] and the extended entity relationship language GEM [157]. The EXPRESS language [8] is preferred within STEP and is used in the work presented herein. It is a hybrid as it is a textual language, but there is also a graphical component, EXPRESS-G, that can express a subset of the language.

2.3 Schema Heterogeneity

In this section an analysis of sources of schema[1] heterogeneity is presented. An information model or a conceptual schema formalises information about a domain in an unambiguous way at a selected level of abstraction. Schenk and Wilson [133] propose the following definition for information models:

> "An information model is a formal description of types of ideas, facts and processes which together form a portion of interest of the real world and which provides an explicit set of interpretation rules"

However, it is important to note that information models capturing similar portions of the real world, but developed by different stakeholders may have fundamentally different structures. This is due to many factors, including the choice of modelling language, the modelling style applied, the purpose of the model and the abstraction selected when a specific concept is captured. For instance, if a concept is at the centre of interest in one model and in the periphery of another then the concepts are likely to be captured at different levels of abstraction in the two models. Even in cases where modelling language, domain, purpose and the selected level of abstraction coincide for multiple schemas it can be expected that there will exist structural differences in the schemas. Even in trivial cases there are multiple modelling alternatives for capturing the same information. For instance, if representing facts about people is in scope of a schema then the gender of a person may be captured using an attribute or by using subtypes as illustrated in Figure 2.1.

1. In this section the term schema is used as a synonym for information model as this is the term preferred in the database community.

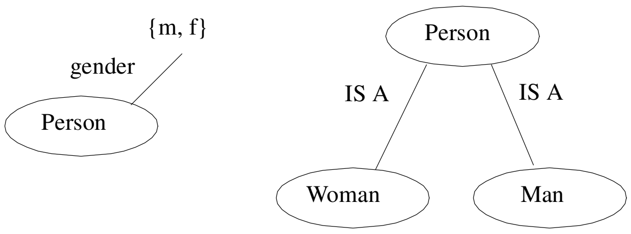


Figure 2.1: Two alternative models for a trivial domain

In the database community there is a long tradition of research in schema integration, identification and resolution of schema heterogeneity for multi-database integration, e.g., [65] [77] [135]. Multi-database research is largely focused on methods for enabling inter-database communication for enabling updates and, in particular, queries spanning over multiple databases. Analyses of the spectrum of database heterogeneity encountered are presented by, e.g., Fang et al. [46] and Kim and Seo [82]. In the work by Fang et al. five aspects of database heterogeneity is considered.

- *Meta-data language heterogeneity*: The component databases may use different classes of languages for structuring data. For instance, one database may utilise the relational data model and another may use object relational model. Heterogeneity in this aspect also includes differences in techniques for capturing model rules and constraints.
- *Meta-data specification* or *conceptual schema heterogeneity*: Component databases may use independently developed schemas with different scopes, abstractions or implemented using different structures.
- *Object comparability heterogeneity*: Component databases may agree to a common conceptual schema, but there are differences in how specific facets of information are represented between the components. Also the interpretation of atomic data values may differ across databases.

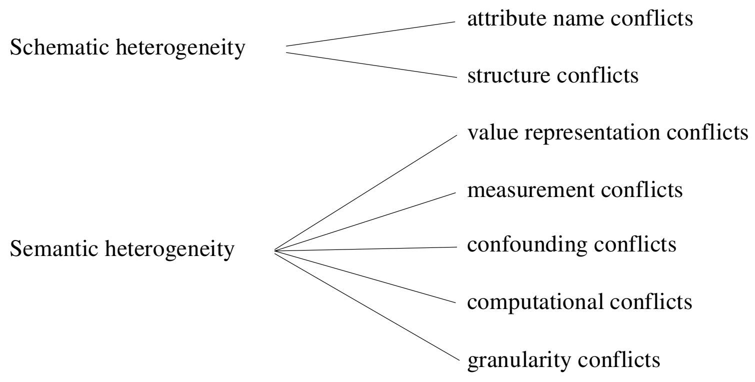


Figure 2.2: Schematic and semantic heterogeneity

- *Data form/format heterogeneity*: Component databases may agree on the language, schema and object level, but may use different low-level representations for representing atomic data values.
- *Tool heterogeneity*: Component databases may use different tools to manage and provide an interface to their data. This kind of heterogeneity may exist with or without the aspects described above.

Goh et al. [50] use a different classification and extend on the definition of object comparability and data format heterogeneity as defined above by considering:

- Schematic heterogeneity
- Semantic heterogeneity

The definition nature of schematic and semantic heterogeneity is presented in Figure 2.2:

2.3.1 SCHEMATIC HETEROGENEITY

Schematic heterogeneity includes attribute name conflicts and schema structure conflicts. *Attribute name* conflicts include cases where different names are used to capture the same concept in different schemas (synonyms) and the cases where the same name captures different concepts in different schemas (homonyms).

Structure conflicts are a result of the same piece of information being captured in different conceptual structures. A concept in one schema may be captured by a set of entity attributes, while being captured by a relationship in another. Figure 2.1 illustrates a structural conflict. Methods for analysis and resolution of structural conflicts have been proposed by, e.g., Johannesson [72] and Batini et al. [21]. The transformation rules proposed in the cited work are not only applicable for resolution of structural conflicts but also serve as a guideline for good information modelling practise.

2.3.2 SEMANTIC HETEROGENEITY

Semantic heterogeneity originates from multiple interpretations of attribute values. As with schematic heterogeneity there may be *representation conflicts* in attribute values. These occur when synonyms or homonyms are used to represent the value range of an attribute. For instance, the priority of a requirement may be captured on the binary scale *high, low* in one system and while the values *important* and *normal* may be used to capture the same semantics in another system.

Measurement conflicts occur when different units of measurements or scales are used in different schemas to represent common information. For an example of measurement conflicts consider two schemas designed for capturing the weight of some objects of interest where one assumes the weight is given kilograms and the other assumes imperial pounds. An illustrating example of the impact of measurement conflicts is given in [40]. In the cited example measurement data for a product was assumed to be given in inches when they in fact where given in millimetres. As a result the product was realised 25 times larger than intended!

Representation conflicts arise when different syntactical representations are used to capture the same attribute value. For instance some representations may encode numeric values using fractions, e.g., $5\frac{5}{16}$ or real values may be used, e.g., 5.3125. Similarly, in some countries, e.g., Sweden, a comma (,) is used as decimal delimiter while other countries, e.g., the UK, use the decimal point (.) as delimiter.

Confounding conflicts are due to assignment of different meanings to a common concept. For instance, the weight of a system may be the weight as specified or the weight as realised.

Computational conflicts are due to the use of different methods or algorithms to compute a value. Finally *granularity conflicts* occur when data are managed at different levels of abstraction in different databases.

Detection of semantic heterogeneity may appear trivial, but is complicated by the fact that contextual information is frequently implicit or assumed to be unambiguous in the context of a single schema. Problems materialise when the implicit or explicit assumptions made in one schema is not taken into account when data is be transformed from one schema representation to another.

2.4 Schema Mapping

This section presents a framework for reasoning about the consequences of the mapping concepts from an arbitrary source schema A to a sink schema B.

A mapping function f_{map} defines how concepts in two schemas relate semantically and schematically [39] [73] [87]. If a pair of schemas (A, B) is considered then a mapping function can be defined to capture how a specific concept in schema A shall be represented in schema B. Thus a mapping function may include the resolution of schematic and semantic heterogeneity. A general mapping function for a finite, non-empty set of elements (entities, relationships and/or attributes) in a schema A to a set of elements in schema B can be defined as:

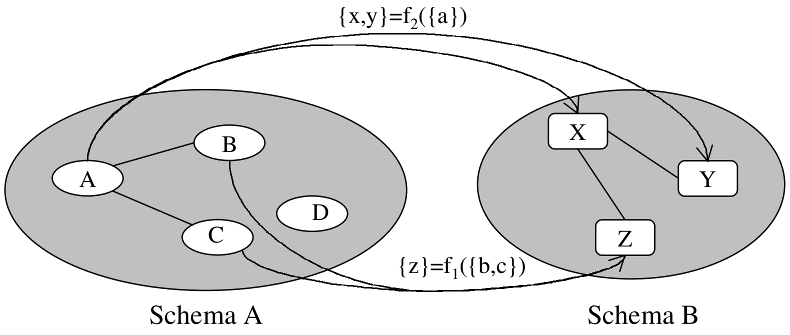


Figure 2.3: Schema mapping function example

$$b = f_{map}(a)$$

where

$$a = \{a_1, a_2, \ldots, a_n\} \in schema\ A \wedge$$
$$b = \{b_1, b_2, \ldots, b_m\} \in schema\ B \vee \varnothing \wedge$$
$$m, n > 0$$

The empty set of elements is included in the value domain of a general mapping function to illustrate the case where there exists no corresponding concept in the sink schema. Even though a mapping function is defined to operate on sets of elements, it is important to keep in mind that it operates on a subset of a schema. It is expected that several mapping functions will be required to completely define the relationships between a pair of schemas. Two mapping functions are illustrated in Figure 2.3. Mapping function f_2 illustrates the case when a concept is represented using a single entity in the source schema and several entities are required in the sink schema. Similarly, mapping function f_1 is an example of the case where several entities in the source schema are represented by a single entity in the sink schema. The fact that a mapping function may

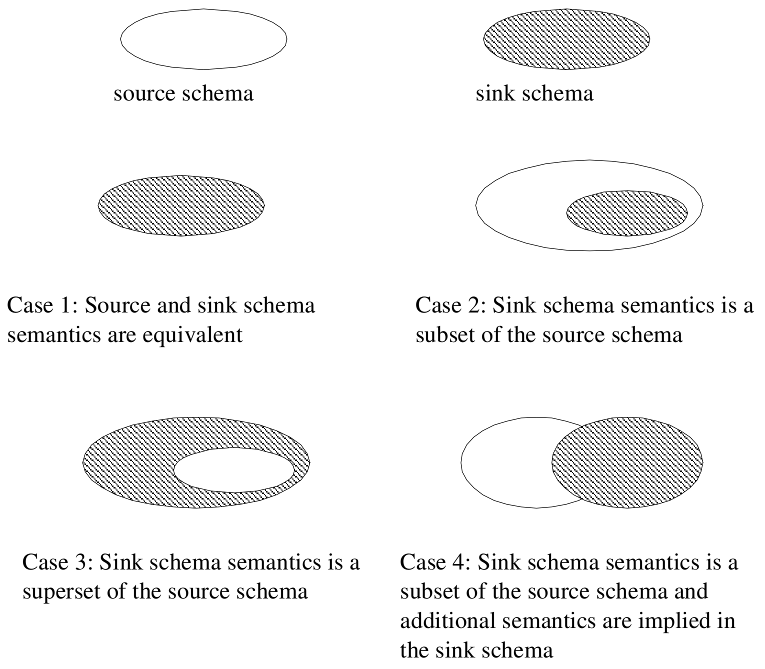

source schema
sink schema
Case 1: Source and sink schema semantics are equivalent
Case 2: Sink schema semantics is a subset of the source schema
Case 3: Sink schema semantics is a superset of the source schema
Case 4: Sink schema semantics is a subset of the source schema and additional semantics are implied in the sink schema

Figure 2.4: Mapping function classes

resolve schematic heterogeneity, e.g., multiple elements in a source schema may map to a single element in a sink schema or vice versa, does not in itself imply information modification through the application of the mapping function.

2.4.1 IDENTIFYING SEMANTIC HETEROGENEITY

Mapping function quality is not just a matter of comparing entities, relationships and attributes of the involved schemas. Mapping functions must also be analysed with regard to semantic heterogeneity. It may be the case that there exists a natural mapping from elements in schema A to elements in schema B, but the resulting representation in schema B may have a different semantics compared with the original one. Four cases can be identified:

1. The application of a mapping function results in a representation with equivalent semantics in the sink schema.
2. The application of a mapping function results in a representation whose semantics is less specific than the original one. Semantic heterogeneity between the schemas results in the loss one or more properties when the mapping function is applied. In extreme cases no information at all is conveyed by the mapping function.
3. The application of a mapping function results in a representation whose semantics is more specific than the original one. Semantic heterogeneity result in the addition of one or more properties when the mapping function is applied.
4. The application of the mapping function results in a representation whose semantics is in part less specific than the original and in other aspects more specific than the original one. This is a combination of the second and third alternative above.

The Venn diagrams in Figure 2.4 illustrate the properties of the four classes. The characterisation of mapping functions presented above is similar to that of attribute equivalence for databases presented by Larson et al. [87] with the difference that mapping to and from more than one object or attributes are considered in the work presented herein.

The following trivial examples illustrate how the application of a mapping function modifies a specification. Assume that a source schema has the static capability to represent two classes of requirements - functional and non-functional requirements. In the representation selected in the source schema, a requirement is either functional or non-functional. If the sink schema also supports the definition of two requirement classes with equal definitions as for the sources schema then case 1 above applies. No information will be lost or added in the mapping of classification information from schema A to schema B. It is important to note that the requirement classes defined in schema B need not carry the same names as the ones in schema A. It is sufficient that the underlying definitions are equivalent.

If the same source schema A, and the mapping to a schema where there is no provision for requirement classification, is considered then this information will be lost in the transfer. This corresponds to case 2 above.

Case 3 above is illustrated by the following example. If the same source schema A is considered but with a mapping to a schema with four requirement classes, e.g., functional, performance, physical and constraints. The definition of a functional requirement may be common to both schemas, but the mapping of a non-functional requirement to any of the classes defined is a mapping from a general to a specific concept. If mapping is performed automatically then a non-functional requirements in the source schema will receive a more specific (and possibly incorrect) classification in the sink schema.

2.4.2 SEMANTIC HETEROGENEITY ANALYSIS

In the preceding paragraphs, schema overlap has been discussed in terms of the semantics of individual mapping functions. It can be expected that a large set of mapping functions must be applied if two schemas with substantial overlap are considered. The five classes defined in Table 2.1 can be used to facilitate analysis of the overlap between two schemas. Note that a fifth class, the Inequality class, has been added compared to the previous enumeration above to explicitly represent the set of mapping functions that do not carry any information.

Table 2.1: Mapping function classes

Name	Description
Equality	A class for the mapping functions whose application results in equivalent semantic in the source and sink schema.
Restriction	A class for the mapping functions whose application results in a semantic in the sink schema that is more specific than the original semantics.

Table 2.1: Mapping function classes

Name	Description
Generalisation	A class for the mapping functions whose application results in a semantic in a sink schema that is more general than the original semantics.
Distortion	A class for the mapping functions that exhibit characteristics of both the Restriction and Generalisation classes.
Inequality	A class for the mapping functions for which no representation exist in the sink schema.

Whenever a mapping function that does not belong to the *Equality* class is applied the result will be a modification, however slight, to the original specification semantics. The effect of a mapping function modification may not be visible in the sink environment. The data in the sink environment may be semantically correct but not semantically equivalent with the original. The extent of a modification can only be established through an analysis of the mapping function and the schemas involved.

The analysis would be significantly simplified if all mapping functions either belonged to the *Equality* or the *Inequality* classes as the extension of the modification imposed by mapping functions belonging to these classes are bounded. Either there is no modification or no data is carried over by the mapping function. For the other three mapping function classes the extent of the modification imposed cannot be bounded without a detailed study of the characteristics of each individual function.

2.4.3 SCHEMA MAPPING — SUMMARY

The value of mapping specifications from one schema representation to another must be evaluated against the impact of mapping function related modifications incurred in the process. The extent of modification that can be accepted is situation dependent. In some cases minor semantic modifications may be sufficient to nullify the value of data exchange. In other cases there may be a high level of tolerance for semantic modifications.

Regardless of the case, the extent of modifications incurred in the mapping process must be understood by the users utilising the data exchange environment or there is a substantial risk that critical modifications is not considered at all.

2.5 Database Integration Architectures

In the database community there have been multiple proposals for architectures for integrating multiple autonomous database systems. A survey of different approaches and systems are presented in [48]. Two classes of architectures can be identified based on the point in time when integration is performed.

- In *tightly coupled integration* the focus is on resolution of schema heterogeneity through the development of shared schemas that hides heterogeneity from the user. Sometimes a distinction is made between global and federated schema architectures [77]. The approach in a global schema system is to create a single schema for all component databases while there may be multiple views (schemas) defined in a federated schema system [86]. Database queries are performed against the global schema or a federated schema. *2n* schema mappings have to be performed to integrate *n* schemas. This approach is based on the assumption that schema heterogeneity can be identified and resolved in a global schema or a set of federated schemas through analysis of component databases schemas. There is an implicit assumption that the component databases are stable and that changes that impact on the structure on the global or federated schema(s) are infrequent. An additional reservation made against this approach is that the global schema may become very complex if there is substantial inter-schema heterogeneity. An example of a tightly coupled integration architecture is presented in [93].
- Proponents for *loosely coupled integration* emphasise the difficulty in constructing and maintaining a common schema for a large number of autonomous databases. Instead the focus is on the definition of powerful data manipulation languages that allow queries of multiple data

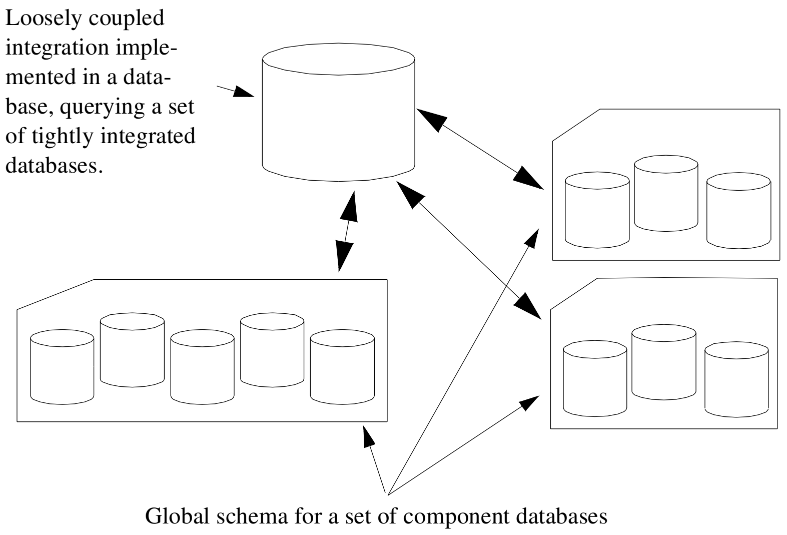

Loosely coupled integration implemented in a database, querying a set of tightly integrated databases.
Global schema for a set of component databases

Figure 2.5: Combining tightly and loosely coupled integration

sources. In a loosely coupled architecture the user is responsible for detection and reconciliation of schematic and semantic conflicts [49]. No global schema is developed, instead mappings are developed on a schema pair basis. Thus, in an environment with *n* component databases there may be up to *n(n-1)* schema mapping alternatives to be considered, but in each alternative heterogeneity need only be analysed and resolved against a schema pair. In cases where semantic or schematic heterogeneity exist the database integrator may select a single data source as the reference. An example on a loosely coupled integration architecture is presented in [92].

Selection of integration architecture depends on the characteristics of the constituent databases. A tightly coupled integration approach appears advantageous in cases where the heterogeneity of the component databases is well understood and bounded. Furthermore component schemas must be assumed to be stable over time. On the other hand, a loosely coupled integration solution appears appropriate in cases where component databases are subject to frequent changes, where there is substantial heterogeneity in component schemas and where there exist a limited number of data sources where the data can be retrieved.

Finally it must be noted that architectures selection is not mutually exclusive. Groups of tightly coupled integrated databases may be loosely integrated with each other as illustrated in Figure 2.5, and a schema that implements loosely coupled integration may be a component schema in a tightly coupled integration architecture.

2.6 Engineering Tool Data Exchange

The preceding sections introduced and discussed database heterogeneity, schema integration and schema heterogeneity as they have been presented in the database community.

In the terminology introduced by Fang et al. [46] (presented in Section 2.3) data exchange between a pair of tools may be of value if *conceptual schema heterogeneity* is bounded, i.e., there is a substantial overlap in tool domain support. There may be substantial heterogeneity in the *meta-data language*, in *object comparability* and *data formats* classes. All heterogeneities must be identified and understood before specification elements exchanged can be used.

The integration architectures identified in Section 2.5 are applicable to tool data exchange as well. Loosely coupled schema integration corresponds to developing mapping functions directly between the constructs used in a source schema and the constructs in the sink schema. The data exchange mechanisms may be implemented via direct queries or via file based data exchange. In directed file based data exchange from tool *i* to tool *j* this can be implemented through the use of the native file format of either tool as the common data format as in Figure 2.6 (top). A set of mapping functions (*A* in Figure 2.6) resolve heterogeneities and present data in a format suitable for tool *j*. The set of mapping functions are specific for the exchange from tool *i* to tool *j*.

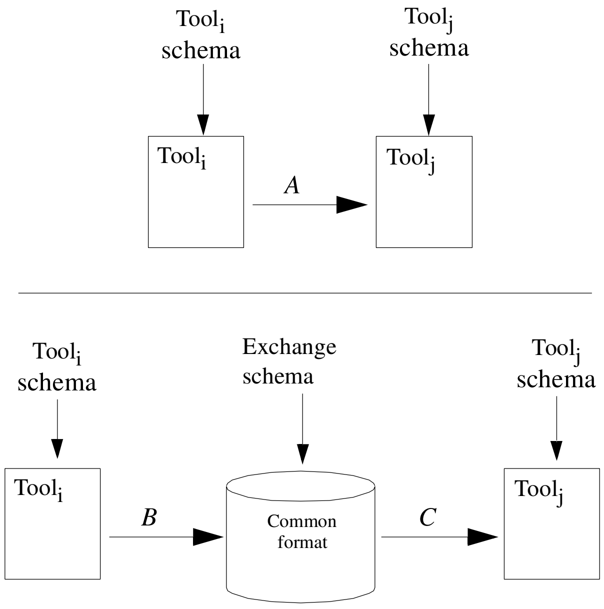


Figure 2.6: Approaches to data exchange

Tightly coupled schema integration corresponds to the development of an exchange schema rich enough to resolve conceptual schema heterogeneity for engineering data relevant for a domain and the selection of a common data format for representing instances of the information defined in the schema. The architecture is illustrated in Figure 2.6 (bottom). Two sets of mapping functions (*B* and *C* in Figure 2.6) resolve heterogeneities from a tool specific source schema to the exchange schema and from the exchange schema to a tool specific sink schema.

Advantages and disadvantages of respective approach for generic databases have been discussed in Section 2.5. For well-established and homogeneous engineering domains the tightly coupled schema integration approach appears advantageous. If there exist a common view on the information managed by a specific class of tools then schema harmonisation can be expected to be relatively straightforward and schematic and semantic heterogeneity issues are likely to be minor.

The tightly coupled schema integration architecture is also attractive in cases where domain information exhibits a high degree of complexity, e.g., for engineering information [115] [28]. In such cases the cost of developing and maintaining multiple tool interfaces become prohibitive. The total development cost for a global schema may be substantially lower than for maintaining multiple loosely integrated tools, especially if the global schema architecture is coupled with a common data exchange format.

A comparison of the two integration architectures for engineering data exchange yields the following results.

1. *2n* interfaces are required to enable data exchange capabilities among *n* tools if a tightly coupled integration architecture is used compared with up to *n(n-1)* interfaces with a loosely coupled integration architecture.
2. An agreement on a common exchange schema coupled with a formal mapping to a data exchange file formats and access primitives allows for automation of large parts of the interface development process. File readers/writers and temporary and permanent repository structures with standard access functions can be generated. Those parts that are not easily automated, i.e., the mapping from entities in the exchange schema to the corresponding entities in a tool schema, are supported by the definitions available in the exchange schema. This shall be compared with the manual process of mapping between two tool specific formats that has to be used if a loosely coupled integration architecture is employed.
3. A tightly coupled integration approach allows domain experts to express their views on important domain information. In this sense the approach not only facilitates information exchange across tool boundaries — it also allow users to express requirements on future tools.
4. A tightly coupled integration architecture is less efficient than a loosely coupled integration architecture as two sets of mapping functions must be applied in order to complete a data exchange. The risk for mapping function introduced modifications is thus doubled.
5. A tightly coupled integration approach is less specific and less flexible

than a loosely coupled one as the global schema cannot incorporate all concept of all tools and modifications cannot easily be introduced in the common schema [116].

Despite the known drawbacks of tightly coupled integration it has proved very successful for enabling data exchange across engineering tool boundaries. The STEP framework reviewed in next section has proved especially successful mainly in the mechanical engineering domain, and is also the framework selected for the work presented later in the thesis.

2.7 STEP

This section introduces the STEP[1] (ISO 10303) [6] standard framework, its background, objectives and constituent parts. The overview is necessarily brief, in depth information on STEP and its components are available in [81] [116] [133].

The objective of STEP is to provide the framework for the unambiguous representation and exchange of computer-interpretable product data throughout the life of a product [6]. The framework is independent of any particular computer system and is partitioned into a large number of parts. The STEP framework is evolving constantly. In this section the original architecture is presented first, followed by an analysis of the approach, descriptions of additions and modifications applied to overcome identified problems.

STEP parts belong to one of the following content dependent classes:

Description methods The description methods are used in the definition of *integrated resource* and *application protocol* classes described below. The description method preferred within STEP is the language Express [8]. An overview of EXPRESS is provided in Section 2.8.

Implementation methods An implementation method defines a standard data representation format and the mapping from a *description method* to the data representation. There are implementation methods defined for

1. STEP is the abbreviation for STandard for the Exchange of Product data.

file based exchange [ISO10303-21] and generic application programming interfaces SDAI (Standard Data Access Interface) [123] for repositories including programming language bindings for C++ [104], C [94] and XML [124].

Conformance testing methodology and framework Defines the procedures for validating STEP data exchange implementations. The existence of a published testing methodology is a prerequisite for independent evaluation of the quality of tool interfaces.

Integrated resources The integrated resources are a set of generic product data model fragments that are potentially common to multiple *application protocols*. Integrated resources are used as a basis for application protocol development and are not intended for direct implementation. They define reusable components intended to be combined and refined to meet a specific need. The existence of a common baseline of integrated resources is the corner stone for application protocol interoperability and also defines an information modelling style common to all STEP application protocols. There are two sets of integrated resources: Generic resources (part 4X) which are application and context independent, and application resources (part 10X) which are developed for a specific application areas common to many domains. Examples of the first category include 10303-41 Integrated generic resources: Fundamentals of product description and support [9] The data architecture defined by the integrated resources is further discussed in Section • in this chapter.

Application protocols The application protocols provide the definition of data representation requirements identified for an engineering domain, e.g. AP-203, Configuration Controlled Design [7]. The vast majority of application protocols produced up to date are focused on different aspects of mechanical engineering. Notable exceptions are AP-210, Electronic printed circuit assembly, design and manufacture, and AP-212, Electrotechnical plants.

An application protocols is a substantial document. It is not uncommon with documents larger than 2000 pages. The structure of an application protocol is outlined in Figure 2.7.

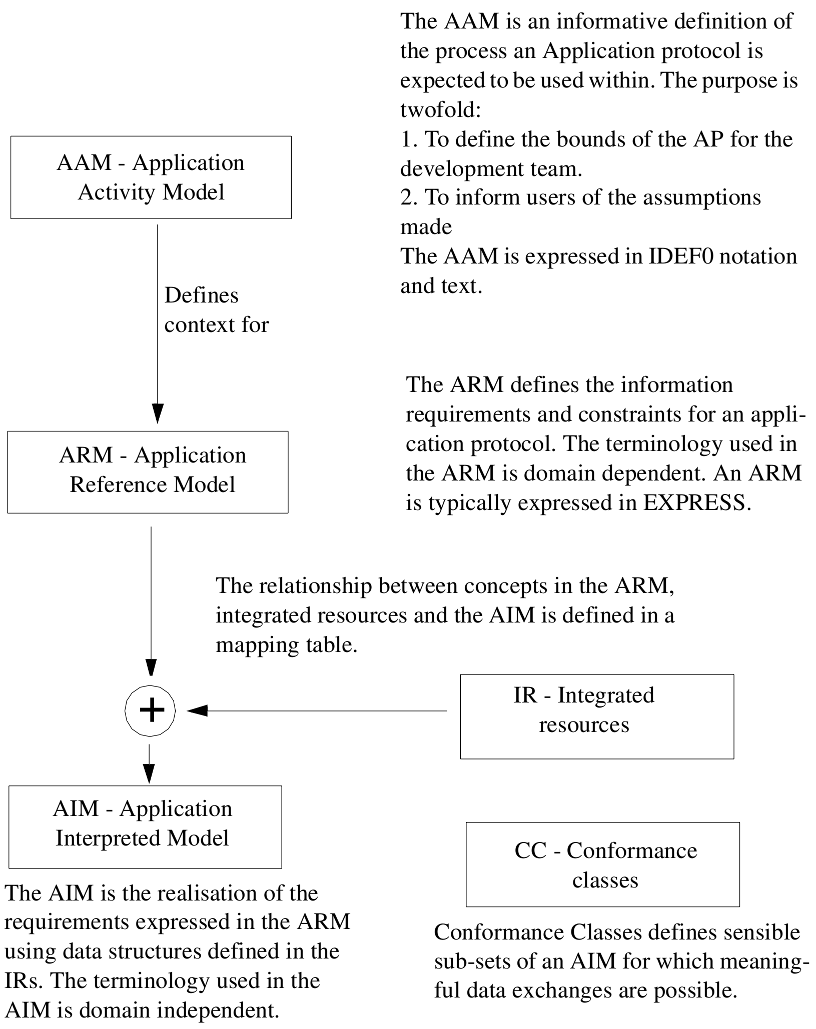

AAM - Application Activity Model
The AAM is an informative definition of the process an Application protocol is expected to be used within. The purpose is twofold:
1. To define the bounds of the AP for the development team.
2. To inform users of the assumptions made
The AAM is expressed in IDEF0 notation and text.
Defines context for
ARM - Application Reference Model
The ARM defines the information requirements and constraints for an application protocol. The terminology used in the ARM is domain dependent. An ARM is typically expressed in EXPRESS.
The relationship between concepts in the ARM, integrated resources and the AIM is defined in a mapping table.
IR - Integrated resources
AIM - Application Interpreted Model
CC - Conformance classes
The AIM is the realisation of the requirements expressed in the ARM using data structures defined in the IRs. The terminology used in the AIM is domain independent.
Conformance Classes defines sensible sub-sets of an AIM for which meaningful data exchanges are possible.

Figure 2.7: Application protocol structure

Abstract test suites Along with each application protocol there shall be set of test cases including definition of example data files defined so that implementers can validate their implementations against the application protocol requirements.

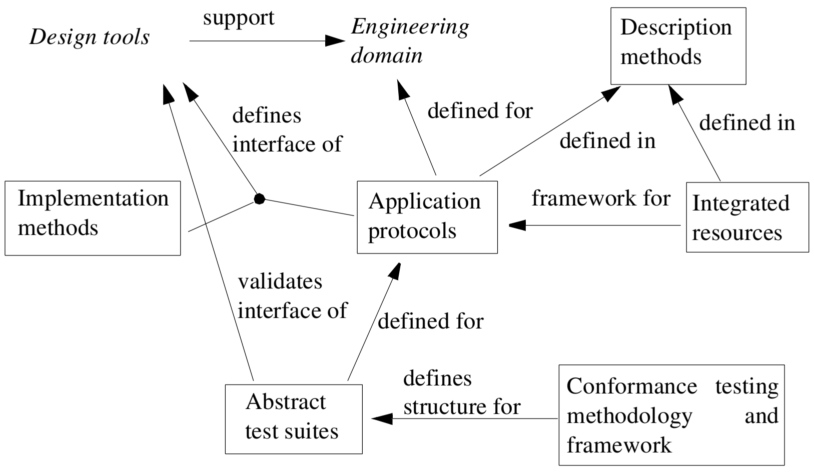


Figure 2.8: STEP classes and their relationships

The relationships among the six classes outlined above are illustrated in Figure 2.8.

2.8 The EXPRESS Language

This section presents the capabilities of the information modelling language EXPRESS [8] and its graphical format EXPRESS-G. EXPRESS is based on an extended entity relationship formalism. EXPRESS supports the definition of:

- Schemas: The mechanism for grouping related model concepts. Inter-schema referencing is possible that allows for a common resource to be defined independently and then used from several other schemas. Each schema has its own unique name space.
- Entities: The representation of a concept of interest within the scope of a schema.

- Basic types: Elementary types that cannot be further subdivided. STEP supports the normal set of basic types found in standard imperative programming languages, e.g., the integer and string type.

EXPRESS offers multiple capabilities for defining entity and type properties. From an information modelling viewpoint a property may be expressed using combinations of the constructs below.

- Attribute: represents an aspect of an entity. EXPRESS provides the capability to define mandatory, optional and derived (attribute value is calculated using a formal expression) attributes.
- Inheritance relationships: The specialisation/generalisation relationship between entities. There are three basic constraints that can be defined for an inheritance relationship: Under the *One of* constraint an instantiation of a supertype entity is exactly one of the subtypes. The *And* constraint defines that an instantiation of a supertype has the combined properties of all of its subtypes. Finally the *AndOr* constraint defines that an instantiation of a supertype possess the properties of any combination of the subtypes of the entity. Inheritance constraints may be combined using regular expressions. EXPRESS also support the definition of multiple inheritance, i.e., a subtype may inherit the properties of multiple supertypes.
- Aggregates: attributes may be defined to be an aggregate of a basic type or entity. Aggregates may be ordered or unordered, closed or open-ended.
- Textual constraints: EXPRESS supports the definition of formal constraints on entities, relationships, attributes and other modelling constructs. Uniqueness constraints can be defined such that only a single instance of an entity with a given attribute value or value combination is allowed in database. Textual constraints can be applied to basic types and to entities. In addition it is possible to define global rules that are not associated with any specific element. The expressive power of EXPRESS is comparable to the combination of UML and OCL as used in [109].

The graphical view of EXPRESS, EXPRESS-G is a subset of the textual view in that it does provide representations for entities, relationships and attributes, but there is no representation of, e.g., textual constraints or advanced inheritance relationship information.

2.8.1 EXPRESS MODEL EXAMPLE

This section introduces the EXPRESS language through a trivial, and not necessarily realistic, example. The portion of the world of interest for the example concern vehicles and audio systems. The following statements define the scope of the model.

- An audio system may be installed in at most one vehicle.
- An audio system has either support for a cassette or a CD.
- For each vehicle there exist an upper limit of audio systems that may be installed.
- Each vehicle and audio system can be identified through a serial number

An information model in EXPRESS-G meeting the statements above is presented in Figure 2.9. The syntax in EXPRESS-G is explained below:

- Entities are depicted as rectangles enclosed in solid lines. Two of the entities in Figure 2.9 are *vehicle* and *audio_system.*
- Entities may be defined as being abstract, indicating that objects of an entity may not be instantiated without the instantiation of one of its subtypes. Abstract entities are identified by the keyword (ABS). In Figure 2.9 *audio_system* is an abstract entity.
- Basic data types are depicted using solid rectangles with an extra vertical bar close to the right side of the rectangle. In Figure 2.9 the *string* and *integer* data types are used.
- User defined data types are depicted using dashed rectangles. Select types have a vertical bar close the left-hand side and enumeration types a vertical bar close to the right-hand side. The *natural_number* type in Figure 2.9 is a user defined type.

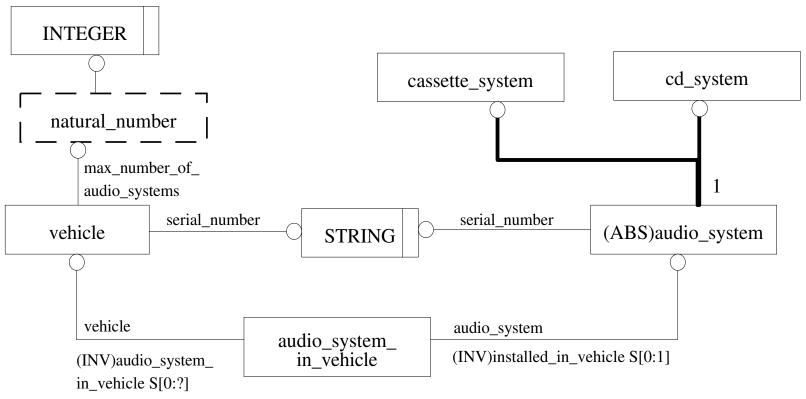


Figure 2.9: EXPRESS-G model capturing the relationship between vehicles and audio systems

- Attributes are depicted as a line from the source to the destination object. A ring indicates the destination object. Optional attributes are depicted with a dashed line and mandatory with a solid line. Attributes can also be defined as sets, arrays and bags. In EXPRESS-G the labels, S, A, and B are used to indicate sets, arrays and bags of objects. Cardinality constraints may be defined on both forward and inverse attributes. Inverse relationships are identified by the keyword (INV).
- Thick lines are used to represent inheritance relationships among entities. A ring is used to indicate the subtype in the relationship. In Figure 2.9 the entity *CD_system* is a sub-type of the entity *audio_system*.

EXPRESS-G can only represent a sub-set of the EXPRESS language. In Figure 2.10 the model presented in Figure 2.9 is extended to illustrate some additional features of EXPRESS.

In the textual representation a model can be extended to include:

- Derived attributes, attributes that can be calculated from other elements in a model. In Figure 2.10 the value of the attribute *installed_audio_system* of the entity *vehicle* is determined by the *audio_system_in_vehicle* objects assigned to each vehicle.

```
SCHEMA example;
TYPE natural_number = INTEGER;
WHERE
  SELF >= 0;
END_TYPE;
ENTITY vehicle;
  serial_number : STRING;
  max_number_of_audio_systems : natural_number;
DERIVE
  installed_audio_system : natural_number := hiindex (audio_system_in_vehicle);
INVERSE
  audio_system_in_vehicle : SET [0:?] OF audio_system_in_vehicle FOR vehicle;
UNIQUE
  serial_number;
WHERE
  wr1 : installed_audio_system <= max_number_of_audio_systems;
END_ENTITY;
ENTITY audio_system
  ABSTRACT SUPERTYPE OF ( ONEOF(cassette_system,cd_system) );
  serial_number : STRING;
INVERSE
  installed_in_vehicle : SET [0:1] OF audio_system_in_vehicle FOR audio_system;
END_ENTITY;
ENTITY audio_system_in_vehicle;
  vehicle : vehicle;
  audio_system : audio_system;
END_ENTITY;
ENTITY cassette_system
  SUBTYPE OF (audio_system );
END_ENTITY;
ENTITY cd_system
  SUBTYPE OF ( audio_system);
END_ENTITY;
END_SCHEMA;
```

Figure 2.10: EXPRESS representation of Figure 2.9

- Uniqueness constraints, defines that a specific attribute value (or a combination of attribute values) for an entity type shall be unique in a database. In Figure 2.10 the attribute *serial_number* shall be unique for all vehicles.
- Rules, declarative rules can be defined to constrain the number of valid instantiations of a model. Rules may either be local to a specific

entity or type, or global. In Figure 2.10 a rule is defined for the *vehicle* entity that ensures that the number of audio systems installed shall be less or equal to the maximum allowed number.

- Specialisation of inheritance relationships, the EXPRESS for the *audio_system* specifies that an audio system is either a CD system or a cassette system. This is stipulated by the *oneof* constraint defined for the entity *audio_system*. Alternatively inheritance relationships may be constrained with *andor* or *and* constraints. If the *andor* constraint were used for the *audio_system* entity in Figure 2.10 then an *audio_system* would be either a cassette system, a CD system or a combination thereof. Under the *and* constraint an *audio_system* would be the combination of a CD system and a cassette system.

2.9 Product Data Modelling in STEP

This section presents the basic data structure defined in the integrated resources in STEP. Since the structure is defined in the integrated resources it is common to all application protocols. This does not imply that the same information is handled the same way in two application protocols. In fact, one of the fundamental problems encountered in application protocol development is to ensure that similar requirements are captured in compatible structures in different application protocols. The problem is that application protocols currently in development may have more extensive requirements than those already standardised. For backward compatibility new application protocols are strongly encouraged to build on the previously accepted structures.

In Figure 2.11 the five basic elements of the STEP data architecture is presented [47]. The main notion is that of a *product*, which may represent anything in the range from utterly complex to exceedingly simple. The product concept is central as it provides the basis version management.

Any number of *formations* (versions) may be defined for a product and relationships between formations may be captured.

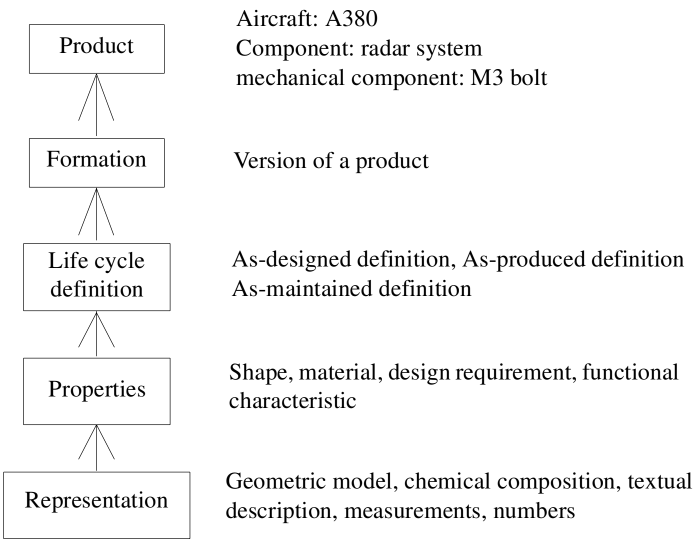

Product
Aircraft: A380
Component: radar system
mechanical component: M3 bolt
Formation
Version of a product
Life cycle definition
As-designed definition, As-produced definition
As-maintained definition
Properties
Shape, material, design requirement, functional characteristic
Representation
Geometric model, chemical composition, textual description, measurements, numbers

Figure 2.11: The STEP data architecture

For each formation there may exist any number of *life-cycle definitions*. This allows for capturing different views on the same product, e.g., a production oriented view and an analysis oriented view.

A product life cycle definition may be characterised by the association of *properties*, where each property is expressed in a defined *representation*. Example of representations are 3D-shape, textual and maths expressions.

2.9.1 CHARACTERISTICS OF A PRODUCT

The notion of *product* is very important to STEP as it defines the class of elements for which configuration and version management is applicable. The definition of what can be encompassed by the product concept has expanded over time. In the original definition *product* was specific and tied to manufacturability. In ISO 10303-41 [9] a product is defined as:

> A product is the identification and description, in an application context, of physically realizable object that is produced by a process.

As STEP has expanded in scope the definition of a product has become less rigid. In ISO 10303-1017, [107] a *product* is defined as:

> A product is something that an organisation has identified for some purpose.

In the same document five distinct classes of *products* are identified:

Part

A part is an item that is intended to be produced or employed in a production process. The item refers not only to the finished part but also to any physical constituent or in-process configuration that makes up the finished part.

Document

A document is a form of information that is controlled and communicated as one single unit. A document may be a physically bound book, collection of pages or may exist only in electronic form as one or more files on some electronic medium. More generally a document is a product of the documentation process and records information about some subject.

Function

The characteristic actions, operations, or kind of work a person or thing is supposed to perform; e.g., the engineering function or the material-handling function.

Raw material

A crude or processed material that can be converted by manufacture, processing, or combination with other raw materials and products into a new and useful product or raw material.

Requirement

A requirement is a statement identifying a capability, physical characteristic or quality factor that bounds a product or process need for which a solution will be pursued.

2.10 STEP Discussion

Since its initial release in 1994 there have been many proposals for extending the scope of STEP and changing the architecture of STEP parts. Many proposals have been related to the following issues:

1. How to ensure application protocol interoperability in an environment where multiple overlapping protocols are developed concurrently.
2. Definition of mechanisms for application protocol extensibility. In the original architecture there is no mechanism for incremental extensions to an application protocol or to combine independently developed application protocols.
3. Shortening application protocol development time. The sheer size of an application protocol make it very time consuming to develop and validate.

The STEP framework allows application protocols to be developed independently and to be added to the catalogue of existing standards one at a time. There are many cases where there exist significant overlaps in scope of application protocols. For instance, AP-214 [105] is effectively a superset of AP-203 [7]. The rule is that overlaps shall be modelled consistently in all affected application protocols, e.g., there shall only be one representation for 3-D geometry.

There are several implications to this rule. In some cases determining the existence of an overlap may be straightforward. This is especially the case if the terminology used and abstractions made are common for the overlapping parts in the ARM of an application protocol. In cases where multiple application protocols share the same requirements the architecture presents no problems as the least mature protocol can reuse all material produced in more mature protocols. The Integrated Resources available in STEP in this case preserve a common modelling style that is acceptable to all stakeholders in the Application Protocol development process. In this perspective STEP defines a tightly coupled schema architecture as defined in Section 2.5.

There are also cases where common concepts are captured at different levels of abstractions in different application protocols. The challenge is to support a detailed view using the same basic data structures used to define a more abstract view while adhering to the structures available in the integrated resources. There is no easy solution to this problem if single representations of common concepts are preferred.

An alternate architecture known as Implementable or Fully attributed ARMs [154] was proposed to complement the traditional AIM based approach to allow for more flexibility. Under the proposed approach application protocols without AIM would be accepted, thus allowing data exchange models completely based on domain views on data. The main arguments put forward for the change was shorter development cycles and consequently lower development cost, higher implementation freedom and easier extensibility into new domains. Overlaps between Application Protocols would be resolved in a loosely coupling approach as defined in Section 2.5.

The concept of standardised Implementable ARMs were dropped and replaced by the Application Integrated Constructs (AIC) [12] intended to define how requirements common to multiple application protocols should be represented. An AIC defines how a set of integrated resource entities shall be used to fulfil a set of requirements. The intention was to create large reusable building blocks that could be used shared across many application protocols. While conceptually sound the fact that AICs are based on integrated resources meant that overlaps with existing application protocols was often not discovered until late in the development process. Consequently costly harmonisation activities had to be launched late in the development process of some application protocols [12].

The next approach in STEP was to develop a modularised approach where the modules are defined in both ARM and AIM. This approach forces an application protocol developer to consider the set of available modules before formalising domain specific requirements thus facilitate reuse of existing modules [125]. At the time of writing this thesis application modules are the preferred approach to develop application protocols within STEP. Compared with earlier approaches the advantage of the modules is a forced early harmonisation with existing STEP models. This is certainly beneficial for application protocol developers as the set of available modules highlights the rich set of models available in STEP.

However there is still an underlying clairvoyance assumption — all future data representation requirements can be met through the extension of currently available structures. Extending the scope of STEP to domains not currently covered could cause significant interoperability and harmonisation activities.

2.11 Alternatives to STEP

STEP is not the only framework for information modelling for product data exchange. A number of similar frameworks have been proposed and some have achieved high acceptance levels in industry and academia. The EDIF (Electronic Design Interchange Format) [78] and CDIF (Case Data Interface Format) [29] are frameworks defined for data exchange in specific domains. Of the two EDIF have won commercial acceptance as a data exchange format between engineering design tools. Later versions are implemented in EXPRESS. EDIF is, however, not a STEP application protocol.

CDIF has not been successful commercially but have had a large impact on the Meta-Object Facility (MOF) [13] used as a meta-modelling language for the Unified Modelling Language (UML) [130]. The latter framework has received a lot of interest recently, undoubtedly fuelled by the success of UML as the de-facto standard for specification of software systems. As with EXPRESS the MOF also incorporates support for defining textual constraints on modelling elements. UML and MOF is utilising XMI, a XML DTD, for exchange of UML specifications and provide a CORBA mapping for repository access. In addition OMG also provide the means for model standardisation. In these respects it is a compelling alternative to STEP especially for domains with close relations to UML and software engineering.

2.12 Summary

In this chapter an introduction to information modelling has been presented with emphasis on modelling for exchange of data. The risk of modifications to data due to mapping from one schema to another has been discussed and an analysis framework based on the quality of individual mapping functions presented.

The STEP framework and the EXPRESS language have been introduced as it is the modelling language used for the information model presented in this thesis. Some alternatives to STEP have also been discussed.

Chapter 3 Systems Engineering Data Representation

Systems Engineering is an engineering discipline for the development of complex systems. In this chapter an introduction to Systems Engineering is presented to provide an understanding of the scope of the discipline and restrictions made to the scope of the information model.

The introduction is necessarily short. Readers interested in more in depth information on Systems Engineering are referred to [24] [32] [57] [101] [139].

3.1 System — a Definition

At a first glance the word ‘system’ is anonymous. It seems to give little guidance in defining what is a system. However the word system in itself provides a good definition of the engineering domain. According to the Oxford English dictionary [141] a system is:

> “A group, set, or aggregate of things, natural or artificial, forming a connected or complex whole”[1]

The definition is very general, it encompasses natural systems and human-made systems of arbitrary size. Examples of *natural systems* range from immense as the universe in which we live in — to diminutive atoms.

The definition can be applied to the elements (components) of a system as well — systems are often composed of other systems. In a system hierarchy view the components of a system are called subsystems.

A characteristic of most *natural systems* is the high degree of order and equilibrium. The evolution of *natural systems* has been long and individual systems have had plenty of time to adapt to the changes in the environment. Material and energy flows are cyclic. There are no dead-ends and waste is recycled.

3.1.1 HUMAN-MADE SYSTEMS

Human-made systems are, when compared with *natural systems*, a relatively new phenomenon, but they have already had a fundamental and in many cases negative impact on the *natural systems* they are embedded in. In fact the impact of *human-made* systems today is so massive that the environmental equilibrium of the planet we inhabit is threatened.

3.2 Systems Engineering

This section presents a concise definition of Systems Engineering and the motivation for applying Systems Engineering concepts to development projects. There have been several attempts to define what Systems Engineering is in a single sentence, for instance in [32] [85] [57] [131]. The problem with finding a specific definition is that the heterogeneity of the systems being engineered today makes it virtually impossible to come up with a definition that satisfies all systems engineers. The very abstract definition in IEEE-P1220, Trial-Use Standard for Application and Manage-

1. Despite the definition, the word system is often paired with words like complex and heterogeneous to stress the problems associated with the development of systems.

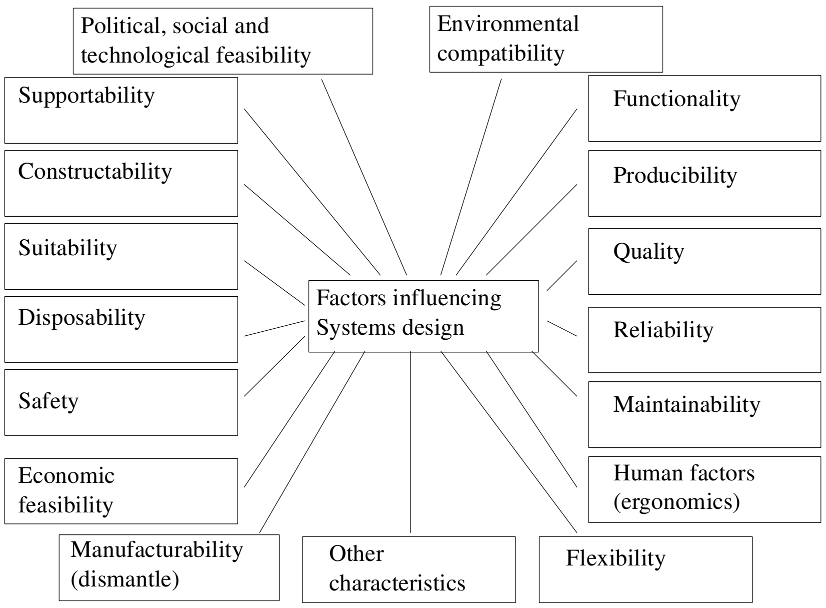


Figure 3.1: Factors influencing systems design (from [24]).

ment of the Systems Engineering Process [67] focus on the analysis aspects of Systems Engineering while failing to mention the factors constraining the realisation alternatives for system, i.e., time and resources.

> "An interdisciplinary collaborative approach to derive, evolve, and verify a life-cycle balanced system solution which satisfies customer expectations and meets public acceptability"

One of the central notions from the definition is that of system 'life-cycles'. The life-cycle of a system begins with the identification of a need for a system and extends over the design, development, production, deployment, support, operations and finally disposal of the system [101].

The objective to define a life-cycle balanced system implies that solutions with excellent characteristics for some life-cycles or aspects thereof shall be dropped if they display inadequate characteristics in other life-cycles.

The need for an interdisciplinary collaborative approach can be understood by the trend towards more and more complex systems. Many human-made systems are of such a high level of complexity and heterogeneity they cannot be effectively developed within a single design team or using methods from a single engineering discipline. Interdisciplinary analysis must be introduced to increase the understanding of what a system shall perform in its different life-cycles and how it shall be realised in the most efficient way, e.g., in terms of life-cycle cost, adverse environment impact or development time. Figure 3.1 illustrates an excerpt of the wide variety of factors influencing system design and underlines the importance of involving domain experts early in the system life-cycle.

Four issues characteristic for Systems Engineering can be identified (the structure of the list is adopted from [24]):

1. A top-down analysis approach that focus on the whole system as opposed to placing focus on its parts. The objective is to develop a balanced system that meets the stated requirements given constraints in terms of available resources and time constraints. This does not imply that the systems shall be built top-down. There may be constraints on the use of existing components which governs the capabilities of the overall system.
2. Life cycle orientation — Systems Engineering shall address all phases of a systems life cycle, from analysis and design though development, production, operation, maintenance and support to retirement, phase-out and disposal.
3. A focus on understanding and analysing the initial requirements on a system and relating requirements to design criteria and the follow-on analysis effort to validate the effectiveness of early decisions. By thorough analysis of the system a baseline, the set of relevant requirements, can be formed to guide the individual design teams involved in developing a system.

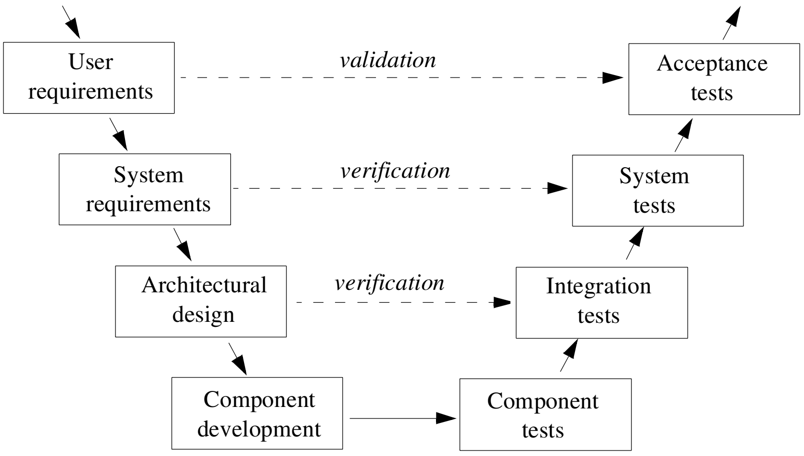


Figure 3.2: Part of the system life-cycle presented as a V-model

4. An emphasis on the interdisciplinary approach. A thorough understanding of all aspects of the system is required to ensure that all design objectives are met. One key practise to attain this objective is to involve specialists with different backgrounds early in the system life-cycle.

Systems Engineering is not tied to a specific engineering discipline as electrical or mechanical engineering. Instead Systems Engineering is applied to increase the effectiveness and quality of traditional engineering tasks. The objective is to analyse, define requirements for a system and partition the system into well-defined components whose functionality and performance are optimal with regard to the requirements on the overall system. Requirements in this context include all factors concerning a system such as function, performance, risk and budget. In many cases this implies components that can be easily integrated, upgraded or replaced within an evolving system. The output of this part of the Systems Engineering process is not in the form of production specifications (CAD drawings, program code) for physical or logical products, but specifications of systems and their components for all of its life-cycle phases.

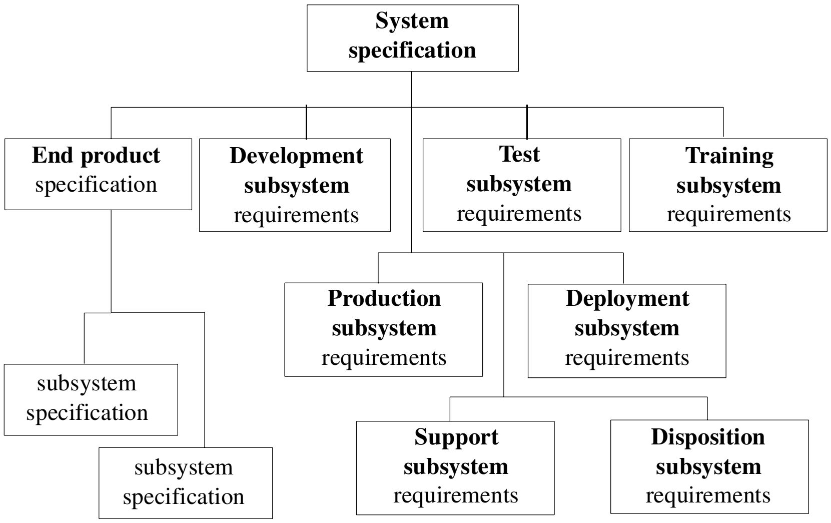

System specification
End product specification
Development subsystem requirements
Test subsystem requirements
Training subsystem requirements
Production subsystem requirements
Deployment subsystem requirements
subsystem specification
subsystem specification
Support subsystem requirements
Disposition subsystem requirements

Figure 3.3: System specification for multiple life-cycles [101]

The partitioning of requirements on a complex system into more manageable parts implies a future task to integrate the realised components into a system whose capabilities shall comply with the initial requirements. The definition of verification and validation criteria for a system is also important Systems Engineering activities. These aspects of Systems Engineering are often illustrated using a V-model as presented in Figure 3.2.t

In many cases the identified components of a system are so complex that they need to be treated as systems of their own right. In such cases the components are called subsystems of their parent system and the engineering process is applied to the subsystem as well, with the important constraint that the design space for a subsystem shall not violate that defined for the parent system.

Systems Engineering shall not only consider he end products of a system, i.e., its subsystems or components, but also the set of enabling products that support the system through its life-cycles, i.e., the development, production, test, training, deployment, support and disposition subsystems as indicated in Figure 3.3. Systems Engineering principles and process shall be applied to all subsystems identified for a system.

3.2.1 SYSTEMS ENGINEERING, A MOTIVATION

Systems are difficult to realise. Delays and increase in system cost are commonplace. One reason for this is that technical problems, like subsystem incompatibility are often discovered at a late stage in the development process. These problems require substantial extra effort to correct. Systems Engineering cannot prevent the appearance of these kind of problems. However, with the emphasis on a thorough interdisciplinary analysis of a system early in the development process it can be expected that problems will be fewer and less severe.

Figure 3.4 illustrates a number of important facts complicating engineering of complex systems. It is estimated that 50 - 75% of the projected system life cycle cost is already committed based on engineering design and management decisions made during the conceptual and preliminary design phase [24]. In these phases relatively little is known about the system. Later in the process, when more is known about a system it is relatively difficult to introduce modifications. A change in one component may have consequences to many other components with expensive modifications as a consequence. For optimal system development it is thus critical to attain a good understanding of the system already at the early stages of the development process.

3.2.2 APPLICATIONS OF SYSTEMS ENGINEERING

In the literature there are a number of suggestions on situations where Systems Engineering shall be applied. Systems Engineering is recommended if any of the following applies [24] [101]:

1. The system is complex.
2. The system is not available off the shelf.
3. The system requires special materials, services, techniques, or equipment for development, production, deployment, test, training, support, or disposal.

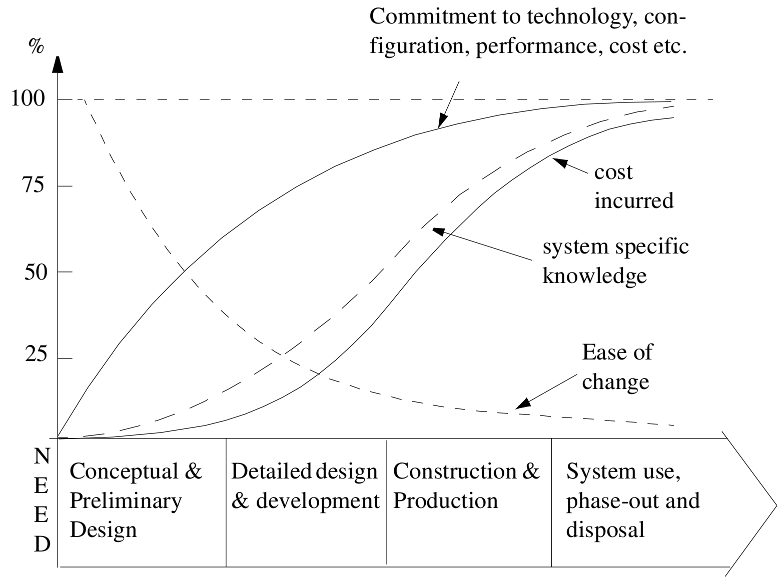


Figure 3.4: Commitment and ease of change (from [24])

4. The system cannot be designed entirely within one engineering discipline (e.g., mechanical design).
5. There are several suppliers involved in the design and development process.

Originally, Systems Engineering was developed and employed in large, mainly military, projects, e.g., the 'Manhattan project' and the US nuclear missile projects in the 1950'ies — Atlas, Titan and Minuteman [45]. Since then, Systems Engineering have been adopted by many other domains where system complexity is becoming a major concern, e.g., aerospace, telecommunication and automotive industries.

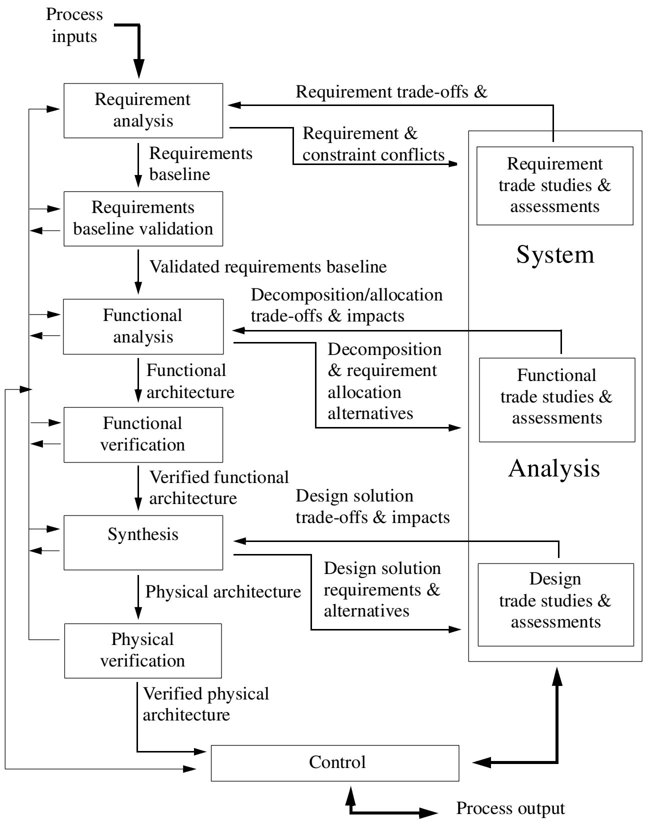

Process inputs
Requirement analysis
Requirement trade-offs &
Requirement & constraint conflicts
Requirements baseline
Requirement trade studies & assessments
Requirements baseline validation
System
Validated requirements baseline
Decomposition/allocation trade-offs & impacts
Functional analysis
Decomposition & requirement allocation alternatives
Functional trade studies & assessments
Functional architecture
Functional verification
Analysis
Verified functional architecture
Design solution trade-offs & impacts
Synthesis
Design solution requirements & alternatives
Design trade studies & assessments
Physical architecture
Physical verification
Verified physical architecture
Control
Process output

Figure 3.5: The IEEE-1220 systems engineering process

3.3 Systems Engineering Processes

This section provides a short overview on work on Systems Engineering processes. A large number of specialists contribute to the development of a complex system. In order to manage the contribution of individual specialists a coherent process shall be established. At its most minimal level the Systems Engineering process consist of the following activities [100]:

- Attain an understanding the problem before attempting to solve it.
- Examine many alternatives prior to selecting a solution.
- Check the work performed before advancing to new problems.

The items presented above are not specific to Systems Engineering, but basic characteristics of good engineering practice. They are emphasised in Systems Engineering as the complexity and heterogeneity of modern systems makes the problem of understanding, exploration of the problem and solution space as well as verification especially difficult.

Comprehensive Systems Engineering standards have been developed to guide Systems Engineering efforts. These documents contain definitions of activities that shall be undertaken to ensure that high quality systems are engineered. Examples of such standards include IEEE-1220 [126], EIA-632 [102] and ISO/IEC 15288 [14]. The value of these standards and other process definitions shall not be underestimated as they provide guidance to individual Systems Engineering organisations. In co-operation projects the application of a common process model could improve communication significantly.

Figure 3.5 illustrates the top-level activities of the IEEE-1220 process. The rigidity of the process depends on factors like system complexity and criticality, customer requirements and project organisation. The process is inherently iterative, several iterations are typically required to capture the complete system specification.

3.4 Systems Engineering Tool Data Exchange

Scenarios for Systems Engineering data exchange are identified by Carlsson [27] and by Schier et al. [134]. The seven scenarios identified by Schier et al. are presented below.

1. Tool migration, one time exchange from an old tool environment to a new.
2. Parent/child integration with different tools, data is entered in the parent tool, exchanged and refined in the child tool, then returned to the parent tool.

3. Peer to peer integration with different tools, data maintained at the same level of abstraction in multiple tool formats. The objective is to maintain specification consistency in multiple tools over time.
4. Tool to tool traceability with data from a tool managed in a second one for recording of traceability relationships and other information.
5. Data transformation/views, selection of a subset of the information stored in a tool for exchange to another tool.
6. Integrated CM process across tools. Configuration management information for design data is maintained independent of the tools where design data is created. This corresponds to a PDM system for management of product data.
7. Navigation of Systems Engineering database to user desired data, in this approach a systems engineer access a restricted set of data in a repository.

The scenarios listed indicate the diverse nature on tool data exchange. The only common denominator for all scenarios is the requirement for a common data representation for the exchange.

3.5 Identifying Systems Engineering Data

The objectives of and process for performing Systems Engineering has been presented in previous sections of this chapter. However, there is little guidance in the Systems Engineering literature on how to identify a finite set of concepts that Systems Engineering organisation wish to exchange.

3.5.1 PROCESS FOR DEFINING INFORMATION MODEL SCOPE

Three approaches have been followed in this research to answer the question: What data representations are in use in Systems Engineering in the Systems Engineering process?

1. Investigation of Systems Engineering data representation requirements in the European aerospace industry. The result of this investigation is presented in Section 3.5.3.
2. Investigation of methods recommended in Systems Engineering proc-

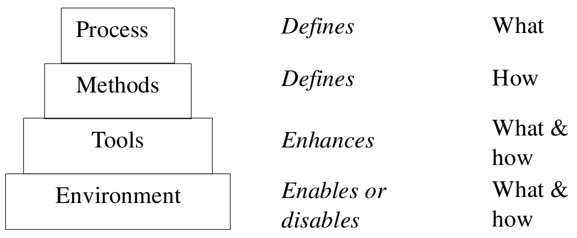


Figure 3.6: The PMTE paradigm pyramid

ess standards and other literature. The result of this investigation is presented in Section 3.6 and Section 3.7.

3. Investigation of typical professional roles (and the associated data set) performed by systems engineers.

The first two items was the primary and early source for identifying scope of the information model given the projects the research was carried out in.

As outside interest in the AP-233 work grew stronger and more organisations became involved it became apparent that individual stakeholders had mutually conflicting views on the activities and tasks of a systems engineer and consequently the associated data representations that should be supported by the information model. As a consequence item three was investigated to better understand the appropriateness of the restrictions made when the wide range of roles performed by systems engineers became apparent.

3.5.2 THE PMTE PARADIGM

The problem of identifying the scope of Systems Engineering data is illustrated by the PMTE paradigm [101] illustrated in Figure 3.6. A *Process* defines a logical sequence of tasks to be performed to achieve a particular objective. The process defines *what* task shall be performed, not *how* it shall be done. A *Method* consists of techniques that define *how* a task shall be performed. *Tools* are employed with the intention to enhance effectiveness of methods and the *Environment* is the set of factors that influence an engineer in his work.

A process is essentially independent of method. Two organisations may agree on the engineering process they employ, but use completely different methods to carry out the tasks in the process. Likewise organisations may agree on a common set of tools, but use them to carry out different tasks in the process. Consequently people from different organisations may agree on the activities performed in the process, but not on the data representations associated with each activity.

3.5.3 SYSTEMS ENGINEERING DATA REPRESENTATION REQUIREMENTS

An analysis identifying the initial set of requirements for the information model scope was performed in the SEDRES project by Barbeau et al. [19]. According to this document a suitable scope for a Systems Engineering data exchange information model is restricted to representations for capturing:

1. System identification — elements for identifying a system under specification, its life-cycles and composition structures.
2. Specification elements — defining what a system shall conform to using abstract, technology independent representations.
3. Process reference — capturing the set of activities that a specification was developed in and why specific alternatives where selected. The intention is not capture an idealised process, but the actual process as occurred in the development of a system.
4. Configuration management — capturing how the definition of individual specification and system elements evolve over time and its relationship to other elements.
5. Administrative information — capturing the person and organisation involved in activities and authorship, as well as element approval information

Items 4 and 5 above represent support information that can be applied to items 1 - 3.

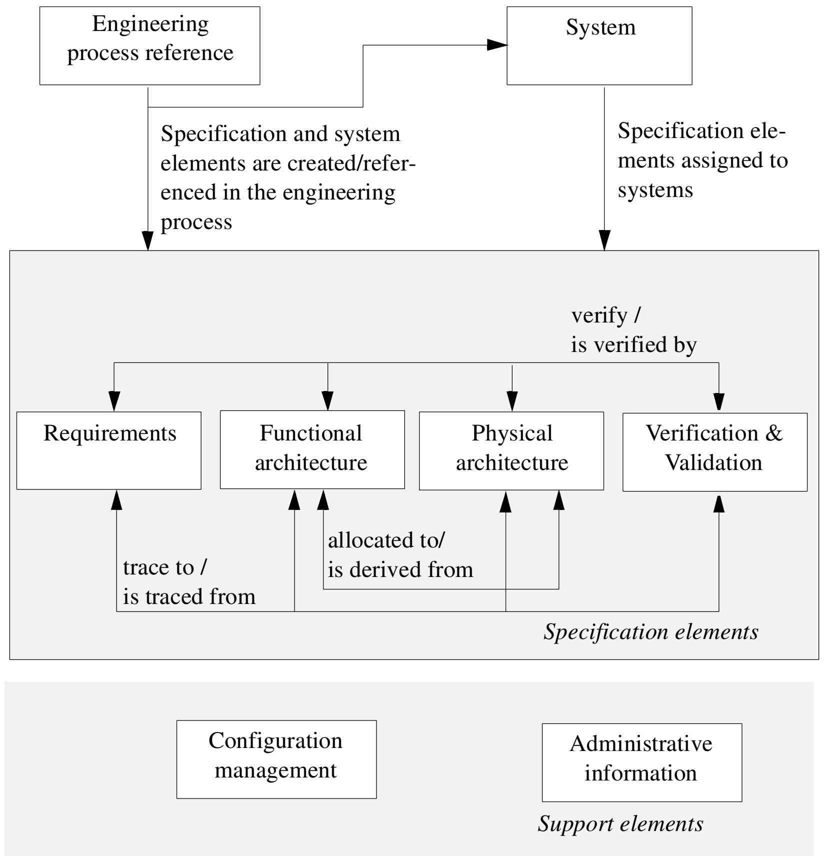

Engineering process reference
System
Specification and system elements are created/referenced in the engineering process
Specification elements assigned to systems
verify / is verified by
Requirements
Functional architecture
Physical architecture
Verification & Validation
trace to / is traced from
allocated to/ is derived from
Specification elements
Configuration management
Administrative information
Support elements

Figure 3.7: Information model scope

Specification elements (item 2 above) can be further classified into statements of what a system shall comply with in the form of:

- Requirement, constraint and verification statements.
- System functional architecture definition.
- System physical architecture definition.
- Verification and validation definitions at the same level of abstraction as the requirements.

In addition the information model shall contain elements for establishing relationship between different specification elements such that:

- Requirements can be traced to and from functional and physical elements to indicate that a requirement statement is related to that element,
- Verification and validation definition can be traced to requirements, functional architecture elements and physical architecture elements, and
- Functional elements can be allocated to physical elements.

The information model scope identified for the SEDRES project is illustrated in Figure 3.7.

3.5.4 CONFIGURATION MANAGEMENT INFORMATION

Representation of version and configuration management information is given a high priority within the information model. It is anticipated that a specification will go through a number of configurations in its life-cycle and it is the objective that it shall be possible to capture each individual configuration within the information model. Moreover it is anticipated that there will be a large degree of commonality between individual configurations of any specification. Storage of redundant data elements shall be avoided in order to facilitate the identification of a degree of commonality between configurations.

3.6 Information Model Scope vs. Systems Engineering Literature

The selected scope for the information model appears realistic as it is in line with the understanding of Systems Engineering data presented in information models by Buede [25], Mar [100] and Jackson [71]. A slightly wider scope is presented by Oliver [114] who also includes project management and planning information.

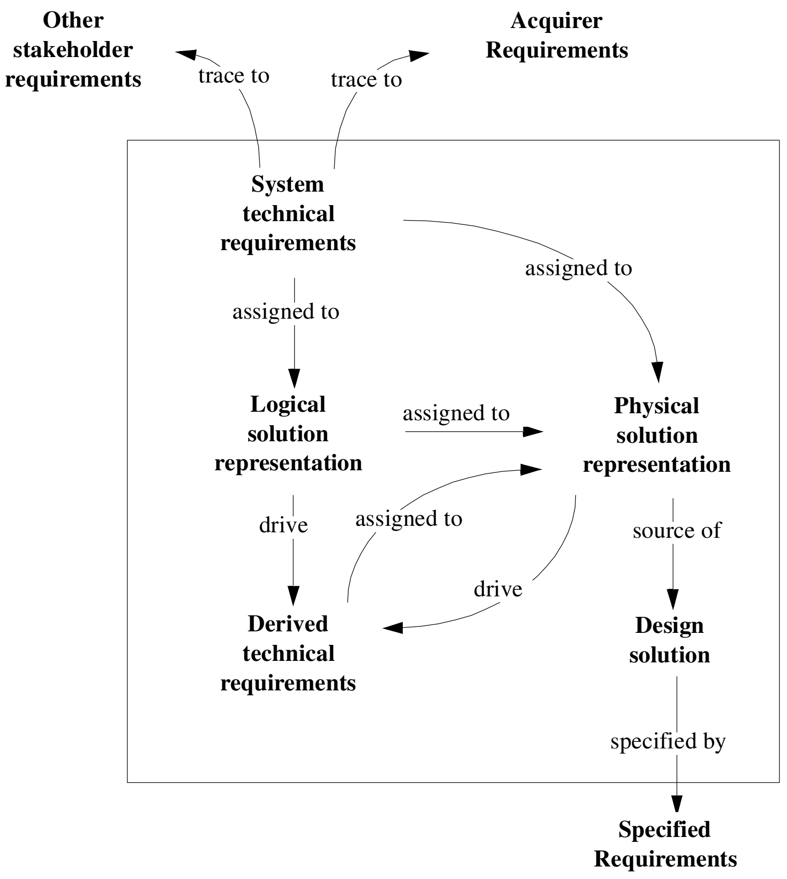

Other stakeholder requirements
trace to
trace to
Acquirer Requirements
System technical requirements
assigned to
assigned to
Logical solution representation
assigned to
Physical solution representation
drive
assigned to
source of
drive
Derived technical requirements
Design solution
specified by
Specified Requirements

Figure 3.8: Requirement relationships in the system design process according to EIA-632

Recently, object oriented techniques have been proposed as an alternative to system functional architecture models [15] [34] [111]. While we acknowledge the qualities of object oriented modelling techniques, especially for software intensive systems, we have not included these techniques in the scope of the information model presented in this thesis. A study on the prospects for integrating functional and object-oriented systems modelling is presented by Pandikow [117].

3.7 Information Model Scope vs. Systems Engineering Process Standards

When the structure of the IEEE-1220 (presented in Figure 3.5) is compared with that of the information model there is close correspondence. There is also a strong correlation to the requirement relationship view identified for the system design processes in EIA-632 [102] in Figure 3.8.

In the terminology used in EIA-632 the *logical solution representation* correspond the system functional architecture and the *physical solution* representation correspond to the system physical architecture.

However, the process standards do not give detailed recommendations on *how* the work shall be performed or which methods shall be employed and how the information generated in the process shall be represented. The complexity and heterogeneity of a system and the knowledge and experience of the development organisation and its organisational culture decide the methods and tools used in the development process. An activity can be performed to different levels of granularity depending on the complexity of the system and the selected development strategy. In some cases a requirement identified early in the Systems Engineering process may be recorded in plain text, while later in the development process requirements may be captured in a detailed CAD model.

Consequently, a process description is decoupled from the representation of data generated in the process. Likewise, it is not realistic to deduce the suitability of the selected scope of the information model representations from any process model.

3.8 Systems Engineering Roles and Information Model Scope

In this section the selected information model scope is compared against identified Systems Engineering roles. Later in this section information model scope is mapped onto the identified roles to illustrate the extent of the information model.

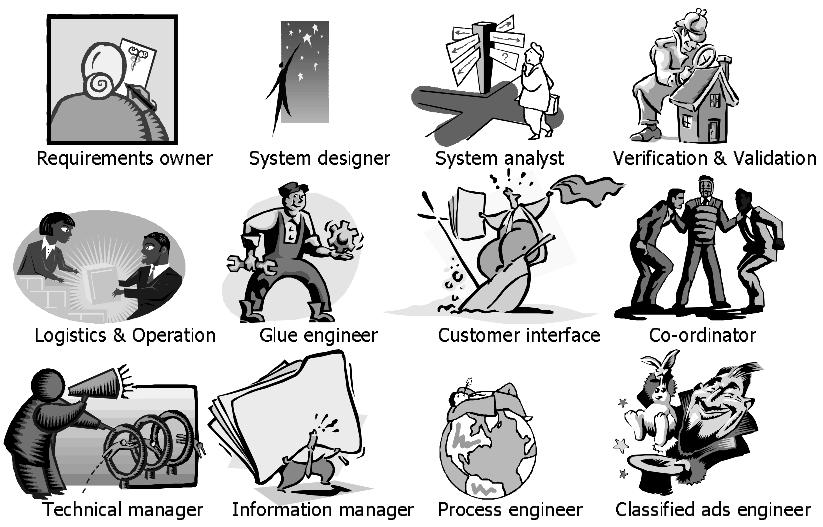


Figure 3.9: Systems engineering roles identified by Sheard [137]

3.8.1 SYSTEMS ENGINEERING ROLES

Sheard [137] and Mar [100] identifies the widely different tasks, which may be assigned to systems engineers in different organisations. One motivation for these papers was to identify the tasks and responsibilities held by people with *Systems Engineering* in their job title, which fall in the scope and interest of the International Council on Systems Engineering (INCOSE). As with any abstractions the result is a grossly simplified view of the tasks of a systems engineer, but it is still a valid illustration of the wide span of activities performed by systems engineers. The twelve roles identified by Sheard are introduced below and in Figure 3.9.

Requirements owner: Identifies system and subsystem requirements from customer needs. The tasks include translating customer needs into specific well-written requirements on systems and components. The focus is on understanding system interfaces and that the system functional architecture correctly captures the need of the customer.

System designer: An engineer in this role creates a high-level system architecture and design and select major components. Possible solutions are compared against requirements. Requirements for the subsystems are described in detail and subsystem specifications are verified. Because of system complexity the emphasis is mostly on architecture, high-level design integration and verification. There is a substantial overlap with the roles of the requirement owner role.

System analyst: Confirms that the system design will meet the requirements. Analyses typically include system properties like weight, power, throughput, availability and reliability. Analyses may be performed at a conceptual or product data level depending on the complexity of the system, the development process phase and the desired analysis fidelity.

Validation/Verification engineer: Implements the system verification and validation programme to ensure that the system developed is compliant with the requirements and customer expectations. Verification and validation engineers may also develop the detailed system test plan and procedures. Engineers in this group need to have a good understanding of the requirements on the system and its intended behaviour.

Logistics & Operations engineers: An engineer in this role captures the back end of the system life cycle. It includes on-call answers to questions and resolution of anomalies. In addition engineers in this role is usually expected to bring maintenance, operations, logistics and disposal concerns to the early phases in the system life-cycle.

Glue engineer: In this role a systems engineer serves as a proactive trouble-shooter, looking for problems and taking measures from preventing them from happening. Much of the focus of this role is on system interfaces and on making sure that systems do not interfere with each other. The problems faced by systems engineers in this role range from conceptual and domain independent to very detailed and domain specific. To fulfil this role an engineer need wide experience and continuous learning to keep ahead of the problems.

Customer interface: In this role a systems engineer may be tasked to represent the point of view of a customer to ensure that it is properly respected throughout a project. Another task is to interact with the customer to ensure that the right system is built. From a data representation perspective there is a substantial overlap with the requirements owner and system designer roles.

Technical manager: This role emphasizes the management aspects of Systems Engineering. The responsibilities include project planning, controlling cost, resource allocation and scheduling, maintenance of support groups like configuration management and computer support.

Information manager: As information and support systems become more complex and more pervasive it becomes more important for someone to identify the overall information needs of a system, and even of business. This role may include data (configuration) management and process asset management.

Process engineer: A Systems Engineering role which emphasize the importance of system documentation, engineering and organisational process improvement issues for future projects. Another task in this role is the definition of metrics against which evaluations of process effectiveness are performed.

Co-ordinator: Because of their broad view systems engineers are sometimes asked to coordinate development groups and resolve system issues to seek consensus or to make recommendations. Skills in facilitating discussions and maintaining a good discussion climate are essential in this role.

Classified Ads systems engineer: This role is included to capture all definitions of Systems Engineering that is not related to the design, analysis and development of complex heterogeneous systems. In this role systems engineers may have applied software engineering tasks or computer maintenance and administration tasks, e.g., computer network system administrators.

3.8.2 SYSTEMS ENGINEERING ROLES VS. INFORMATION MODEL SCOPE

The roles presented in the previous section illustrate the highly heterogeneous world of Systems Engineering. The information model presented in this thesis only covers a small subset of the data representation requirements associated with the roles.

The scope of the information model correspond roughly to the following four Systems Engineering roles:

- Requirements owner
- System designer
- System analyst
- Validation and verification

The extent of support for each role varies. System analysis and validation and verification are only supported for the early phases of the Systems Engineering process. Detailed, domain specific, representations are not supported. The scope of the information model related to all Systems Engineering roles is illustrated in Figure 3.10.

Motivations for not specifically including support in the information model for the remaining eight roles defined by Sheard are presented below. Note, that though many of the roles are not explicitly in scope of the information model they are supported in part by the fact that systems engineers in these roles may use information common to the four roles identified to be the prime focus of the information model. For some roles there is also explicit support in existing STEP applications protocols.

- Logistics and operations: Sheard defines two tasks in this role. The contribution to the requirements owner, system design and system analyst roles are using the same information representation as in those roles. The aspect of the role contributing to the operation and maintenance phase of a system is in itself very complex and the subject of the PLCS (Product Life Cycle Support) project within STEP. The scope of the PLCS project is described in [43].
- Glue engineers: This role is a core Systems Engineering role but not included in the information model scope as a glue engineering is pri-

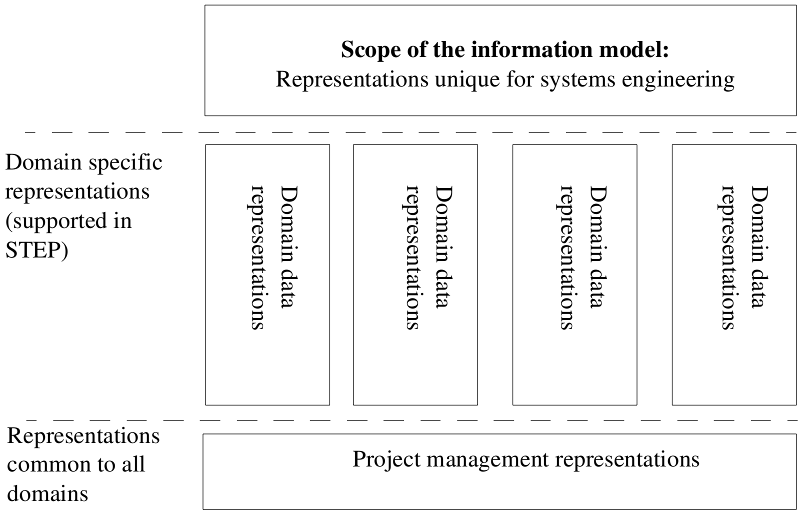

Scope of the information model:
Representations unique for systems engineering
Domain specific representations (supported in STEP)
Domain data representations
Domain data representations
Domain data representations
Domain data representations
Representations common to all domains
Project management representations

Figure 3.10: Scope of a systems engineering information model

marily operating on domain specific representations. There is extensive support for engineering domain data in existing STEP application protocols.

- Customer interface: From a data representation perspective there are no unique data representation requirements. The interaction with a customer will use the representation appropriate for the problem at hand. This may be the representations used by the requirements owner, system designer, system analyst or validation and verification roles or any domain information representation.
- Technical manager: Management and resource planning is a key element of Systems Engineering. But it is not a problem that is unique to Systems Engineering. Rather it is a domain shared with all engineering disciplines. Moreover, project management aspects are identified being outside the scope of STEP in the official STEP framework definition document [70]. Consequently, this aspect of Systems Engineer-

ing was not actively considered in the information model development process.

- Information manager: As with the glue engineer role this role may create and communicate information in any representation, including the representations found in scope of the information model or any domain model within the STEP framework.
- Co-ordinator: Systems engineers in this role are likely to handle diverse information. Some may be in scope of the information model, some may be in domain information models.
- Process engineer: Systems engineers in this role need to consider all design data produced and referenced in the engineering process. However, there is no specific data associated with the role.
- Classified ads engineer: In this role a systems engineer is operating on engineering domain specific data. Hence the same motivation as for the Glue Systems Engineering role applies. For engineering domain data there is plenty of support in existing STEP application protocols.

3.9 Systems Engineering Methods and Tools

A wide variety of tools that support systems engineers in different phases of the Systems Engineering process are in use in industry [35]. From the perspective of this thesis the tools of interest are computer based specification, analysis and verification tools. In this category two classes of tools can be identified:

- General Systems Engineering tools, supporting a large number of activities of the Systems Engineering process.
- Specialist tools, supporting specific activities in the Systems Engineering process.

Examples of the first category are tools like RDD-100 by Holagent [64] and Core by Vitech Corporation [151]. In the second category there are tools like Statemate by I-logix [66]. Typically tools in the first category pay special attention to supporting the process but may be less detailed than in individual areas compared with specialised stand-alone tools.

3.10 Method Selection Criteria

Deciding whether a method or tool is a relevant Systems Engineering method or tool is highly subjective. From an organisations point of view a method may be a core Systems Engineering method, while from the point of view of other organisations the same method may be irrelevant. No matter how much the scope of the information model is extended there will always be methods, considered to be relevant to Systems Engineering by some organisations that is not adequately supported.

The approach selected to determine the whether a method is relevant to Systems Engineering was to consult a reference group of practitioners within the organisations involved in review and validation of the model as well as literature studies. Initially the members from this reference group came from partners in the SEDRES projects, but were later complemented with representatives from INCOSE and the ISO AP-233 working group. It is hoped that the opinions collected in this process are sufficiently complete to satisfy a reasonable number of systems engineers.

In many cases the decisions for determining the scope of the model are arbitrary. For instance, the Extended Functional Flow Block Diagram modelling language [95] was considered a core Systems Engineering language by the reference group, while related languages such as Lotos and SDL [148] were not.

3.11 Summary

Systems Engineering as a domain encompasses many activities and there are many roles for systems engineers. In this chapter we have presented a broad overview of Systems Engineering, the process for selecting the scope of the information model and have outlined the scope of the information model. The impact of the restrictions made has been presented and been related to the scope and structure of STEP and existing STEP application protocols.

PART II

Information modeling

Chapter 4 Information Modelling Principles

This chapter presents the modelling philosophy applied for the information model. The information modelling requirements, and the principles that guided the development of the information model is presented. Detailed information on the constructs of the information model is presented in chapters 6 to 11.

4.1 Context

Information modelling is about formalizing information in an unambiguous way, or as put in [133]:

> "An information model is a formal description of types of ideas, facts and processes which together form a portion of interest of the real world and which provides an explicit set of interpretation rules"

In this thesis the portion of the real world of interest is the product data captured in system specifications as outlined in Chapter 3. More specifically the focus is product data as captured by Systems Engineering tools.

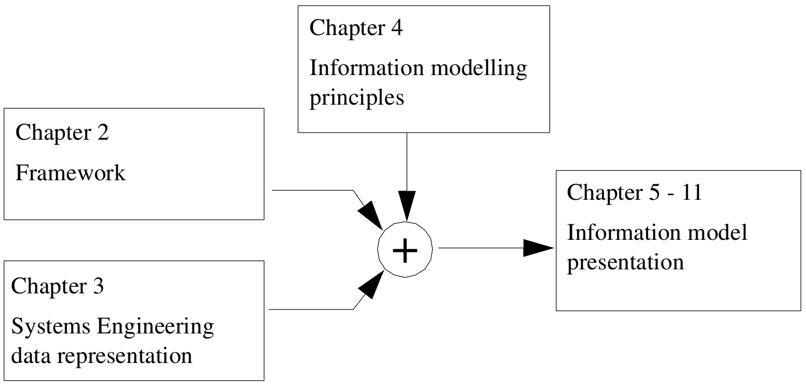

Chapter 4
Information modelling principles
Chapter 2
Framework
Chapter 3
Systems Engineering data representation
+
Chapter 5 - 11
Information model presentation

Figure 4.1: Thesis organisation

The relationship between earlier chapters of the thesis and the content of this and later chapters are illustrated in Figure 4.1 and presented further below. Methods for and issues in information modelling and tool data exchange has been presented in Chapter 2 and the scope of the domain have been introduced and discussed in Chapter 3. This chapter introduces the non-functional requirements, representing the development philosophy, that have driven the development of the information model. Further on in the thesis there are detailed presentations on key aspects of the information model, its structure and semantics including comparisons with methods defined in literature or as instantiated in Systems Engineering tools.

4.2 Terminology

This section presents a terminology framework for definition of the information model requirements presented in Section 4.3. The relationship between the terms are presented in EXPRESS-G notation in Figure 4.2.

A computer based Systems Engineering *tool* support an identified *method*. Each method is supporting one or more *concepts* that each has a defined *semantics*. Concept semantics may be explicitly defined or be implicit

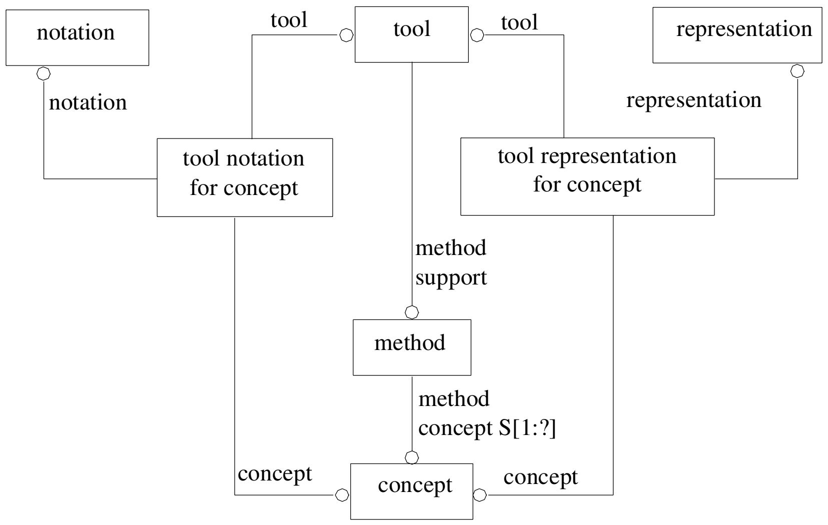


Figure 4.2: terminology framework

(inferred through a tool implementation). It is not uncommon that multiple methods use homonyms and synonyms for concept identification, c.f., the discussion schema and semantic heterogeneity in Section 2.4.

A tool uses a specific *notation* to present a concept to a user. The notation may be graphic, symbolic or textual. A *representation* is a subset of a data storage schema defined to capture a concept within a tool database. For each tool there exist an unambiguous mapping between the tool notation and the tool representation.

4.3 Information Modelling Requirements

This section presents seven basic non-functional requirements that have guided the development of the information model.

The task of defining an information model for data exchange for a domain where so many heterogeneous methods are in use as in Systems Engineering is fundamentally different from defining a data model for representing engineering method specific data. The aim with the information model has been to avoid defining new Systems Engineering and analysis methods. Instead the model shall allow the exchange of specifications captured by Systems Engineering tools currently in use.

The order in which the requirements are presented in below does not imply importance.

4.3.1 PROCESS AND METHOD INDEPENDENCE

In such a diverse domain as Systems Engineering it is important to avoid encoding specific process and method constraints in the information model.

> **Requirement 1:** The information model shall be method and process independent.

The rationale for this requirement is that the information model shall not dictate what is a good Systems Engineering process or method. There shall neither be any preference for specific processes or method concepts. In cases where multiple relevant method concepts are identified then the information model shall be fair in the support for all.

4.3.2 COMPLETENESS ASSUMPTION

Just as the information model shall be fair in process, method and tool support there shall be no assumed threshold for specification completeness encoded.

> **Requirement 2:** No assumption on the level of completeness of a specification involved in a data exchange shall be encoded in the information model.

It shall be possible to map tool data to the information model regardless whether the specification is complete or not from a engineering point of view. This requirement is justified by the fact that it is not meaningful to make any prior definition on the structure and content of a ‘complete’ specification both for individual elements in the specification and for the complete specification. The concepts employed to represent parts of a specification differ with organisation and methods used. Moreover, similar concepts may be captured at different levels of granularity in tools. Mandatory concepts in one tool may not be applicable in another. From a data exchange point of view it is not possible to judge a certain system specification structure as being more complete or better than another. The specification structure and content is simply a result of the processes, methods and tools in use at a particular organisation. Specification completeness can only be determined through analysis where organisation and project specific criteria are taken into account.

There are also cases where data exchange or storage of known incomplete models is beneficial. For instance, an incomplete specification may be exchanged in order to generate early feedback.

Specification ambiguity is another aspect of completeness. In its final version a specification can be expected to be without *known* ambiguities. But intermediate revisions may well contain conscious, undetected or not yet corrected ambiguities. For instance, it may be the case that in a revision of a system specification the weight of a component may not be equal to the sum of the weights of its immediate child components. While this may be suspicious it may have been made on purpose to indicate a known problem with overweight, or in case of underweight to allow for a safety margin in the case that some component may come out heavier than specified. Consequently the presence of ambiguities shall not prevent the mapping of a specification onto the constructs defined in the information model. However, the information model representation shall be formal enough to allow for consistency checking to be performed on the structures defined by the information model.

4.3.3 CONTEXT INDEPENDENCE

The Systems Engineering process typically generates multiple alternate designs. It is desirable to keep track of how elements in the specification evolve over time. This includes both elements that change and elements that stay the same in different versions of a system specification.

Requirement 3: The information model representation shall be such that key concepts can be used independently in multiple system specifications.

For Systems Engineering specification elements this implies that an object representing a requirement may be included in multiple systems specifications and in each specification it may be handled differently, e.g., be assigned to different object or be assigned different properties.

The required capability can be used for answering queries like: "Which systems share a particular functionality", "Which versions of a system share a specific requirement" or "How was this particular requirement handled in systems X and Y". Similar requirements are expressed in [41] [99]. This requirement is significant when data representation of multiple versions of a system specification and/or multiple systems is considered.

4.3.4 INFORMATION MODEL DETAIL

The semantics of the concepts supported by the information model and the underlying representations shall be unambiguous, or there is substantial risk that specifications exchanged are modified in the exchange process. If such modifications are not detected the consequences for a development project may be catastrophic.

Obviously, tool data exchange may introduce additional risk for specification misinterpretations, as there is a risk that information is lost or modified when data is transformed from one representation to another. Modifications to a specification may occur both in the mapping from tool specific concepts to the entities in the information model (data export) and from information model back to tool specific concepts (data import). In Figure 4.3 *f(s)* illustrate the data export and *g(s)* represent data import. See Section 2.4 for a detailed discussion on mapping function characterisation.

Modifications to exchanged data always occur when a specification is imported into a tool whose method does not support the same concepts as the method of the original tool. This kind of tool capability induced modifications can be accepted as long as the data exchange mechanism is detailed enough for detecting the extent of the modifications and there are

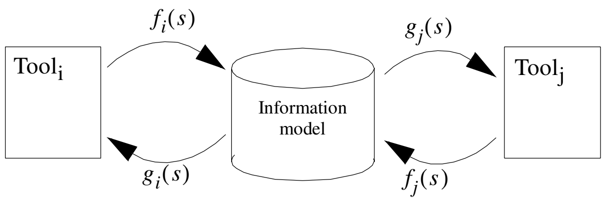


Figure 4.3: Information model mapping functions

means for informing stakeholders in the data exchange process about the modifications.

This leads to the following two requirements on the information model:

> **Requirement 4:** The semantics of individual information model entities shall be detailed enough to determine whether the semantics of a method specific concept is maintained in the information model.

> **Requirement 5:** The information model shall be detailed enough to allow for identification of specification concepts not correctly conveyed from a source to a destination tool via the representation in the information model.

Meeting these requirements are essential if the information model shall be effective and reliable for data exchange.

Identification of modifications incurred to a specification in a data exchange is simplified if there is a common basic representation for each family of related concepts with modifiers that extend the semantics of each basic concept. This approach removes redundant representations for similar concepts and simplifies the traversal and interpretation of data mapped on to the representational elements of the information model.

For each individual concept it is sufficient that there exist unambiguous forward and inverse mapping functions that maintain concept semantics. This may result in the definition of complex mapping functions between a tool and the information model representation. Yet, we believe that a common core for each concept is preferable over a situation where each tool exports data in their proprietary representations.

4.3.5 PRESERVATION OF SPECIFICATION STRUCTURE

It is not sufficient just to preserve the semantics of a specification when exchanging data between tools. For traceability and human interpretation purposes it is important to preserve the structure of the original specification. For instance, it is not acceptable that a Mealy finite state machine is represented as an equivalent Moore finite state machine in the information model as the original machine cannot be recreated from the representation.

Requirement 6: The information model shall be constructed such that the representations in the information model shall rich enough to maintain the structure of a specification.

The requirement is not in conflict with requirements 4 and 5 above. Method specific constructs are only accepted *iff* they are necessary to maintain the structure of the original specification.

In Figure 4.3 $f_i(s)$ and $g_i(s)$ define the set of mapping functions from a tool specific representation to the information model and vice versa. $f_i(s)$ is the set of mapping functions for a specification s expressed in a tool representation i to the information model, $g_i(s)$ is the set of mapping function for a specification s from the information model to tool representation i. The structure preservation requirement can be expressed for a specification S can be formalised as:

$$S =_S g_i(f_i(S))$$

Where $=_S$ denotes semantically and structurally equivalent representations. In other words, mapping a specification from a Tool_i onto the information model and back to Tool_i shall result in a specification whose semantics and structure is equivalent to the original one.

4.3.6 PRESERVATION OF SPECIFICATION LAYOUT

Specification layout is important for the readability of a Systems Engineering model. A well thought out model layout simplifies interpretation of a model significantly. Preservation of layout information is of high importance when complex specifications are transferred between organisations. If layout is not preserved in a data exchange it will be very difficult for engineers at different locations to discuss the content of a specification, unless time consuming layout restoration activities are undertaken.

> **Requirement 7:** The information model shall provide the capability to represent the layout of a specification.

Layout information shall neither convey any semantics nor be notation specific. The sole purpose is to allow for recreation of the general layout information from the original tool.

At the same time the existence of layout information in a data exchange is not mandated. Specification correctness does not depend on the availability of layout information.

4.4 What can be Standardised?

After analysing the information modelling requirements it is important to ask the question:

> What can be standardised in a data exchange information model?

In the context of this thesis an answer can be derived through an analysis of requirements 1, 2, 4, 5 and 6. Requirement 1 states that the information model shall be process and method independent. Requirement 2 states that there shall be no completeness threshold for storing data onto the structure defined in the model, i.e., a user shall never be prompted to fill in additional information on an object — information that may not be possible to specify in the particular tool used. Requirements 4 and 5 emphasize detail and semantics for all relevant concepts and requirement 6 implies the

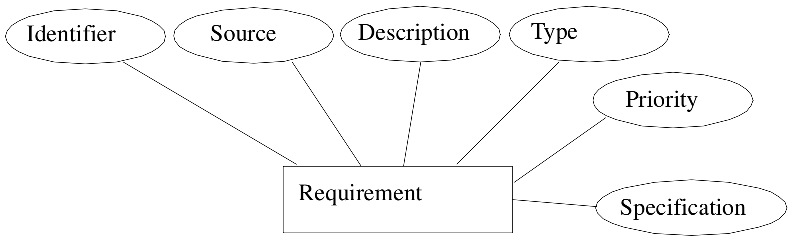


Figure 4.4: Information model structure for requirement

existence of forward and inverse mapping functions for tool specific representations to the information model that preserve the structure of the specification.

If the information model shall support the concepts found in multiple Systems Engineering methods then there must exist a common set of entities with a well-defined semantics that is agreed upon in all methods. It is assumed that this core set is finite for the Systems Engineering domain — under the restrictions made when the model scope was defined.

The next question is whether a finite set of attributes could be defined for each core entity identified. A potential approach to defining a model would be to try to enumerate all potential variants or properties of an entity and harmonise this set. This approach is feasible in cases where a small number of variants acceptable to all stakeholders can be identified. However, the problems associated with this approach is illustrated by the small information model fragment presented in Figure 4.4 taken from the book Requirements Engineering by Kotonya and Sommerville [84]

The example is trivial, but holds for entities of any complexity. The model in Figure 4.4 specifies that a requirement is characterised by an identifier, a source definition identifying the owner of the requirement, a descriptive text, a classification according to type of requirement, a definition of priority and the actual specification of what is required. The model prescribes a comprehensive data set for representing requirements. However, if used for data exchange under the requirement identified earlier in this chapter some severe problems can be identified based on the following two questions:

1. Does the set of attributes adequately cover the data representational needs of an advanced requirements management method? The answer is no — additional attributes can always be added to capture organisation, method and process specific aspects. If more attributes are added to the model then there is the risk there may be attribute ambiguities.
2. Does the set of defined attributes support the minimal data representational needs of a basic requirements management method? Again the answer is no, a basic tool may not provide data for the identified attributes. In the example above it should be noted that there are basic tools where requirements classification or prioritisation are not considered.

Items 1 and 2 above are obviously in conflict with each other. A coarse granularity view prescribed under item 2 cannot effectively be combined with a fine granularity view prescribed by item 1 as long as the approach is based on explicit identification of entity attributes.

The approach selected for the information model is to define a minimal number of attributes for each core entity in the model and provide support for assignment of any number of method and process specific properties, e.g., authorship and ownership information, comments, prioritisation, status, the representation of physical and functional characteristics.

No constraints are defined and there are no explicit guidelines that define the set of properties a *well-specified* specification element shall possess. Consequently, the information model is not normative in the sense it defines a fixed set of attributes for each entity. Instead, method and process harmonisation is a prerequisite for organisations involved in Systems Engineering tool data exchange. This is in line with requirement 2 presented above. Likewise, à priori determination of the appropriateness of a particular property of an entity is not possible. Determination of completeness and soundness of a specification must be performed on a case by case basis under the constraints defined by individual projects.

4.5 Alternate Approaches

The view on specification completeness taken in this thesis coincide with that taken for STEP application protocols but is in contrast with that taken in, e.g., the RQML data exchange information model for textual requirements [56]. In RQML complete (complete with regard to requirement engineering best practise) and fixed set of attributes are defined for each entity. Default values are defined for each attribute in case no value is supplied from a tool.

While this approach guarantees that a specification is complete with regard to attributes defined in the information model the result does not necessary reflect the intention of a user. For instance, in RQML a requirement is assigned a specific priority level if the user supplies no priority value. However, it is not necessarily the case that the assigned priority corresponds with the intention of the user in charge of entering the requirements. Such subtle modifications to a specification may convey an incorrect view of the intent behind the specification.

4.6 Summary

In this chapter the guidelines applied in the information model development process has been presented. These guidelines were identified and used in order to allow for flexible, yet well-defined instantiation combinations for the information model. The chapter also discusses consequences of different approaches to standardisation and highlights the potential risks with the addition of data beyond what is captured in a source tool in the exchanged data.

Chapter 5 Information Model Overview & Structure

This chapter presents an overview of the scope and structure of the information model. The purpose is to present an overall picture of the model architecture before the detailed presentation of model elements in chapters 6-11. The chapter also contains an analysis on how the basic structures in the STEP data architecture can be adapted to comply with Systems Engineering data representation requirements. The consequences of each alternative on the overall capabilities of the information model are discussed and the selected architecture is presented.

5.1 Information Model Overview and Scope

The information model presented in this thesis corresponds to an Application Reference Model (ARM) in the STEP terminology. This implies that representation structures are captured using terminology used in the domain and no mapping to the STEP integrated resources has been performed.

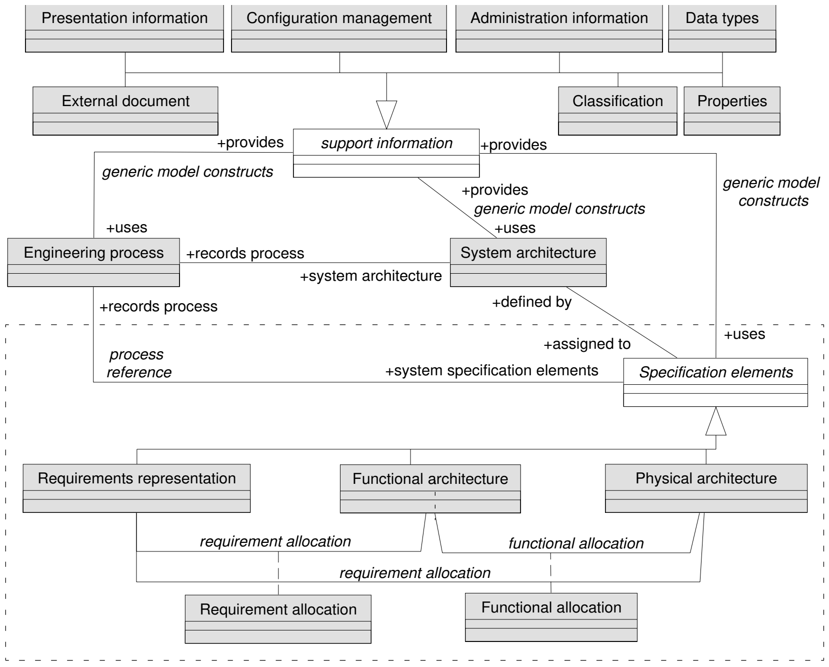


Figure 5.1: Information model conceptual view

The scope of the information model is illustrated in Figure 5.1 using UML syntax. In the figure each UML class symbol represents a group of related entities. The classes enclosed in the dashed rectangle represent the supported core Systems Engineering data and those outside represent generic support concepts that in many cases are shared with other STEP Application protocols. Transparent class symbols represent generalisations inserted to improve the presentation of the model. These classes do not have a counterpart in the information model.

The main groups of the model are:

- System architecture, representing the building blocks for grouping all information valid for a system, a stakeholder or life-cycle view on a system or system interface. There is also support for representing system composition structures.

- Specification elements, an abstraction introduced to cover common specification techniques, including requirements, functional architecture and physical architecture as well as verification and validation data. The specification element group of the information model is outlined in more detail in Section 5.1.1.
- Requirement and functional allocation, defining the mechanisms for maintaining traceability within and across views on a system. There is support for tracing requirements to individual elements in the functional and physical architecture and for allocating functional architecture elements to physical architecture elements respectively. Additionally there is support for relating verification and validation data to the elements they relate applied to.
- Engineering process, defining the building blocks for representing activities in the Systems Engineering process and associating specification information to the activities they originate/relate to. The entities for capturing engineering process can be used for capture of design and analysis activities, as well as design and trade-off decision. Entities for representing change proposals, change impact analyses and change orders are also included in this group.
- Support information, representing the building blocks for representing supplemental information. This large group is an abstraction of groups for representing configuration management information, visual layout information, and mechanisms for referencing physical or electronic documents or other artefacts of relevance to a system specification, administrative information (e.g., element ownership and specification element approval), structures for classification of specification elements, data types for simulation purposes and properties for specification elements.

The relationships between the groups are as follows.

- *Specification elements* objects define the content of *System architecture* objects.
- *Engineering process* objects captures the process in which *Specification elements* and *System architecture* objects are created and referenced in.
- The *Support information* group includes entities that provides general information that can be related to a large set of elements is applicable to entities in the other groups.

5.1.1 SPECIFICATION ELEMENTS

The group of entities for representing requirements, functional and physical architecture of a system is at the heart of the information model. The capabilities of individual groups are outlined in below:

Requirements representation

The requirements representation group contains entities for representing requirements (individual statements of capabilities a system shall conform with) in text, in structured formats or in functional, physical architecture models or external documents. There is support for representing requirement composition structures, requirement classification and for capturing relationships between individual requirements. There are also entities in the requirement representation group for capturing system verification and validation data and the fulfilment and results of activities carried out.

Functional architecture

The functional architecture group contain entities for representing a function composition structure, and different semantic variants of functional interaction. There is support for representing continuous and sampled systems. Functional behaviour may be expressed in, e.g., data flow [5], structured analysis [61] or behaviour diagram [95] notations.

Physical architecture

The physical architecture group contain entities for representing an abstract view on the physical components of a system. Individual components may be related to each other and the interface of each component is defined. Detailed component properties such as geometry is not supported. The relationship between the physical architecture group of the information model and existing STEP application protocols supporting mechanical engineering data is elaborated in Chapter 9.

5.1.2 INFORMATION MODEL STRUCTURE

This section discusses five aspects of the information model structure and how different elements of the model relate. The aspects considered are:

- Specification completeness
- Process and specification relationship
- Structure to capture method specific attributes
- Structure for context independent representation
- Specification elements and system architecture elements

A brief introduction is presented in this section. Detailed information is presented later in the thesis.

Specification completeness

The information model supports three tiers of specification elements for a system specification: a requirements tier, a functional, a physical architecture tier. In addition it is possible to include verification and validation data, and requirement and function allocation data. However, the information model does not mandate that all tiers are populated in an individual specification. In fact, a system architecture element without any requirement, functional or physical architecture data assigned is a complete specification according to the information model. This is in accordance with requirements 1 and 2 presented in Section 4.3.

Engineering process and specification relationship

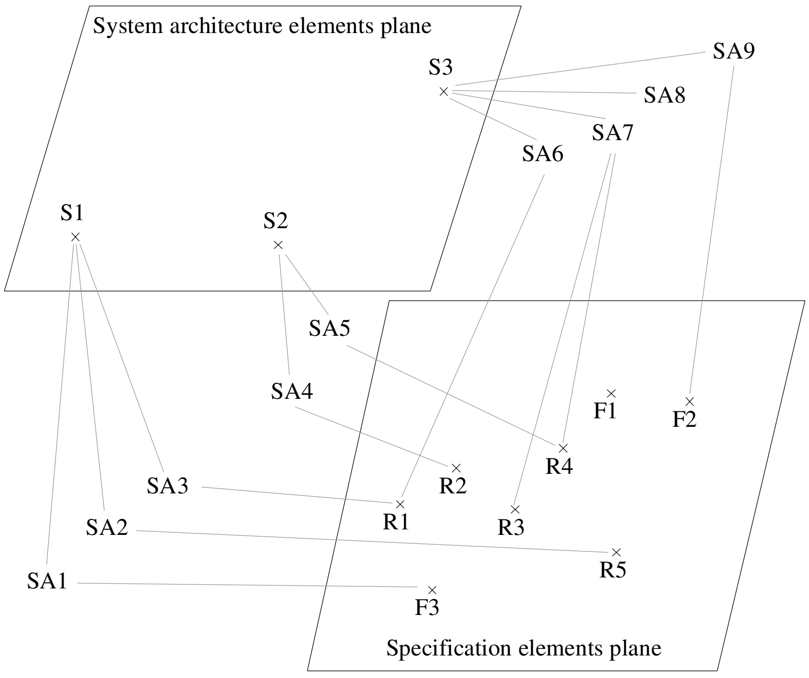
The support for representing the Systems Engineering process is independent to the specification elements and the system architecture support. Consequently, a database defined by the information model may be populated with specification element data, engineering process data or a combination thereof.

Capturing method specific specification element attributes

For each specification element only a minimal set of attributes are defined. Additional information on elements is captured through the assignment of property data. This structure allows basic and advanced tools to share basic data structures. The rationale for this structure discussed in Section 4.4 and the correspond information model representations are introduced in Section 5.3.

Context independent representation

It is possible that the same specification element, e.g., a requirement or a function is common to two or more systems, versions on the same system specification or views on a system. For traceability and data management it is beneficial if a common specification element is stored once only to avoid redundant information. At the same time it is important to capture context specific properties of a specification element shared by multiple views. For example consider a requirement statement that is applicable to multiple systems, but the priority of the requirement may be set differently in each system. Consequently a separation is made between the representation of system invariant (e.g., a requirement statement) and system variant information (e.g., system specific requirement properties). This approach is in accordance with requirement 3 defined in Section 4.3.


System architecture elements plane
S3
SA9
SA8
SA7
SA6
S1
S2
SA5
SA4
F1
F2
R4
SA3
R2
R1
R3
SA2
R5
SA1
F3
Specification elements plane

Legend:
Sx: System architecture element number x
Rx: Requirement element number x
Fx: Functional element number x
SAx: System assignment relationship x

Figure 5.2: Relationship between specification elements and system architecture elements

Specification elements and system architecture elements

Specification elements are essentially independent of the system architecture elements in the sense a specification element may be assigned to zero, one or more system specifications and a system architecture element may have zero or more specification elements assigned. System specific properties are captured for each assignment of a specification element to a system architecture element. The relationship between specification elements and system architecture elements can be illustrated using two planes, one for specification elements and one for system architecture elements. Explicit assignments are required to relate elements on the planes. This is illustrated in Figure 5.2, where a number of requirements are assigned to three different systems.

5.2 Discussion

The information model does not contain any fixed definitions of what the contents of a system specification shall be. Instead it offers the capability to define multiple views on the system under specification (in line with the requirements presented in Chapter 4). These may capture requirements for a specific system life-cycle or at a selected abstraction level using any of the notations supported by the information model. A view may also capture the system from a specific stakeholder perspective, for instance from the perspective of a system safety specialist. Life-cycle and stakeholder views are potentially independent. A stakeholder view may, for instance, be valid for multiple life-cycle views or a version of the specification.

The scope and structure of the information model is similar to with the one proposed by Oliver [114], Jackson [71] and Compatangelo [36]. Also the structure with three kinds of specification elements corresponds to the structure prescribed by the IEEE-1220 standard (illustrated in Figure 3.5). Moreover the system architecture and requirement representation parts of the information model correspond to the scope of the RQML data exchange information model for textual requirements [56].

5.3 Systems Engineering Data and STEP Data Architecture

The STEP data architecture also influences the structure of the information model. Every STEP application protocol depend on the information model fragments defined in the STEP Integrated Resources. Consequently the basic data architecture is largely similar in all application protocols. The architecture defining product structure is outlined in Section 2.9, is

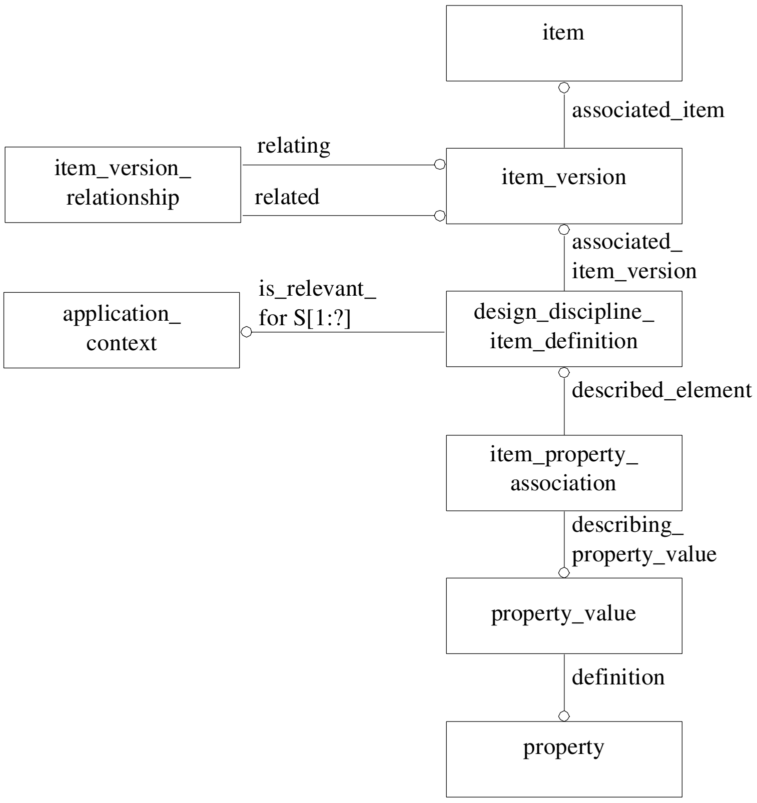


Figure 5.3: STEP data architecture as instantiated in AP-214

defined in part 41 [9], [10] and [11] of the STEP framework. For reference, Figure 5.3 outlines how the STEP data structures are instantiated in the Application Reference Model (ARM) of AP-214 [105]. The figure only illustrates an small excerpt of the AP-214 ARM. There are multiple entities and relationship not depicted. Note that the terminology used is somewhat different compared to the overview structure presented in Figure 2.11. This is explicitly allowed in an ARM as it captures concepts using the terminology used within an engineering domain. A comparison between the terminologies used in AP-214 and by Barnard Feeney [47] as presented in Figure 2.11 is presented in Table 5.1.

Table 5.1: Terminology comparison

Barnard Feeney	**AP-214**
product	*item*
formation	*item_version*
product_view_definition	*design_discipline_item_ definition*

The following comments on the architecture are of importance:

- The attribute *associated_item* from *item_version* to *item* shall be interpreted as an *item_version* object is associated to exactly one *item* and that an *item* can have any number of *item_version* objects associated.
- The entity *item_version_relationship* allow the definition of directed graphs of *item_version* objects. Cycles are not allowed in the graph.
- The relationship between *design_discipline_item_definition* and *item_version* objects is similar to that of *item_version* and *item*. Each *design_discipline_item_definition* is associated to exactly one *item_version* object via the *associated_item_version* attribute, and there may be any number of *design_discipline_item_definition* objects associated to an *item_version* object. Each *design_discipline_item_definition* is applicable to one or more life-cycle or application domains as captured by the *application_context* entity.
- Any number of properties, in different representations can be associated to a *design_discipline_item_definition* through the *item_property_association* entity. In the mechanical engineering domain typical property specify the mass, cost or quality data on a *design_discipline_item_definition*.

In addition to the model elements presented in Figure 5.3 there are also a large set of entities defined for representing composition or assembly-component structures between *design_discipline_item_definition* objects. There are also entities defined for relating *property_value* and *property* objects respectively. It is thus possible to capture that a specific property object is a property of another property object.

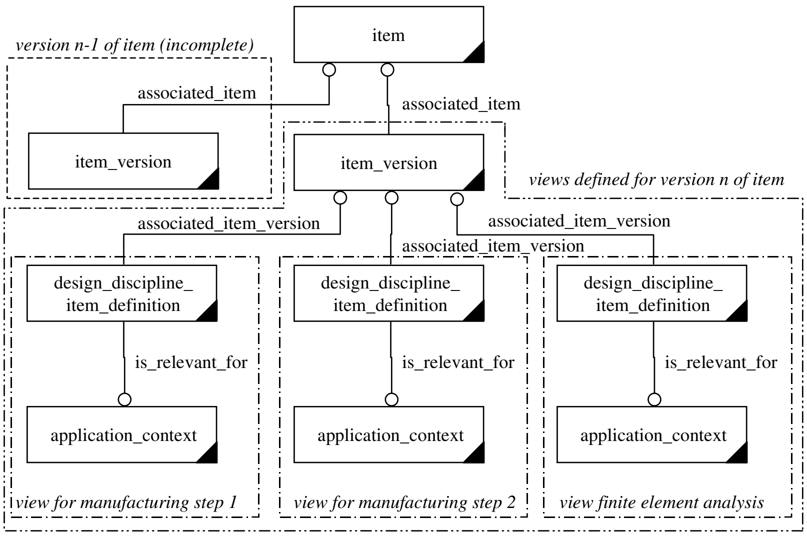

item
version n-1 of item (incomplete)
associated_item
associated_item
item_version
item_version
views defined for version n of item
associated_item_version
associated_item_version
associated_item_version
design_discipline_ item_definition
design_discipline_ item_definition
design_discipline_ item_definition
is_relevant_for
is_relevant_for
is_relevant_for
application_context
application_context
application_context
view for manufacturing step 1
view for manufacturing step 2
view finite element analysis

Figure 5.4: STEP data architecture instantiated example

The described structures lend itself naturally to mechanical product design. Any number of versions can be maintained capturing old or alternate design approaches for a product. Life-cycle views, for instance, the number of steps in a production process a parts goes through is captured by *design_discipline_item_definition* objects. The same construct can also be used to capture different analysis models, e.g., for finite element analysis. Figure 5.4 illustrates a sample instantiation pattern with multiple product views associated to a single *item_version* object.

5.4 Usage of the STEP Data Architecture

The main initial design decision is the identification of Systems Engineering concepts that shall be represented as products/items in the STEP data architecture, i.e., the set of entities for which version and configuration

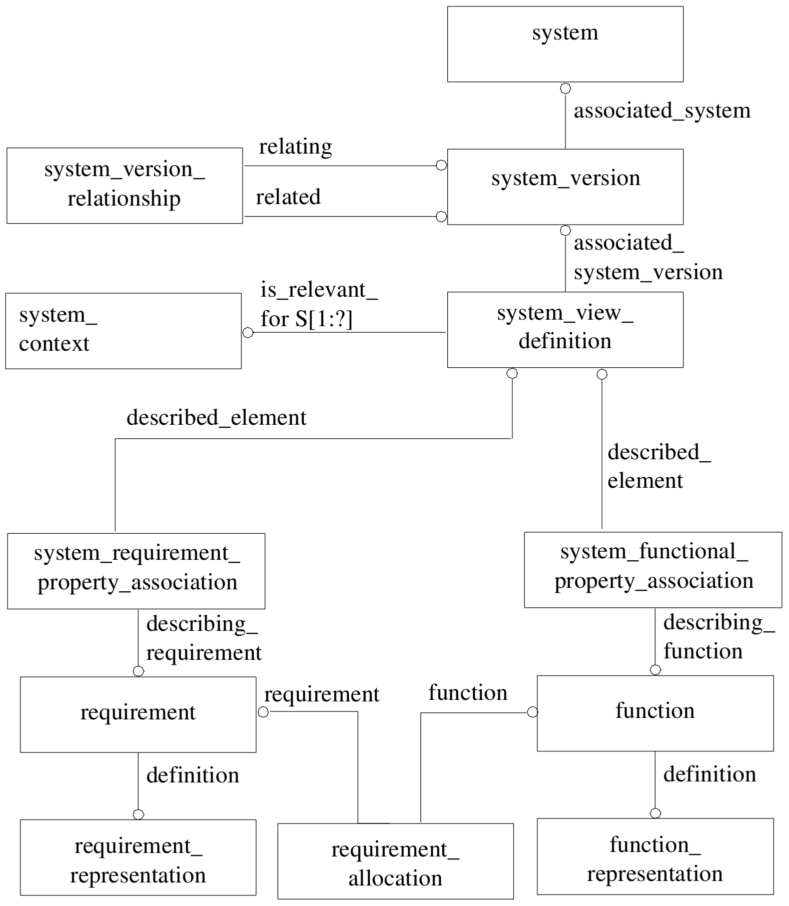


Figure 5.5: Alternative architecture for systems engineering data data representation

control will be supported. There are at least two alternative approaches to representing Systems Engineering data within the architecture as outlined below.

5.4.1 SPECIFICATION ELEMENTS AS PROPERTIES

The most straightforward approach would be to model the entity capturing the system under development corresponding to an *item* in AP-214, then each version of the system would be represented in analogy with the *item_ version* in AP-214. Likewise the life-cycle definitions would correspond nicely with the *design_discipline_item_definition* concept. Following this approach specific *property* entities should be defined to represent specification elements such as requirements, functionality or system components or verification and validation data. Traceability from requirements to functions could be established by relating a requirement property to a function property or to physical component property. The structure of such an architecture is outlined in Figure 5.5.

The architecture in Figure 5.5 fits nicely with the data representation requirements presented in Section 3.5 with the exception that only the version history of a system is maintained over time and not the individual elements of a specification. There is no mechanism to capture how individual requirements and other specification elements evolve in time.

5.4.2 SPECIFICATION ELEMENTS REPRESENTED AS *ITEM*

The second alternative is to model individual specification elements using the same structure as for *item*, *item_version* and *design_discipline_item_ definition* as presented in Figure 5.3. In this approach the resulting model structure for system, requirement and function data is presented in Figure 5.6. In the architecture presented in Figure 5.6 there is support for representation of version history information for individual requirements and functions as well as for complete systems specifications. In addition, any number of properties may be captured for each of the '_definition' objects in Figure 5.6. This is in line with the requirements presented in Section 3.5 and the architecture selected for the information model.

In Figure 5.6 unique entities are used for capture of identification and version management information for requirements, functions and systems. A large number of entities with similar definition will have to be defined if this schema is adopted for all elements requiring version management support in the information model.

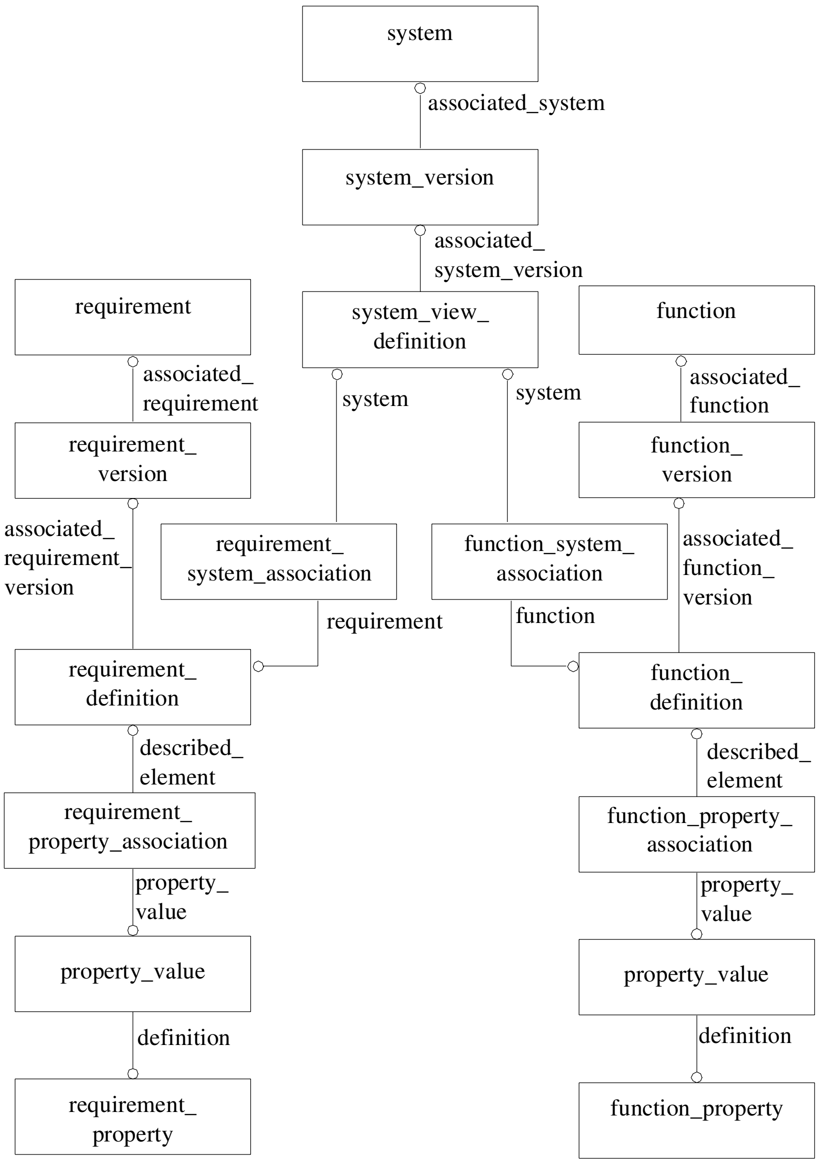


Figure 5.6: Preferred architecture for systems engineering data representation

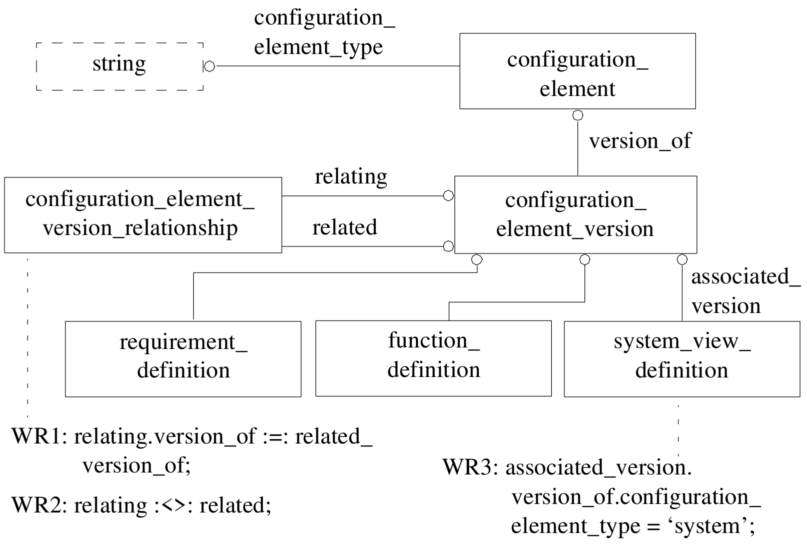


Figure 5.7: Information model architecture outline

Alternatively the formal constraint capability in EXPRESS can be used to reduce the number of entities in the model to the structure illustrated in Figure 5.7. A single common structure defined by the entities *configuration_element*, *configuration_element_version* and *configuration_element_version_relationship* can be used to identify systems, functions, requirements and their respective versions and version relationship.

The attribute *configuration_element_type* of the entity *configuration_element* is introduced to distinguish between different types of *configuration_element* objects. For *system_view_definition* type objects the attribute shall be set to the string 'system' and for *function_definition* type objects the attribute shall be set to the string 'function'. The rule WR3 in Figure 5.7 of the *system_view_definition* entity enforces the constraint. Similar rules are defined for the *function_definition* and the *requirement_definition* entities.

The rule WR1 constrain the use of the entity *configuration_element_version_relationship* such that the pair of *configuration_element_version* objects related must be versions of the same *configuration_element* object, Finally the rule WR2 specifies that the pair of *configuration_element_version* objects related by a *configuration_element_version_relationship* object shall be two distinct objects. More complex rules are defined in the STEP integrated resources. Rules defining that the graph defined by version and version relationship objects contain no cycles are available in ISO 10303-41 [9] and not duplicated in the information model.

5.5 Representation of Composition Structures

This section presents the structure for representing composition of specification elements in the information model. Composition structures are common for systems and specifications elements. System — subsystem, requirement, functional and physical architecture composition are commonly used in Systems Engineering methods. In the information model the structure for representing composition structures is similar to that in the ARM models of, e.g., AP-214 [105]. Three entities are employed to capture composition as outlined for composition of systems in Figure 5.8. A system may be composed of instances of other systems. Each instance of a system within a system composition structure is represented by the *system_instance* entity. The *system_view_definition* entity provides the definition for any number *system_instance* objects. Composition is represented by the entity *system_composition_relationship* that relate a parent *system_view_definition* object to one of its immediate subsystem represented by a *system_instance* object.

Composition structures built by *system_instance*, *system_view_definition* and *system_composition_relationship* type objects shall not contain any cycles. The selected structure for representing composition is not only used within STEP. Equivalent structures can also be found in the CDIF data flow model subject area [30].

The appeal with the selected structure is a separation between the usage (as represented by the *system_instance* entity) of an element and how it is defined (as represented by the *system_view_definition* entity). A definition

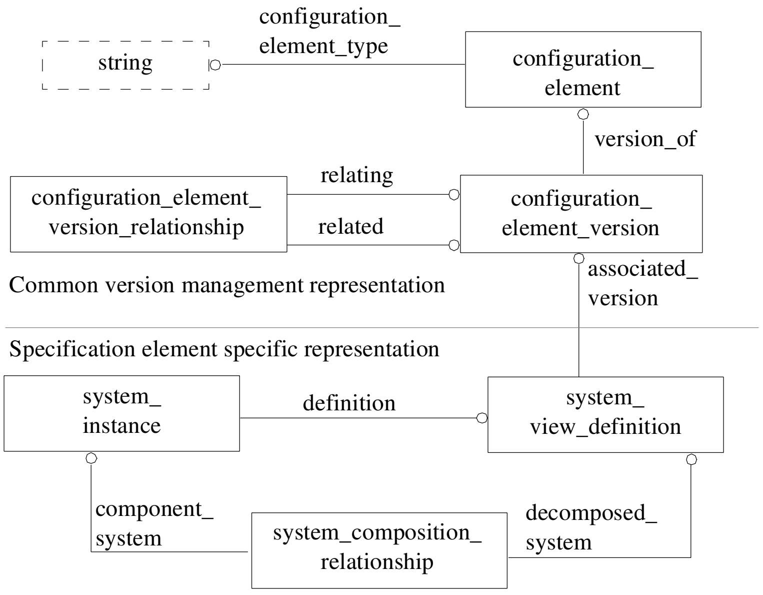


Figure 5.8: Structure for representing composition

object is self-contained and may be referred to by any number of instance type objects. Instance objects from multiple composition structures may reference shared a definition object. This could occur for specification elements for systems or for system versions with high degrees of commonality.

Also a definition object may be referenced by multiple instance objects from the same composition structure. This would be the case if there are multiple instances of an element in a composition structure. For a system composition structure this would happen in cases where there are multiple identical subsystems. Figure 5.9 illustrates a system composition structure for a control and supervision system with three subsystems. There is a primary and secondary control system with identical specifications indicated by the two *system_instance* objects (at label 1 in Figure 5.9) referring to the same *system_view_definition* object.

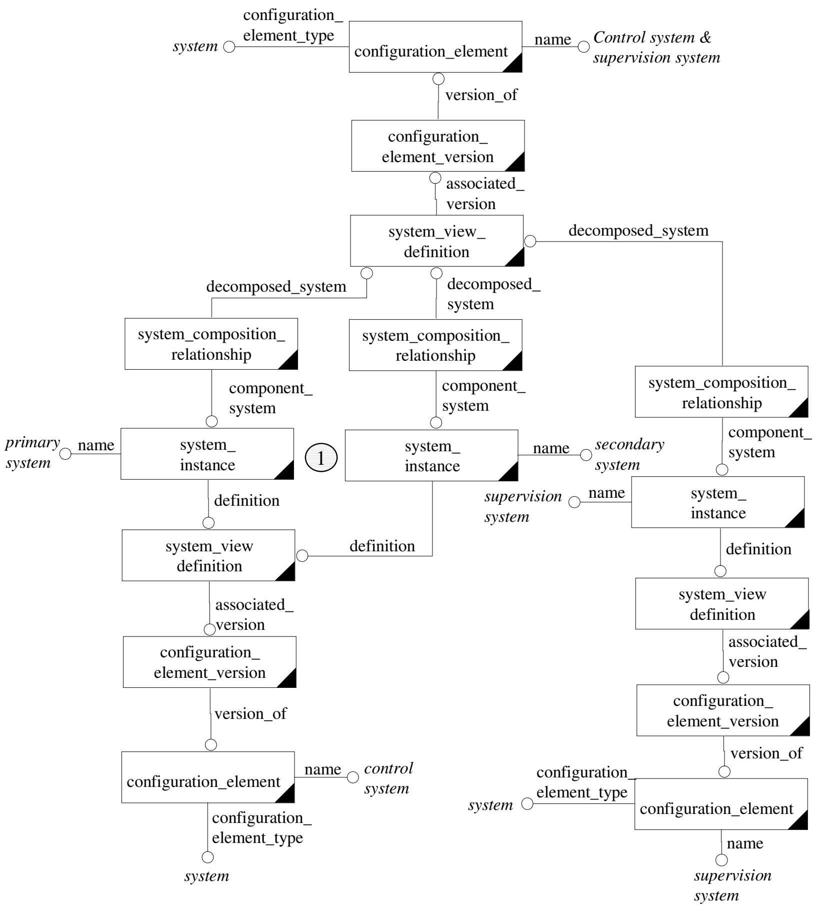


Figure 5.9: System composition structure with two identical sub-systems

A consequence of the selected structure for representing compositions is that a child element in a composition can only be uniquely identified through its parent. Usage or context specific properties for elements in a composition structure must be assigned with an unambiguous reference to the root element in the composition. This issue is further discussed in Section 5.6.

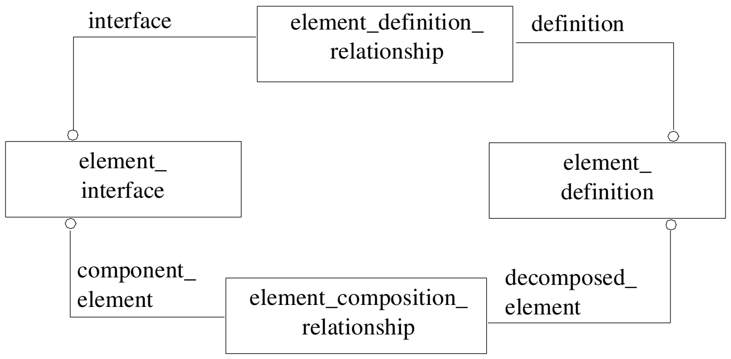


Figure 5.10: Mascot model for representing composition in EXPRESS-G

5.5.1 ALTERNATE STRUCTURE FOR REPRESENTING COMPOSITION

A more elaborate structure for representing composition is presented for the Mascot method [140]. An instance type object in Mascot may refer to multiple definition type objects, indicating multiple feasible realisation alternatives or configurations for an *element_instance* object. The structure of the Mascot information model, encoded in EXPRESS-G, is presented in Figure 5.10.

The Mascot model structure were considered in the information model, but was not selected on the ground of added information model complexity and caution whether it would be possible to map the Mascot primitives onto STEP integrated resources. As a consequence any method supporting the structure presented in Figure 5.10 cannot be adequately represented in the information model.

5.5.2 COMPOSITION STRUCTURE CONSTRAINTS

Composition structures are constrained such that all elements on the composition must be of the same basic type, e.g., a requirement may only be composed of other requirements. However, the representation for individual requirements may be different. Traceability (allocation) relationships are used to capture relationship between elements of different types. These structures are further outlined in Chapter 11.

5.6 System Variant and Invariant Information

The information model supports multiple composition structures for a system specification, e.g., for requirements and functions. As noted in the previous section these structures are general and may be used in multiple system specifications. In this sense a composition structure is system invariant.

Another important view supported by the information model is representation of traceability links capturing allocation of requirement, functional and verification plan elements. In contrast with the composition structures allocation is local to the context of a single system. This is illustrated by example 1 below:

> **Example 1:** Two systems may share a common functionality, but the requirements for selecting the functionality in the respective system may not be identical. Hence a common functional solution is derived from non-identical requirements
>
> ■

Likewise there are cases where properties captured for a specification element is local for a specific system as illustrated by example 2 below:

> **Example 2:** A requirement statement may apply to multiple systems, or versions there of, but the priority set for the requirement is decided locally within the context of each individual system.
>
> ■

The examples above indicate the need for a system variant view on specification elements. The information model shall be structured such that it shall be possible to make a system specific reference to any element in a composition structure to support property assignment and allocation of a

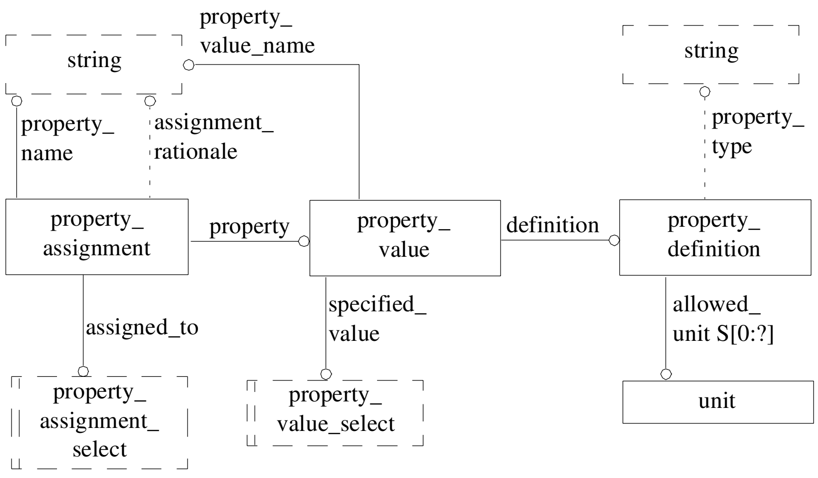


Figure 5.11: Generic property representation structure

specification element to another in the scope of a specific system. The objective is to support combination system invariant and system variant representations within a single specification.

5.7 Property Representation

Only a very small set of attributes is defined for each core entity in the information model. This is the consequence of the requirements for the information model to be tool and method independent and the requirement on absence of completeness criteria presented in Chapter 4.

A *property* is a piece of data that apply to one or more design elements. It does not have a meaning on its own, but adds information to the assigned object.

In the information model there are several structures defined to capture and assign properties to other information model elements. The space of possible properties and property representations are method dependent and essentially infinite. Consequently the structures for capturing property information in the information model are generic and applicable to multiple entities.

Figure 5.11 illustrates the generic property representation structure selected. The structure is similar to that defined for AP-214 [105]. The type of a property, e.g., weight, is captured by the *property_definition* entity. The intention is that only one *property_definition* object for each basic property shall be instantiated.

The *property_value* entity provides the capability to capture a value for the property via the attribute *specificed_value*. The attribute type, *property_value_select* defines multiple value representations, e.g., textual or numeric representations. The value may be captured by a single item or by an aggregate. Numeric representations include open and closed intervals, tolerances and capture of property value distributions functions.

The *property_assignment* entity is the mechanism for associating a property to another entity in the information model via the attribute *assigned_to* of type *property_assignment_select*.

All three entities capturing property information have a name attribute that captures the usage of the respective entity:

- *property_definition*, attribute *property_type* captures the type of the property, e.g., **weight** or **cost**.
- *property_value*, attribute *property_value_name* combines the specified value with a usage definition for the value, e.g., **empty** weight or **development** cost
- *property_assignment*, attribute *property_name* define the role for the property in the scope of the assigned object, e.g., **desirable** empty weight or **estimated** development cost.

In addition to the representation presented above there are property representations similar in structure defined to capture:

- Person and organisation information of a stakeholder in a specification element.
- Specification element approval information.
- Effectivity, a time period for which a specification element or property is valid for association with another specification element or property.
- General comments and assessments to a specification element.
- Prioritisation or ranking of multiple alternatives.

A property type may be assigned to a specification element with multiple roles. There may be multiple prioritisation criteria and multiple stakeholders performing each prioritisation. Property roles are not defined normatively in the information model. The reason for this is that all cases of organisation and method specific use of properties cannot be foreseen. Instead the information model documentation propose a set of roles that a property may be used in, but a user of the information model may define organisation specific roles for each property. For successful data exchanges it is important that property definition and property representation harmonisation is performed between all involved parties. This may be done on a partner to partner or project to project basis, or through the definition of a set of public recommended usage documents for the information model.

5.8 Summary

The chapter has presented a conceptual overview of the information model, provided an high level motivation for the structure selected and illustrated alternatives for representing Systems Engineering data in structures conformant with the structures defined in STEP integrated resources. The overview provided is important for understanding of the next chapters where parts on the information model are presented in detail and in relative isolation from other parts.

PART III

Information model presentation

Chapter 6
System Architecture

This chapter presents the information model support for capturing the structure of a system specification and for representing relationships between systems. Key concepts are illustrated through sample information model instantiations.

6.1 System Architecture

In the Systems Engineering literature the term *system architecture* is used to describe slightly different concepts. Rechtin [127] defines system architecture as a combination of system requirements and system composition structure. In IEEE-1220 [67] the term is used as a synonym for the physical architecture of a system and Lewis [90] defines system architecture as a combination of the functional and physical architecture of a system and also explicitly include the notion of life-cycles in the definition.

In the information model a *system* is the identifier and entry point for all data related to a system specification. In this thesis two architectural views are considered with the term *system architecture*:

- The *internal architecture*, referring to the internal structure of a specification, including system life-cycles and domain specific viewpoints.

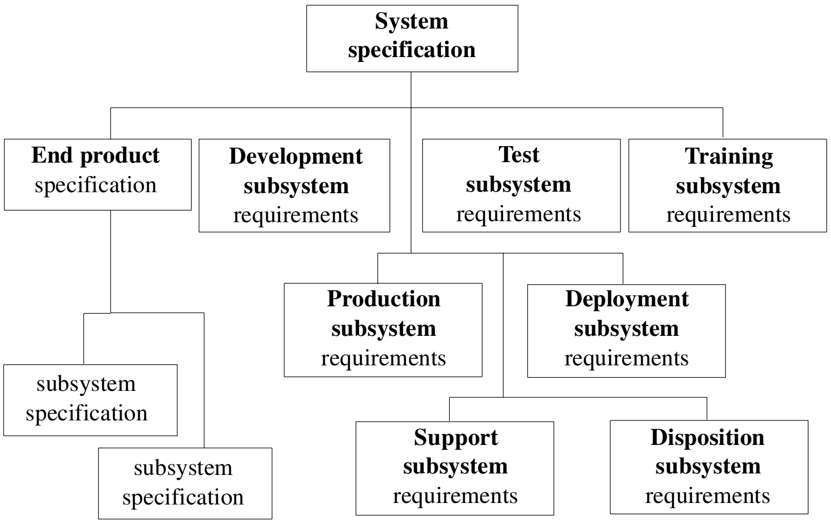

System specification
End product specification
Development subsystem requirements
Test subsystem requirements
Training subsystem requirements
Production subsystem requirements
Deployment subsystem requirements
subsystem specification
subsystem specification
Support subsystem requirements
Disposition subsystem requirements

Figure 6.1: System specification structure including end products and enabling systems [101]

This purpose of this view is to allow grouping of all information relevant to a system under a single object.

- The *external structure*, referring to the relationships between systems, for instance a system composition structure where each node in the structure corresponds to a specific system.

Each architectural view is described in the next two sections.

6.2 Internal System Architecture

It can be expected that a number of stakeholders contribute to the definition of a system specification. The nature of each contribution, and the notation the data is captured in, depend on the role of individual stakeholders and the maturity of the specification.

In literature there are multiple proposals for capturing and storing system architecture information. A common partition is to separate user (or stakeholder) requirements, specifying the expectations the user have on a system under specification, and system requirements, a transformation of user requirements into more precise, technical statements [142].

More elaborate partitions have been proposed. Wymore [156] identifies seven life-cycle oriented partitions. Figure 6.1 illustrates the partitioning of a system into end products and the enabling systems that are required for the realisation and support of the end products. This partition is defined in, e.g., EIA-632 [101]. Each enabling system can be viewed as a life-cycle partition of the overall system specification.

Likewise there are methods that propose the use of system viewpoints, profiles or scenarios for analysis of a particular aspect of a complex system [24]. Methods have been proposed both in a Systems Engineering [3] [38] [91] and in a software engineering context [83] [110]. For example, *use cases* are a fundamental part of the UML language [130].

In this thesis we use the term *viewpoint* to relate to cases where a knowledge source and/or perspective is used to create a logical partitioning of a system under specification. Individual viewpoints may be captured using different methods and thus make use of different notations and representations.

In environments where multiple tools are used it could also be beneficial to store the view on the system held in individual tool representation as viewpoints. Such an approach would allow the special characteristics supported by individual tools to be maintained. If there exist a substantial overlap in concepts supported by a pair of tools (A, B) as indicated by figure Figure 6.2 and each tool is used such that all concepts supported are employed then any attempt to manage the complete system specification in one of the tools will lead to the loss of information as soon as data is exchanged. Information that do not belong to the intersection of the data coverage of the tools will be lost. The preferred approach in the outlined scenario would be to manage the data in two separate viewpoints and exchange data supported by both tools.

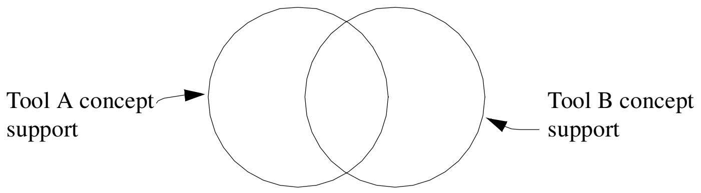


Figure 6.2: Tool concept overlap

6.2.1 VIEWPOINTS AND SPECIFICATION CONSISTENCY

The definition of multiple viewpoints offers advantages in the sense that it facilities for a structured analysis of a system. However there is also the risk of introducing extra-viewpoint inconsistencies. If a viewpoints is updated then there is a risk it will no longer be consistent with other viewpoints. I.e., a modification in one viewpoint may have a subtle impact on other viewpoints. These risks have been acknowledged and resolution strategies have been proposed in [42] [52] [110]. One of the conclusions in the cited work is that specification inconsistencies can and probable must be tolerated in some phases of the development process if multiple viewpoints are introduced.

6.2.2 VIEWPOINTS AND DATA EXCHANGE

There appear to be multiple advantages with introducing support for multiple viewpoints in the information model despite the risks associated with introducing inconsistencies. Viewpoints are of special interest in a data exchange environment as they allow for:

- Maintenance of tool specific views on a system under development.
- Capture of relationships between tool views on a system.
- Management of information originating from multiple stakeholders.

6.2.3 VIEWPOINT DEFINITION

As with identification of system life-cycles there appears to be no consensus on a standard set of viewpoints. Consequently, the system architecture model shall provide a flexible framework capable of adapting to industrial practise rather than a predefined fixed structure presented as solution to all system specifications. Process and method harmonisation, i.e., definition of and agreement on life-cycles to be considered and on the perspective taken in each individual viewpoint, is a prerequisite for effective use of the information model for data exchange.

6.2.4 INTERNAL ARCHITECTURE INFORMATION MODEL

The part of the developed information model supporting representation of the system internal architecture is presented in Figure 6.3. It is based on the structure outlined in Figure 5.7, with extensions to allow for capturing life-cycle oriented views, represented by the *system_definition* entity, and viewpoints represented by the *partial_system_view* entity. The entity *system_view_definition* is a common abstract supertype for both concepts and is introduced to define a common entry point to the set of entities that may be associated with a life-cycle view or viewpoint. The following types of specification elements may be associated with a *system_view_definition* type object:

- Requirements.
- Verification and validation data.
- The top-level node of the system functional architecture.
- The top-level node of the system physical architecture.
- Documents, i.e., files with an arbitrary content.

The mechanisms for assigning individual specification elements to a *system_view_definition* type objects are presented in detail in chapters 7 - 10.

For any version of a system specification there may be any number of life-cycle views (represented by the *system_definition* entity). Each life-cycle view is valid for zero or more defined life-cycle contexts. The model does not contain any directives or definitions of acceptable life-cycle contexts.

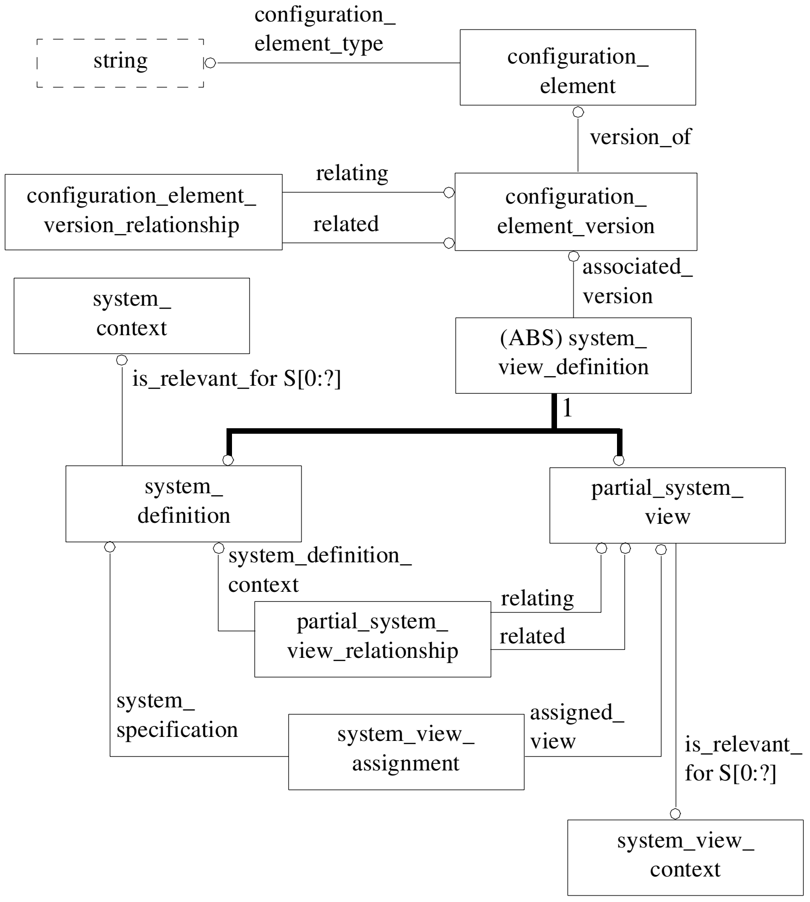


Figure 6.3: System architecture, life-cycles and viewpoints

If desirable more than one *system_definition* object can be defined to be relevant to a specific life-cycle (represented by the *system_context* entity). This enables the creation of clusters of specifications for related life-cycles as illustrated in example 3 below:

Example 3: Two *system_definition* objects defined for the same version of a system may both capture system requirements, one for an operational life-cycle and one for a maintenance life-cycle.

■

System specific viewpoints are captured by the *partial_system_view* entity. Each viewpoint is valid for zero or more contexts, captured by the entity *system_view_context*. A *system_view_context* defines both the scope and analysis fidelity selected for the viewpoint.

partial_system_view and *system_definition* type objects must not be assigned to the same *configuration_element_version* structure, i.e., a system configuration element is either representing a life cycle view or a viewpoint. This is enforced by the EXPRESS rules presented in Figure 6.4.

Two relationship entities are defined to relate *system_view_definition* objects. The entity *system_view_assignment* relates a *partial_system_view* object to a *system_definition* object indicating that the *partial_system_view* defines a viewpoint on the system. A *partial_system_view* may be assigned to any number of *system_definition* objects indicating that the viewpoint is valid for multiple systems, system version or system life-cycles. However it may only be assigned once to a specific *system_definition* object. This constraint realised through a uniqueness clause in the EXPRESS specification requiring the combination of the values of the attributes *system_specification* and *assigned_view* to be unique in the data exchange database. Moreover, only one version of a viewpoint (*partial_system_view*) may be assigned to a specific life-cycle view (represented by a *system_definition* object).

The entity *partial_system_view_relationship* captures a relationship between a pair of *partial_system_view* objects that is valid in the context of a defined *system_definition* object. The entity may only relate *partial_system_view* objects that are assigned to the same *system_definition* object via *system_view_assignment* objects. The semantics of the relationship is captured by a textual attribute intended to clarify to an engineer how the relationship shall be interpreted. Further textual information on the nature of the relationship may be captured in the entity's *description* attribute.

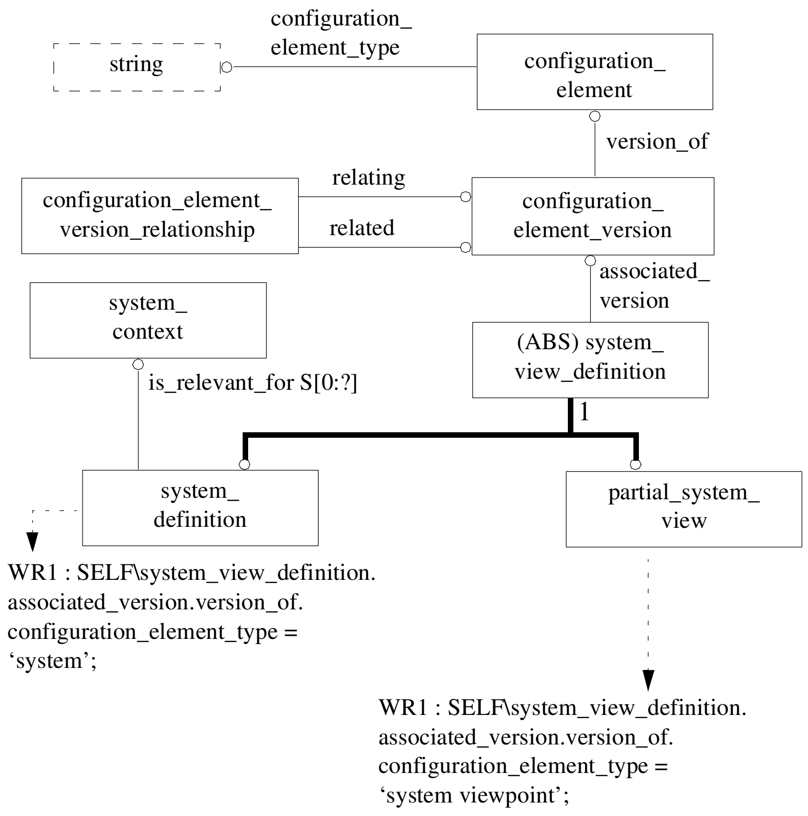


Figure 6.4: Rules enforcing separation of life-cycle and viewpoint version management structures

The information model does not include any formal language to define the nature of a relationship between two viewpoints. Thus there is no built-in support for automated reasoning about inter-viewpoint relationships. This lack of formality may be criticised but is motivated by the fact that there exist no consensus within the Systems Engineering community on how different system viewpoints shall be related. Realistically, any attempt to formally define a fixed number of relationships would fail as individual stakeholder have different definitions for the same concept and attempts to enumerate all potential relationship likely to result in a sub-

```
System A
  Version 1
    life-cycle view 1
      Viewpoint A
        Version 1
  End version 1

  Version 2
    life-cycle view 1
      Viewpoint A
        Version 2
      Viewpoint B
        Version 1
  End version 2

  Version 3
    life-cycle view 1
      Viewpoint A
        Version 2
      Viewpoint B
        Version 2
  End version 3
End system A
```

Figure 6.5: Example viewpoint — life-cycle view configuration for specification versions

stantial, and likely incomplete list. The selected approach does not preclude formal methods from using the framework, but the formal definition of individual relationships is not maintained in the information model representation.

6.2.5 EXAMPLE

A system specification will go through several versions as it gradually becomes more mature. The example in Figure 6.5 illustrates how the internal architecture of a system can be captured across multiple specification versions using life-cycle views and viewpoints.

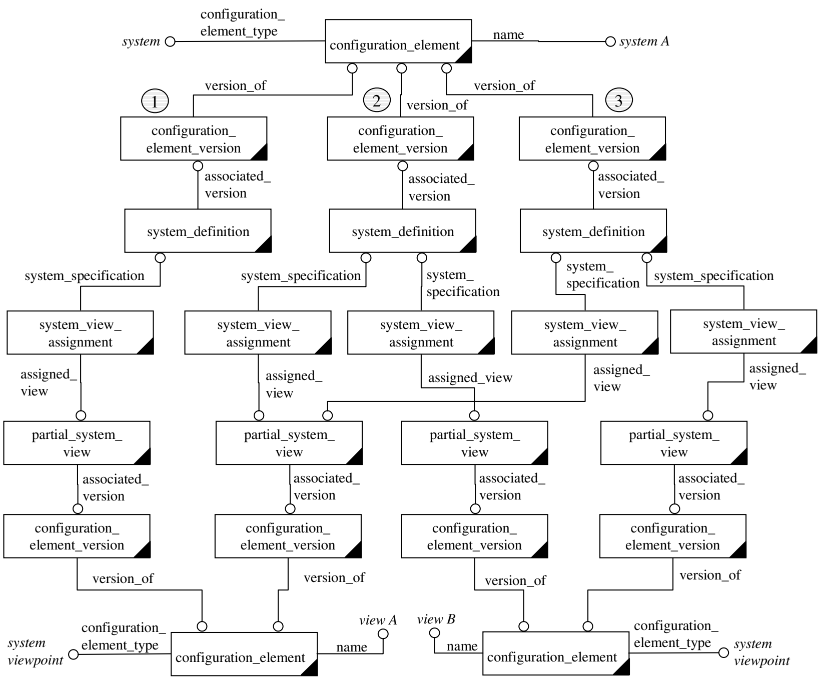


Figure 6.6: Combining viewpoints for versions of a system specification

The corresponding information model instantiation is presented in Figure 6.6. In the upper half the three specification versions (labelled one to three) are instantiated. Note that the relationships defining the version graphs have been omitted. In the lower half of Figure 6.6 the two system viewpoints, each instantiated in two versions are presented. *System_view_ assignment* objects are used to relate individual viewpoints to the system specification life cycle view they apply to. Note how a single *partial_ system_view* object can be assigned to multiple *system_definition* objects. Using this approach the commonality between two versions of the same system specification can be established through an analysis of the *partial_ system_view* objects common to both versions. This is of importance as it allows a stakeholder to quickly assess the extent of modification made between two versions of a specification.

6.3 External System Architecture

The external system architecture defines how individual systems are related logically to each other. Martin [103] identifies two practices in representing the external systems architecture:

- Tightly coupled systems architecture
- Loosely coupled systems architecture

The characteristics of each architecture view are discussed in the next two sections.

6.4 Tightly Coupled System Architecture

For a class of systems it is possible to identify a tightly coupled tree structure where the root is the top-level system under consideration and each of its immediate children nodes are the subsystems. Individual subsystems may be further composed. Recommended Systems Engineering practise, Stevens et al. [142], is that no further decomposition is necessary when a system is scoped such that it can be realised by a group of domain experts. This approach to describe the external system architecture lends itself naturally to complex non-distributed systems where there is a well-defined system boundary.

> **Example 4:** In the development of a fighter aircraft the following subsystems may be considered: the avionics system, the propulsion system, and airframe system.
>
> ■

In Example 4 the number of individual subsystems are fixed for each life-cycle view on the system. There is typically one avionics system, one or two engines in the propulsion system and one airframe system per aircraft system.

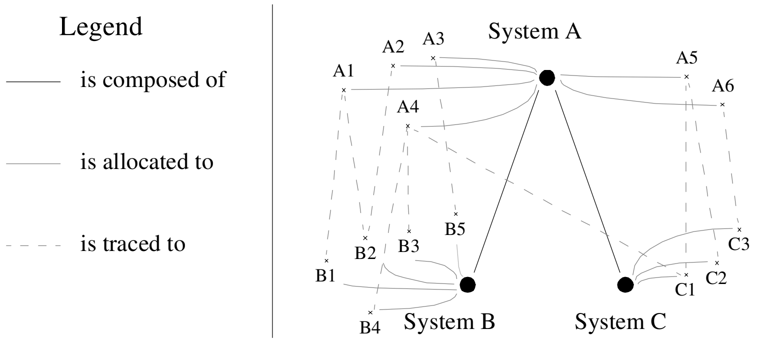


Figure 6.7: Requirement traceability in a tightly coupled system composition structure

6.4.1 SYSTEM CONFIGURATIONS

In the tightly coupled architectural view it can be expected that the subsystems of a system shall collectively fulfil the requirements stated on the system. The requirements stated on a subsystem shall, according to theory, be directly or indirectly traceable to requirements stated on the parent system. These relationships may be implicit or captured explicitly.

Figure 6.7 illustrates the structure for a trivial system architecture where requirement traceability is explicitly implemented. System A is composed of two subsystems, System B and System C. For System A, six requirements, A1..A6, have been identified, illustrated by the grey lines in Figure 6.7. Likewise five requirements, B1..B5, are identified for System B and three requirements, C1..C3, are identified for System C. The dashed lines in Figure 6.7 indicate how requirements for System A are traced to requirements for System B and System C. For instance, requirement A1 on System A is traced to requirements B1 and B2.

6.4.2 REPRESENTATION OF TOP-DOWN CONFIGURATION MODIFICATIONS

Management of the system architecture is complicated by the fact that both system and subsystem specifications may evolve independently of each other. Modifications may be introduced at system level or at subsys-

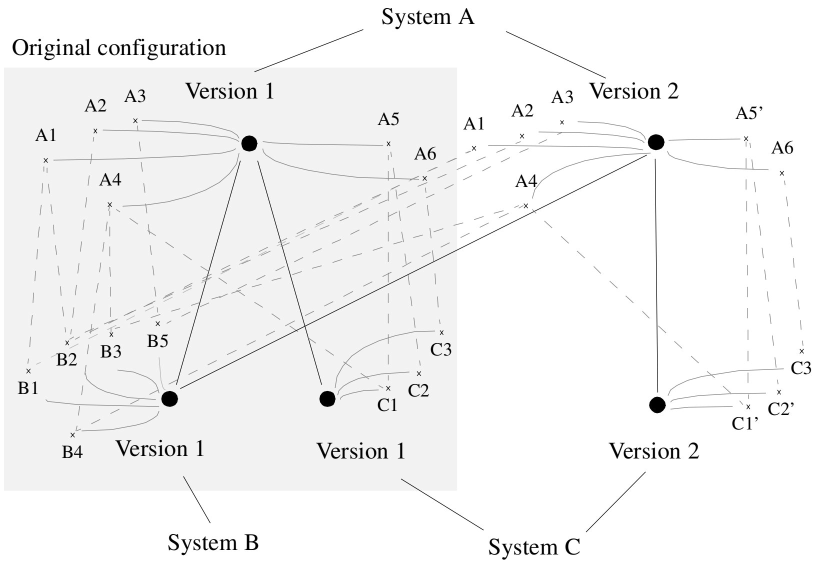


Figure 6.8: Top-down update of the system architecture illustrated by a modification of requirement A5 to A5'

tem level. If a modification is introduced in the parent system then the natural process is to create a new version of the parent specification and to analyse how the modification affect individual subsystems. In cases where the modification impacts on a subsystem requirement then a new version of the subsystem specification is created and linked to the parent system. This analysis is repeated until the leaf nodes in the system composition structure are reached.

Top down configuration modifications is illustrated in Figure 6.8. The shaded rectangle in the left side of the figure contains the system architecture from Figure 6.7 extended to include version management information. In the initial configuration System A, version 1 is composed of System B, version 1 and System C, version 1.

If a requirement on System A, (requirement A5 modified to A5') is modified then a new version of System A is created. Under the assumption that the modification from A5 to A5' neither created new nor broke any

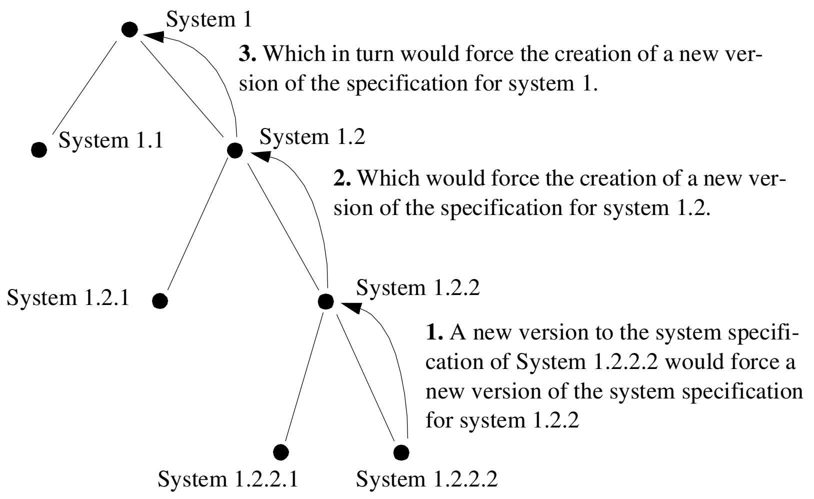

System 1
3. Which in turn would force the creation of a new version of the specification for system 1.
System 1.1
System 1.2
2. Which would force the creation of a new version of the specification for system 1.2.
System 1.2.1
System 1.2.2
1. A new version to the system specification of System 1.2.2.2 would force a new version of the system specification for system 1.2.2
System 1.2.2.1
System 1.2.2.2

Figure 6.9: Consequences of bottom-up modifications to the system architecture

traceability relationships then only requirements C1 and C2 of system C are potentially affected. If the modification had any impact to any of the traced requirements then a new version of system C must be created with requirements C1 and C2 updated to C1' and C2' respectively to comply with the change of A5. Since System B is not affected by the update there is no need to create a new version of the specification.

6.4.3 REPRESENTATION OF BOTTOM-UP CONFIGURATION MODIFICATIONS

Modifications to an element of a subsystem cannot be effectively handled in the same ways as a top-down modification as presented in Section 6.4.2. A modification to a subsystem specification that would result in the identification of a new version of the specification would force the creation of a new version of its parent system specification to identify the new configuration. The number of versions to manage for deep system composition structures would explode as illustrated by Dick and Jackson [41].

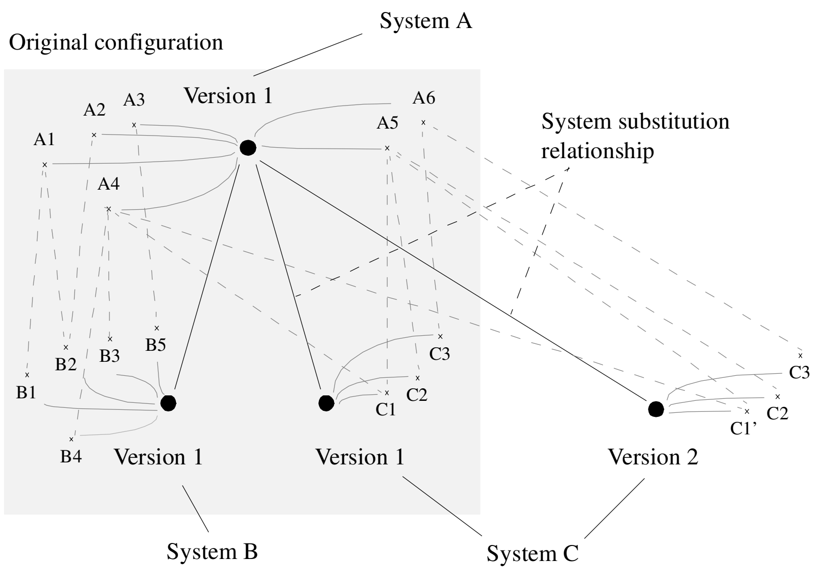


Figure 6.10: Mechanism for controlled bottom-up modification to a system composition structure

Example 5: Consider a system composition structure four level deep. Then an update to a leaf node in the architecture would force three additional updates, one at each level in the architecture as illustrated in Figure 6.9.

■

The approach in STEP for solving the described problem is to indicate that an individual subsystem is a candidate for replacement by another subsystem and qualify the replacement through the use of objects capturing the approval and the temporal validity of the substitution. In the STEP framework this structure is described in ISO 10303-44 [11].

Figure 6.10 illustrates the preferred mechanism for handling updates to a subsystem specification that does not include a modification to the specification of its parent system. A new version of the specification for System C is created when requirement C1 of System C is modified to C1'. Instead

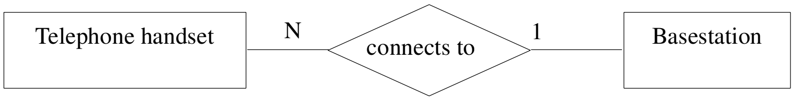


Figure 6.11: ER model for a loosely coupled system architecture

of creating a new version of the specification for System A, the new version of System C is marked as being a potential substitute to the original version. This is indicated by the system substitution relationship in Figure 6.10. Only one of the alternatives shall be considered.

If the update of requirement C1 is accepted then an approval (with appropriate authority) and effectivity objects are assigned to the substitution relationship to indicate that the subsystem version has been substituted. If the substitution is not approved then an approval object with the value 'not approved' is used to indicate that there shall be no change to the specification.

Substitution of subsystems in the in the system composition structure is not limited to system versions. The same concept can be applied to indicate that a pair of systems has been found to be interchangeable subsystems of a system.

6.5 Loosely Coupled System Architecture

For some system architectures there may be no obvious top-level system or system composition structure. Martin [103] describes how this loosely coupled architecture is common in telecommunication systems. A trivial example is the relationship between handsets and base stations of a cellular telephone system as illustrated in Figure 6.11. There are multiple autonomous nodes in the system and no obvious hierarchy relationship between the system components. Moreover, the exact system configuration may not be fixed but differ over time. For instance, the configuration of telephone handsets to a base station varies over time.

6.5.1 EXTERNAL ARCHITECTURE INFORMATION MODEL

The information model for representing the external system architecture is presented in Figure 6.12. The entities *system_instance*, *system_composition_relationship* and *system_substitution_relationship* provide the capability to define a tightly coupled system architecture and the entities *system_instance*, *system_instance_relationship_end* and *system_instance_relationship* provide the capabilities to represent a loosely coupled system architecture. Entity definitions are presented below[1]:

- *System_instance,* represents a specific instance of a particular system life-cycle view. The instance concept is introduced to capture information specific to a particular usage of a system specification in the context of other systems. For instance, the decision to use a system as a subsystem in the scope of another system. Each occurrence of a system in a system architecture is represented by a *system_instance* object.
- *System_composition_relationship*, defines the parent-child relationship between a system and one of its immediate subsystems. In this respect the entity defines a tightly coupled system architecture. The attribute *relationship_type* defines whether the subsystem is mandatory or optional in the context of the parent system. Additional detail on the relationship may be captured in text. The system architecture graph defined by *system_definition, system_composition_relationship* and *system_instance* objects must not contain any cycles.
- *System_substitution_relationship*, defines that a subsystem has been identified as a candidate for replacement by another system with the scope of the system. The alternatives are mutually exclusive. In case multiple alternatives exist in the external architecture then approval and effectivity concepts define the valid configurations. The relationship is transitive, i.e., if system C can substitute system D and system D can substitute system C then system D can also substitute system B.

1. The definition of the entity *system_definition* is presented in Section 6.2.4

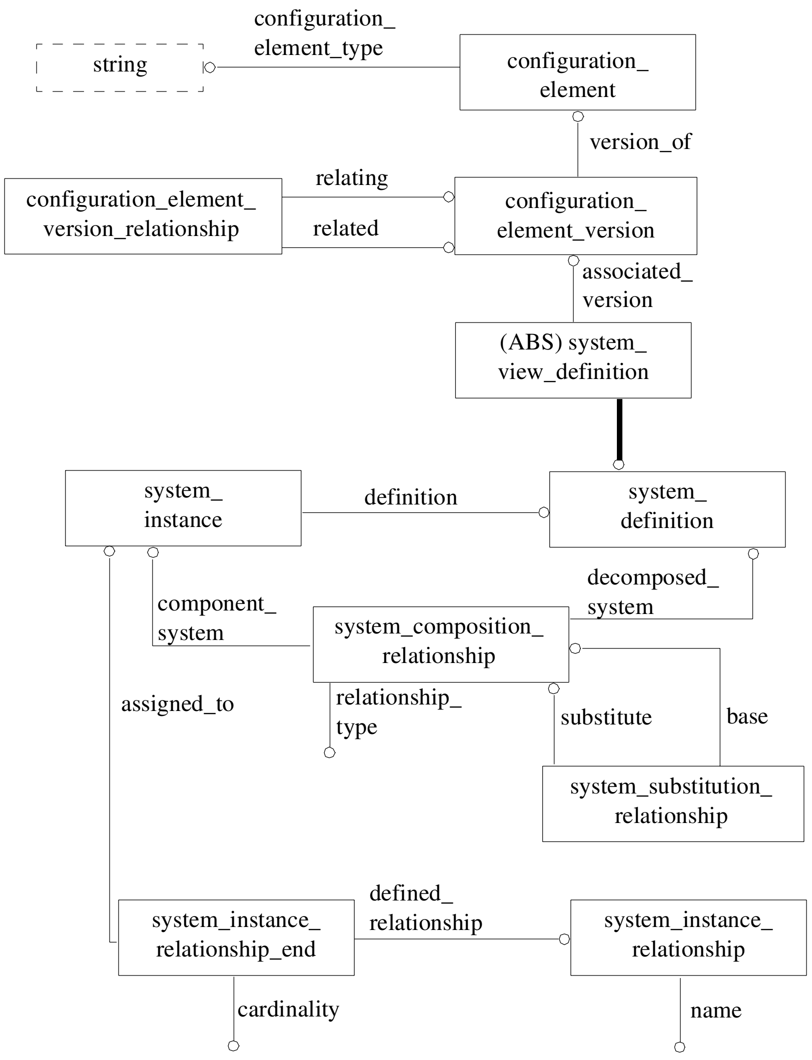

string
configuration_element_type
configuration_element
version_of
configuration_element_version_relationship
relating
related
configuration_element_version
associated_version
(ABS) system_view_definition
system_instance
definition
system_definition
decomposed_system
component_system
system_composition_relationship
assigned_to
relationship_type
substitute
base
system_substitution_relationship
system_instance_relationship_end
defined_relationship
system_instance_relationship
cardinality
name

Figure 6.12: Tightly and loosely coupled system external architecture

- *System_instance_relationship*, defines an arbitrary relationship involving two or more *system_instance* objects. The semantics of the relationship is not predefined but defined upon instantiation.

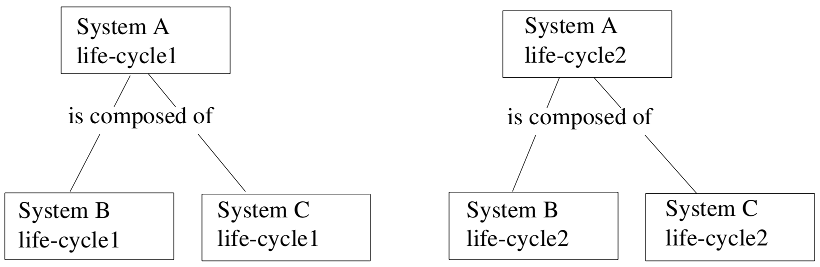


Figure 6.13: Tightly coupled system architecture with multiple life-cycle views

- *System_instance_relationship_end*, defines the cardinality of a *system_instance* object in a *system_instance_relationship*. Cardinality may be expressed as open or closed intervals of natural numbers.

The information model structure for more than one life-cycle view on a particular system is especially suitable for capturing a system external architecture that change according to life-cycle. For instance, if two life-cycles, *operational* and *maintenance*, are identified for a system then there may be a tightly or loosely coupled architecture defined for each life-cycle.

6.6 Examples

The following examples illustrate aspects of tightly and loosely coupled system external architecture. The first example presents a tightly coupled system architecture where the top-level system A is composed of subsystems B and C. Two life-cycle views are identified for each system. An informal view of the example is presented in Figure 6.13. The corresponding information model representation is presented in Figure 6.14. The labels (1 and 2) inserted identify the two life-cycles views in the example.

In this case there is an exact correspondence between the system and its subsystems for each life-cycle view, but in general this need not hold. Each life-cycle view may introduce its unique set of subsystems. Likewise the identification of system life-cycles and subsystems are independent of versions of a system specification.

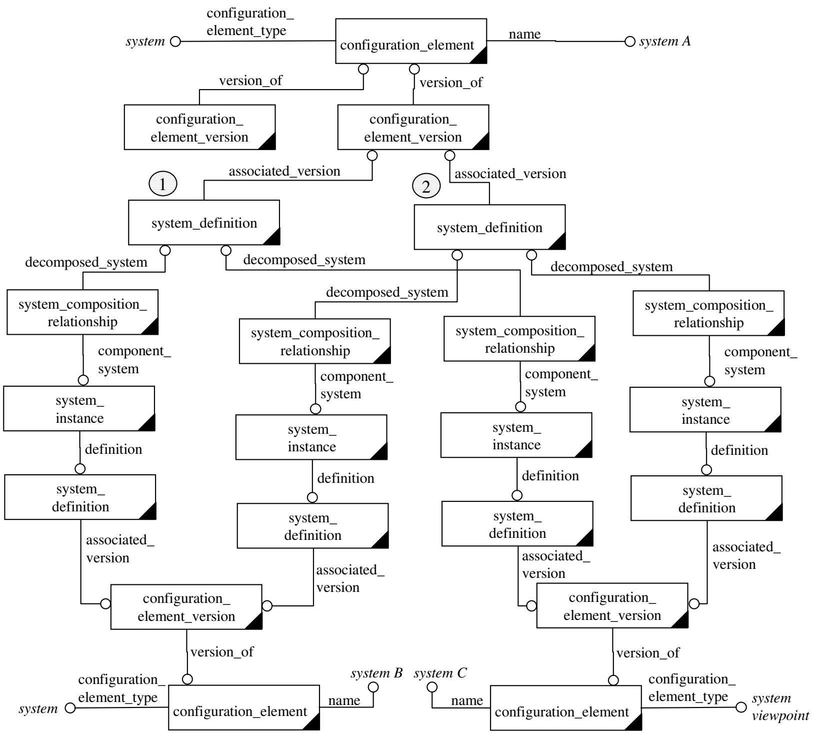


Figure 6.14: Information model representation of architecture presented in Figure 6.13

The example presented in Figure 6.15 is the information model instantiation of the system architecture example presented in Figure 6.10. In the figure, the single life-cycle view identified for System A is composed of three subsystems views: one view on System B and 2 views on two different versions of System C. The *system_substitution_relationship* object at label one in the figure indicates that the two versions of system C represent mutually exclusive alternatives (at labels two and three in the figure). Selection of the substitute over the base alternative is indicated through assignment of approval and effectivity information on the *system_substitution_relationship* object.

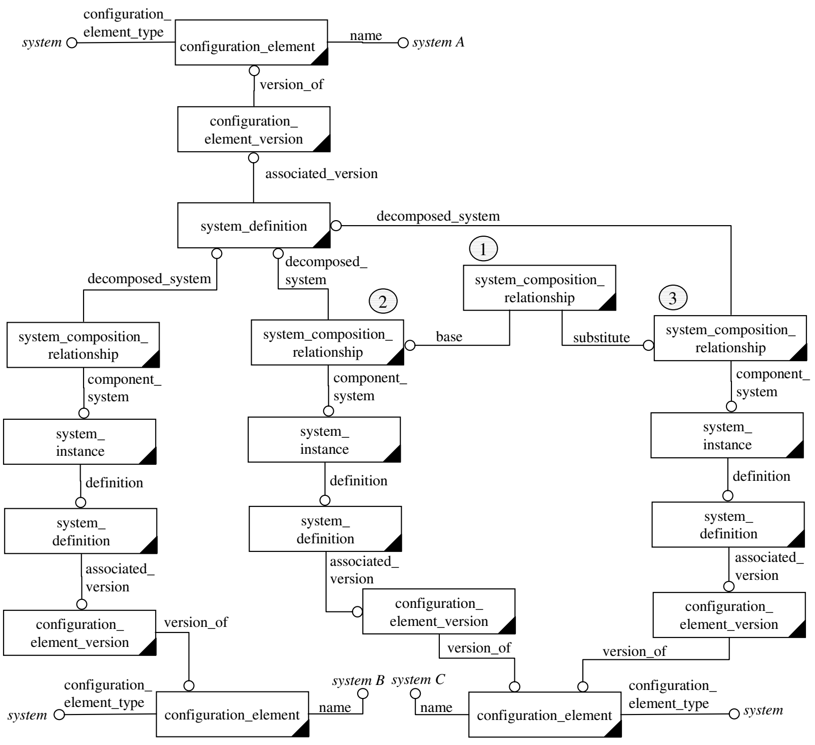


Figure 6.15: Bottom-up modification of a system architecture corresponding to the example in Figure 6.10

The structure for creating a loosely coupled system architecture is presented in Figure 6.16. The example illustrates a trivial architecture where a cellular telephony base station provides service to 0...2000 handsets, and each handset is serviced by zero or one base station.

For each system in Figure 6.16 there may be more than one life-cycle view defined and for each life-cycle view there may be more than one *system_instance* object identified. This allow for representation of more than one Loosely coupled system architecture for a system, or a tightly coupled system architecture definition complementing the architecture presented in Figure 6.16.

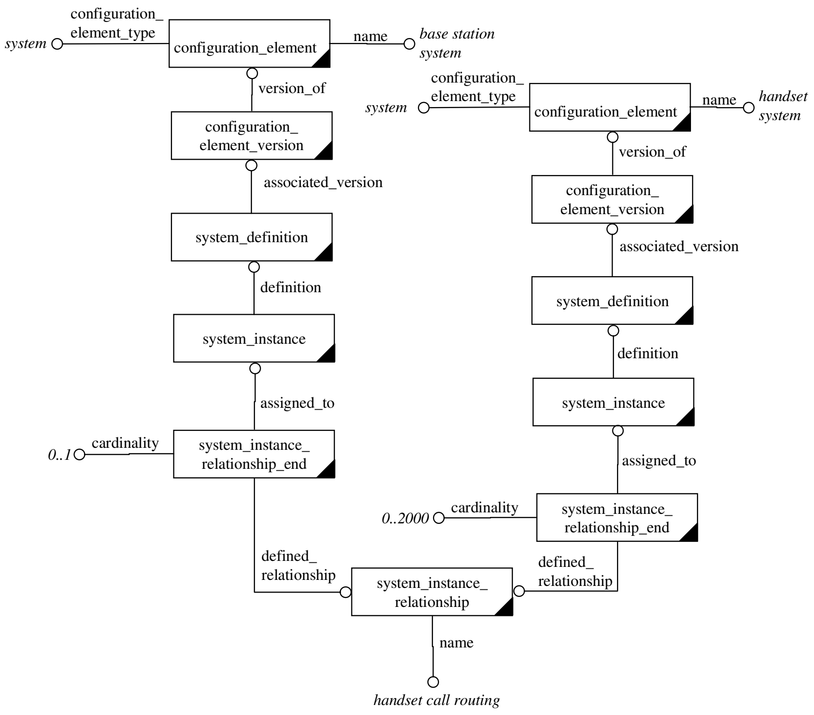


Figure 6.16: Information model instantiation of the loosely coupled system architecture example in Figure 6.11

6.7 Summary

This chapter has presented the system architecture portion of the information model. The focus has been on illustrating the flexibility in the information model for representing system internal and system external architecture. The system internal architecture supports structuring of data for a system under specification in the form of:

- System life-cycle views
- System specific viewpoints

The representation for capturing the external system architecture in terms of:

- Tightly coupled system architecture
- Loosely coupled system architecture

have also been introduced.

The structures for relating system life-cycle views and system viewpoints to individual specification elements are introduced in chapters 7 - 11.

Chapter 7
Requirements Representation

This chapter introduces and motivates the selected representation for requirement statements and associated properties in the information model. The mechanism for associating requirement properties and the assignment mechanism for requirements to system life cycle views and viewpoints is also introduced.

7.1 Requirement — Definitions

The term requirement is used in many contexts. Earlier in this thesis the term *requirements* has been used to refer to the set of information provided as input to an activity in the engineering process. In this usage the term refer to the collection of information used to guide the development of a system.

The term *requirement* is also used to refer to individual statements on the capabilities a system shall conform to. In this usage the term requirement refers to individual elements of a specification that are potentially independent of the engineering process. Requirements may be part of the input to an engineering process activity and the output may be a set of partially modified set of requirements for a system. In this usage a requirement is an element carrying information about an aspect of a system.

However, the exact nature of a requirement is not easy to define. Three useful definitions are:

1. A requirement is an expression of need, demand or obligation [142].
2. A requirement is a statement identifying a capability, physical characteristic, or quality factor that bounds a product or process need for which a solution will be pursued [67].
3. A requirement is something that governs what, how well and under what conditions a product will achieve a given purpose [102].

For data representation purposes the first definition above is the most appropriate, as it does not automatically couple the term to a system product. The identification or recording of a requirement does not automatically imply that a system that shall comply with the requirement. A requirement statement may be recorded, but never associated with a system. Strictly speaking, the statement recorded need not carry any information related to expectations on a planned or existing system.

Traditionally it has been common to equate the term requirement with a textual expression defining capabilities or constraints on what a system shall accomplish. However, it is our view that a requirement can be expressed in any notation.

7.2 Requirement Quality Attributes

Lists of recommendations for high quality requirements have been published. According to the IEEE-1233 standard "Guide for Developing System Requirements Specifications" [68] the set of requirements on a system shall have the following characteristics:

1. Unique Set: Each requirement should be stated only once.
2. Normalized: Requirements should not overlap.
3. Linked Set: Explicit relationships should be defined among individual requirements.
4. Complete: Should include all the requirements identified by the customer, as well as those needed for the definition of the system.
5. Consistent: Should be non-contradictory in the level of detail, style of

requirement statements, and in the presentation material.

6. Bounded: The boundaries, scope, and context for the set of requirements should be identified.
7. Modifiable: It should be possible to modify individual requirements without having to update multiple other statements
8. Configurable: Version information should be maintained
9. Granular: The level of abstraction selected for capturing requirements should be consistent.

An additional desirable characteristic is that a requirement shall state *what* a system shall comply with, not *how* it shall be done [16] [101].

The majority of the items above discuss characteristics unrelated to storage structures. Only items only numbers 3 and 8 contain the most abstract guidance on how requirements should be represented. Identified requirements related to requirements representation is discussed in next section.

7.3 Requirements on the Representation of Requirements

This section covers identified requirements for representation of requirements. To facilitate analysis the requirements are presented in five sections below.

7.3.1 REQUIREMENT REPRESENTATIONS

None of the characteristics identified in Section 7.2 consider suitable representations for capturing individual requirements. Examples of requirements expressed in natural language are common in literature, but any representation can be used to capture a requirement, for instance:

- Requirements expressed in natural language, e.g., *The system shall weight less than 300 kg*.
- Requirements expressed in natural language complemented with properties defined to ease the translation of the requirement into other language representations, e.g., methods supporting the identification of

parts of a functional model from the textual requirement statement as supported by the Medisys process [17].

- Requirements expressed in a formal mathematical notation [118].
- Requirements expressed as a structured expression capturing a property, with value (optionally an open or closed interval, possible with defined tolerances and distribution functions) and unit, e.g.,
  property: *weight,*
  limit qualifier: *less than,*
  value: *300.00,*
  unit: *kilogram*.
- Requirements expressed in models in a representation supported by the information model, e.g., a functional hierarchy model, CORE Extended Functional Flow Block Diagram [95], or a complete system specification.
- Requirements expressed in digital documents with widely varying internal structures, e.g., a line drawing, a 3D CAD representation, a word processor or multimedia document.
- Requirements given a physical manifestation, e.g., a physical prototype of a system.

Selection of a suitable representation for a requirement is method and process dependent. Natural language may be an appropriate representation early in the development process. Later in the process more specific representations may be preferable.

7.3.2 CONFIGURATION MANAGEMENT

The importance of the configuration management for requirements highlighted in item 8 in Section 7.2 is not disputed in this thesis. In fact configuration management support of individual requirements and other specification elements is one of the most important requirements on the information model. We assume that:

- A version of a requirement may apply to multiple systems, system versions or viewpoints.

- A requirement version may be part of multiple composition structures. For instance, a basic requirement statement can be inferred from more than one complex requirement statements. In fact, a statement that is considered to be a root in one requirement composition structure may well be a child node in a another composition structure.

7.3.3 REQUIREMENT LINKS

With capturing links across requirements as expressed in item 3 in Section 7.2 above we understand two kinds of links.

- *Requirement composition* establishes relationships from complex requirements to basic ones.
- *Requirement relationships* indicating logical dependencies between requirements, e.g., that a set of requirements are derived from another set of requirement, or to indicate the existence of alternate (mutually exclusive or redundant) statements.

The next two sections presents definitions for requirement link structures:

Requirement composition

Requirement composition is characterised by a directed acyclic graph $G_{composition}$ = (R, C) [54] where each node ($r, r \in R$) in the graph represent a requirement statement and each edge ($c, c \in C$) the composition relationship from a complex parent requirement to a more basic child requirement. A parent requirement has at least one outgoing edge and a child requirement at least one incoming edge. A requirement may be both a parent and a child requirement simultaneously. The *root requirement* in a requirement composition structure is a requirement that has no parent requirement, i.e., no incoming edge.

When requirement composition is used it is assumed that it shall be possible to directly infer a statement of a child requirement from the parent's statement.

Example 6: Requirement composition is illustrated in the graph in Figure 7.1 (top) where *R = {r.1, r.1.1, r.1.2}* and *C = {c1, c2}*. Require-

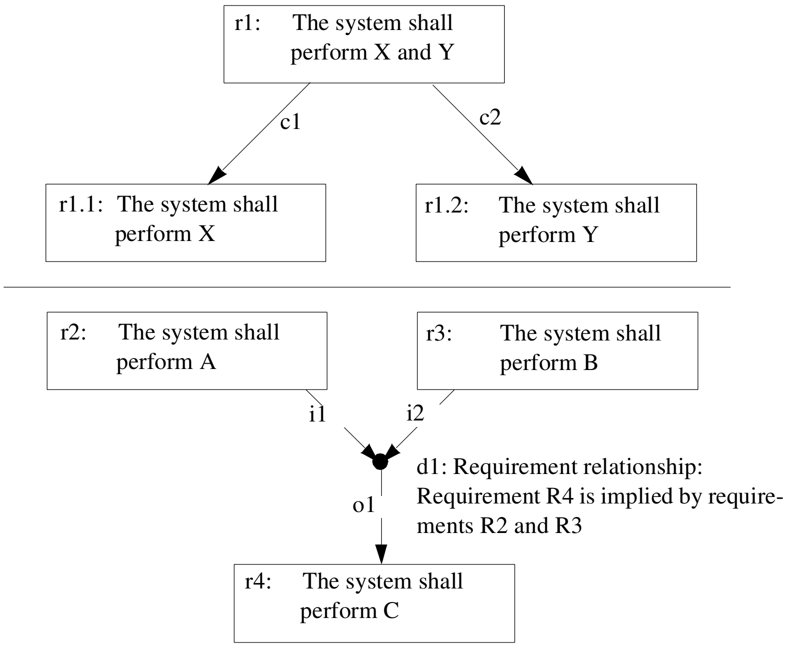

r1: The system shall perform X and Y
c1
c2
r1.1: The system shall perform X
r1.2: The system shall perform Y
r2: The system shall perform A
r3: The system shall perform B
i1
i2
d1: Requirement relationship: Requirement R4 is implied by requirements R2 and R3
o1
r4: The system shall perform C

Figure 7.1: Requirement composition (top) and derived requirement relationships (bottom)

ment R1 is the *root requirement* and requirements R.1.1 and R.1.2 are child requirements of R1 and at the same time requirement R1 is the parent of requirements R.1.1 and R.1.2.

■

We assume that requirement composition is static and essentially system independent. If a requirement *r* is the root requirement in a composition structure then the composition is valid for assignment to any system.

Requirement relationships

Requirement relationships are independent of the requirement composition structure and may involve any number of requirements. Requirement relationships and related requirement objects can be characterised by a bipartite directed acyclic graph $G_{relationship} = (V_r, E_r)$ where V_r consist of two distinct sets of nodes and where E_r consist of two distinct sets of edges between the nodes. The two sets of vertices are the requirements (R) and requirement relationships (D). Edges are either *relationship input edges* (I) or *relationship output edges* (O):

$I \subseteq R \times D$	The elements in *I* corresponds to the edges from requirements to requirement relationships.
$O \subseteq D \times R$	The elements in *O* corresponds to the edges from requirement relationships to requirements, indicating the requirements where created or modified as a result of the relationship

Requirement relationships typically involve more than one *relationship input edges* and thus involve more than one requirement and is system dependent, i.e., a relationship can only exist if all involved requirements are assigned to the same system. Figure 7.1 (bottom) illustrate an example of a requirement relationship graph where V_r = {r2, r3, r4, d1} and E_r = {i1, i2, o1}.

7.3.4 REQUIREMENT PROPERTIES

In addition to capturing an expression of need or demand there is supplemental information that may be associated with a requirement, e.g., administrative information such as authorship, ownership of a requirement, or evaluation or trade-off data such as requirement priority, or comments on and feasibility or risk assessments of a requirement. This is information that may be required in some methods, but not considered at all in others. Consequently it would be inappropriate to mandate a fixed set of attributes for a requirement. Instead the information model mandates a small set of attributes for a requirement and allow for the assignment of any number of properties. For instance, the minimal attribute set for a textual requirement may be:

- Description, a textual definition of what is required
- Id, a unique identifier for identification of requirements stored in multiple databases
- Presentation identifier, an identifier for user identification of a requirement. For instance, the presentation identifier of a requirement may be

7.2.3, indicating it is the third child requirement of the second child of the seventh root requirement for a system.
- Name, a short string that identifies the requirement

Additional information is captured using general or requirement specific representations for properties that is assigned to individual requirements. The information model structure is a consequence of the requirements presented in Chapter 4. The range of potential requirement properties include:

- Requirement prioritisation data with regard to a specific criterion, absolute or relative to other requirements.
- Requirement rationale and motivation data.
- Selected method for requirement verification.
- Requirement source, ownership and list of interested stakeholders.
- Assessment of requirement stability, completeness and risk.
- Requirement approval status.
- Requirement comments and queries.
- Costs associated with implementing a requirement.

Any number of specific properties types listed above may be mandated or supported in specific requirement engineering methods, but it is unlikely that all properties will be mandated by a single method. Moreover, in many cases there are multiple variants to a property to consider. For instance, a method may allow capture of multiple priorities for a requirement, e.g., multiple stakeholder specific priorities.

7.3.5 REQUIREMENT PROPERTY VOLATILITY

The values of requirement properties are assumed to be more volatile than the actual requirement statement. The requirement statement, or a requirement composition structure may be fixed and agreed upon, but, e.g., the perceived importance of an individual requirement to a system in terms of priority may vary over time. This is especially the case when a single requirement statement is common to multiple system specifications. Consequently a separation is made between the representation the static requirement structure and the representation of system specific properties.

7.3.6 REQUIREMENTS CLASSIFICATION

Requirement classification or categorisation is often considered in requirements and Systems Engineering literature as a mean for providing a better overview of the requirements on a system [16]. A large number of classification schemes have been proposed. The IEEE-1220 standard identifies four classes of requirements (Functional, Operational, Physical and Constraints) [67], Wymore identifies six distinct classes [155], while Grady [55] and Oliver [114] identifies five mutually inconsistent classes. The absence of consensus in the Systems Engineering community implies that an open framework for requirement classification is preferable for data exchange to a single set of predefined classes. This is also the assumption in this thesis

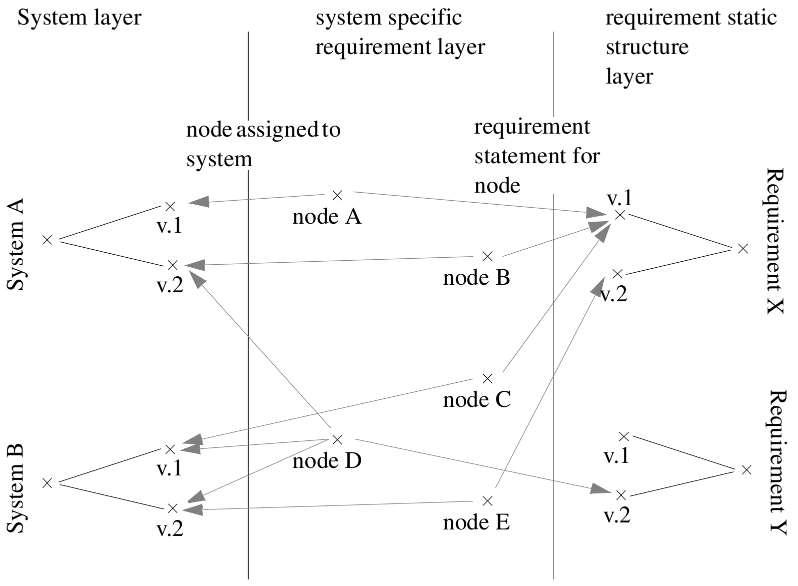
7.4 Presentation of the Requirements Representation Information Model

Two requirements representation structures can be identified. First of all there is the requirement product structure defining the decomposition of complex requirements to more basic ones. This structure is assumed to be reasonable static and system independent. Moreover, the static structure may be common to multiple systems/projects or versions thereof.

The second facet of requirement representation is related to how a requirement is represented within the scope of a single system or a comparable small set of versions of a system. This facet encompasses relationships to other requirements, properties local to a system or a small set of systems.

The requirement information model is implemented in two layers to capture both the static requirement product structure such that it may be shared across multiple system specifications and the placeholder for volatile system specific requirement properties. The layers are illustrated in Figure 7.2[1] and presented further below.

1. Actually the figure contains three layers to indicate that requirements are assigned to systems.



Configurations illustrated:
Requirement X, version 1 is assigned to System A, version 1 and 2 with different system specific properties (via node A and B)
Requirement Y, version 1 is not assigned to any system specification
Requirement Y, version 2 is assigned to System A, version 1 and System B, version 1 and 2 with common system specific properties (via node D)
Requirement X, version 1 is assigned to System B, version 1 (via node C) and Requirement X, version 2 is assigned to System B, version 2 (via node E)

Figure 7.2: Layered approach to representing requirements and systems in the information model

- The *requirement static structure layer* captures individual requirements, the requirement composition structure and provides structures for version management of individual requirements. The entities for defining the requirement static structure layer are defined such that an individual requirement statement can be part of multiple requirement composition structures while maintaining a local composition structure for each composition.

- The *system specific requirement layer* support the identification of requirement nodes that identifies a specific requirement. Multiple nodes may be assigned to a single requirement to support capture of properties that are local to a limited set of systems.

The representation of versions is supported for requirements and system architecture objects. The system specific requirement layer defines the requirement configuration for each system. A number of possible configuration alternatives are illustrated in Figure 7.2.

The information model for the requirement static structure layer is presented first followed by the model for the system specific requirement layer and the mechanism for associating a requirement to the system architecture element (presented in Chapter 6).

7.5 The Requirement Static Structure Information Model

This section presents the selected representation for capture the requirement static structure layer in the information model.

The basic representation structure for the requirements product structure is similar to that for representing system life-cycle views as presented in Figure 6.3. The *configuration_element* and *configuration_element_version* entities are shared with all other concepts modelled with version management support. The attribute *configuration_element_type* of the entity *configuration_element* shall be set to ‘requirement’ for a requirement configuration element. The entities for defining the requirement product structure is presented in Figure 7.3 and presented below.

- *Requirement_definition*, abstract supertype that captures the actual requirement statement. Representation specific requirement statements is captured in the three subtypes *textual_requirement_definition* — for textual requirements, *structured_requirement_definition* — for requirements expressed as a value interval and measurement unit com-

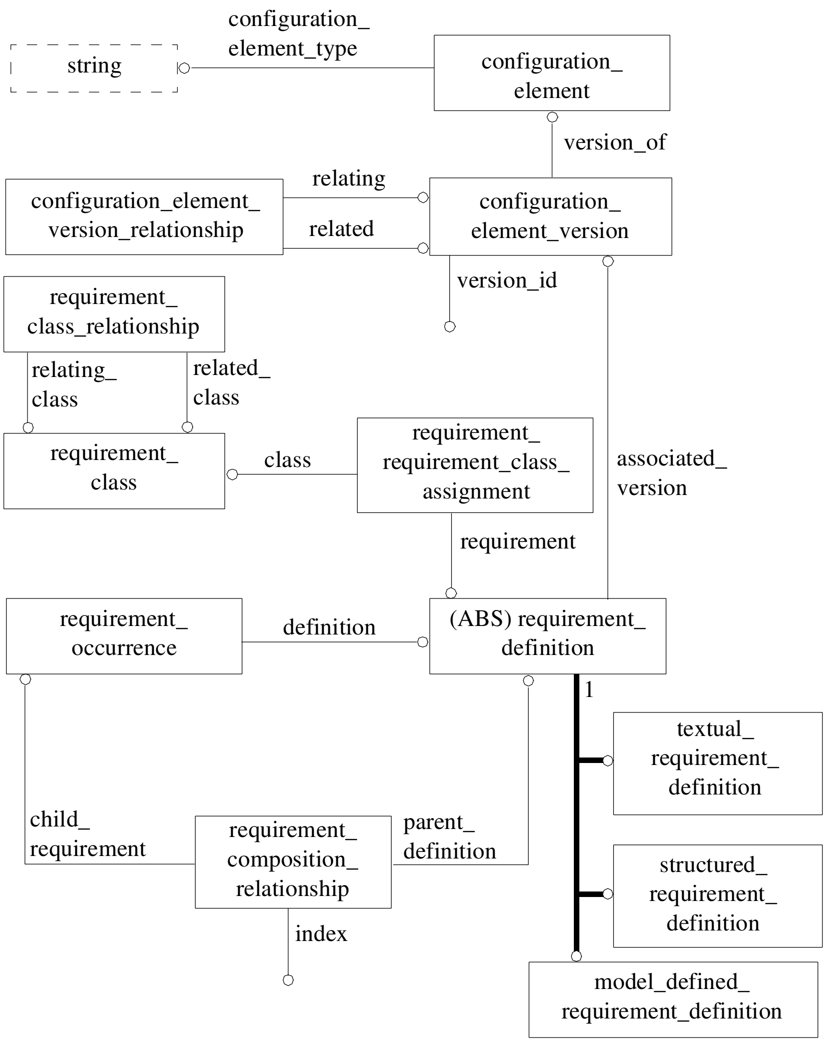

string
configuration_ element_type
configuration_ element
version_of
configuration_element_ version_relationship
relating
related
configuration_ element_version
version_id
requirement_ class_relationship
relating_ class
related_ class
requirement_ class
class
requirement_ requirement_class_ assignment
associated_ version
requirement
requirement_ occurrence
definition
(ABS) requirement_ definition
1
textual_ requirement_ definition
structured_ requirement_ definition
model_defined_ requirement_definition
child_ requirement
requirement_ composition_ relationship
parent_ definition
index

Figure 7.3: Requirement product structure

bination and *model_defined_requirement_definition* — for requirements expressed in models or documents whose format is supported by the information model.

- *Requirement_occurence*, represents the use of a particular requirement in a requirement composition structure. This entity is motivated by a desire to unambiguously identify the parent requirement in the case a *requirement_definition* object is a child in multiple requirement composition structures.
- *Requirement_composition_relationship*, defines the parent child relationship between a requirement and an immediate child requirement. In addition the *index* attribute captures a section of the presentation identifier of the child requirement in the context of the parent requirement. Requirement composition structures shall not contain any cycles.
- *Requirement_class*, provides the mechanism for classifying a requirement according to content. The information model does not define a predefined classification schema as there is no consensus in the Systems Engineering community in the issue [16]. Instead the framework allows user defined sets of requirement classes. Harmonisation of classes used is a prerequisite for effective exchange of this class of data.
- *Requirement_requirement_class_assignment*, defines that a particular *requirement_definition* object is assigned as member to a *requirement_class* object. A requirement may be assigned to many *requirement_class* objects, but may only be assigned once to a specific *requirement_class*.
- *Requirement_class_relationship*, captures a relationship between a pair of *requirement_class* objects. The motivation for including the entity is to capture relationships between heterogeneous requirement classification schemes. Two relationship types are predefined. The *equivalence* relationship indicates that the definitions of the *requirement_class* objects related are considered equal and the *specialisation* relationship indicates that the definition of *related_class requirement_class object* is more restricted than the definition of the *relating_class requirement_class* object.

Example 7: Consider a schema where requirements are classified as being either functional or non-functional and another schema with four requirement classes: functional, operational, performance, and constraints. In this example the *requirement_class_relationship* could be used to indicate that the performance requirement class in the second schema is a *specialisation* of the non-functional requirement class in the first schema.

■

Note, the entities defined in Figure 7.3 captures requirement statements, composition structure and classification but do not associate a requirement to a particular system.

7.5.1 REQUIREMENTS COMPOSITION

This section provides examples on how the requirement composition structure is realised using the entities defined in the information model. The model extends on the definition of requirement composition presented in Section 7.3.3. A requirement statement as captured by the *requirement_definition* entity is self-sufficient and may be inferred from more than one complex requirement statement, i.e., a requirement statement may be derived from multiple sources. A requirement X may be part of the decomposition of the complex requirement ZX as well as the complex requirement YX. Moreover, the requirement X may also be considered a root requirement in the context of one system.

In the information model requirement composition structure is built using the three entities: *requirement_definition*, *requirement_occurrence* and *requirement_composition_relationship*. The selected structure allows each requirement composition structure to be expressed independently, even if the same *requirement_definition* object occurs in multiple composition structures. Two instantiation patterns are presented in Figure 7.4 and Figure 7.5 to illustrate the use of the information model.

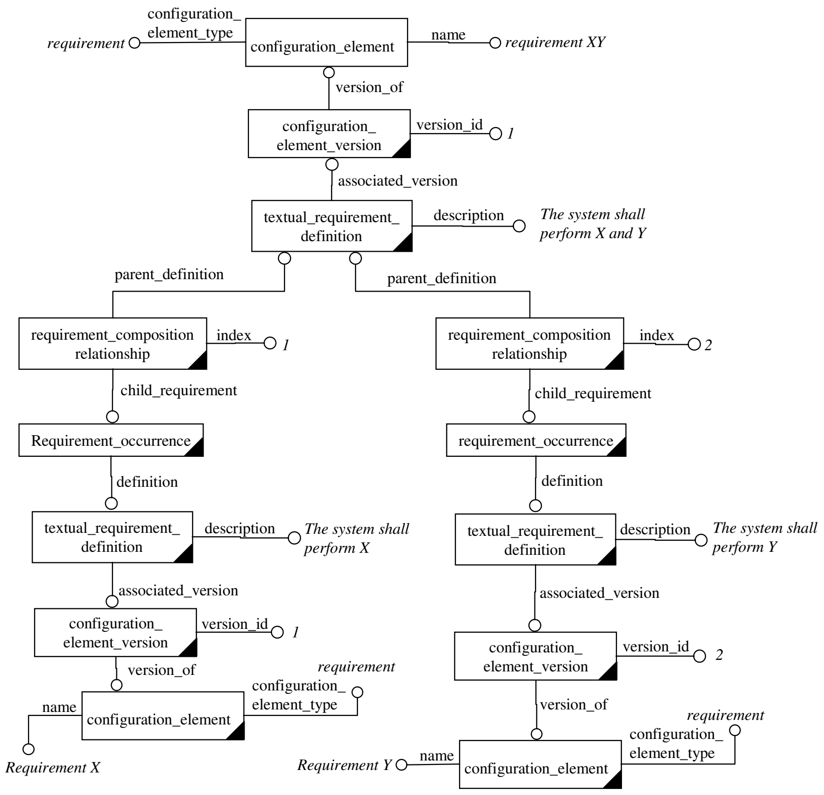


Figure 7.4: Requirement composition example

Figure 7.4 illustrates the instantiated data structures for the requirement composition example presented in Figure 7.1 (top). A complex textual requirement is broken down into a pair of more specific ones. Version 1 of *requirement XY* presented in Figure 7.4 is statistically composed of version 1 of *requirement X* and version 2 of *requirement Y.*

The *index* attribute of *requirement_composition_relationship* objects defines the identifier of each child requirement in the context of the parent requirement definition. The *index* attribute of the left hand requirement in Figure 7.4 is set to 1 in the composition. The global presentation identifier for an individual requirement is computed through a top-down traversal of

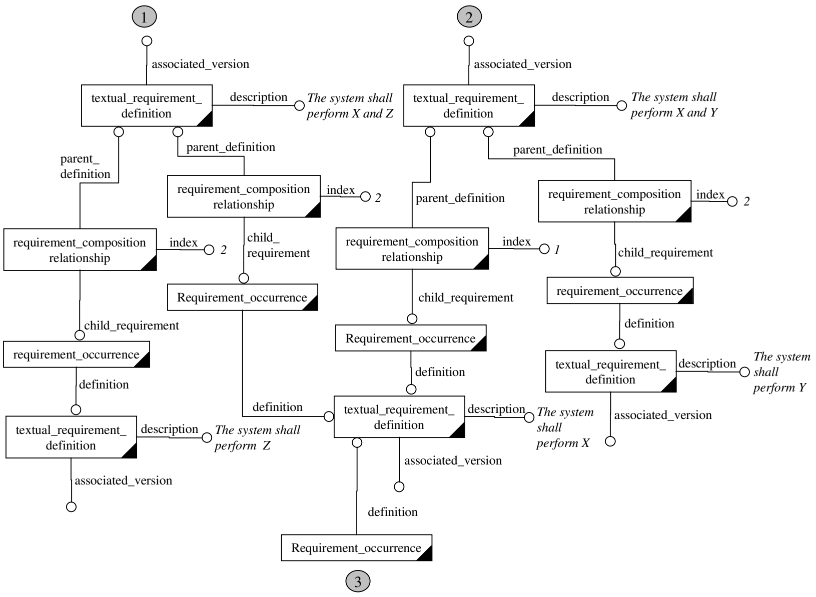


Figure 7.5: 3 requirement compositions with one common requirement statement

the requirement composition structure until the requirement is reached. For instance, if top level requirement in Figure 7.4 is the 7th for a particular system life-cycle view or viewpoint then the global presentation identifiers of the child requirements are 7.1 and 7.2 respectively.

The example presented in Figure 7.5 illustrates three requirement composition structures sharing a common *requirement_definition* object, i.e., the same requirement statement is common to three composition structures. This could occur in cases where there exist an overlap between requirements on the same system or the case could be that the requirement structure are shared through the individual requirements that apply to different systems. The requirement structure indicated by label 1 in Figure 7.5 state that requirement XZ is composed of requirements X and Z. The structure indicated by label 2 state that requirement XY is composed of requirements X and Y. The structure indicated by label 3 state that requirement X is a top-level requirement in a particular context.

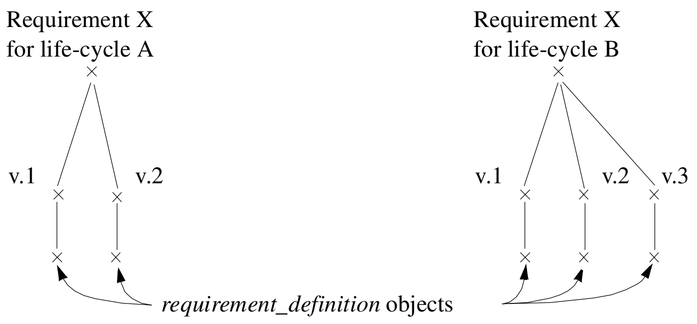
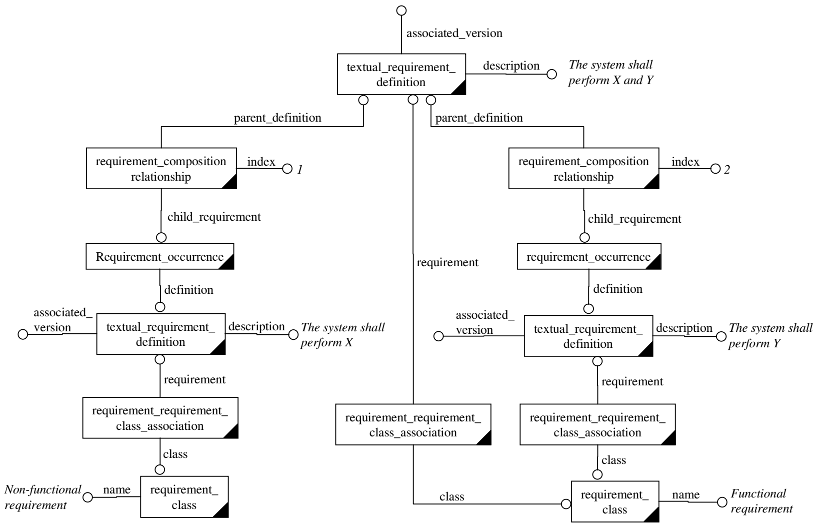


Figure 7.6: Representation of requirements for the same domain but for different system life-cycles

The *requirement_occurrence* entity provides a placeholder for system specific views on requirement composition structures. Requirement X is a root requirement from the perspective of label 3 in Figure 7.5. At the same time requirement X is a child requirement from the perspective of the *textual_requrement_definition* objects at labels 1 and 2 in Figure 7.5. The use of different *requirement_occurrence* objects to reach requirement X in each composition structure allow for a structure where requirement X is seen as being used in isolation in respective composition structure. Properties, such as a motivation for inclusion in respective structure could be captured on each individual *requirement_occurrence* object.

7.5.2 MANAGEMENT OF REQUIREMENTS FOR MULTIPLE SYSTEM LIFE-CYCLES

The model for version management of requirements is identical to that of systems with the exception that only one *requirement_definition* object (life-cycle view) per *configuration_element_version* of a requirement is allowed, i.e., only one requirement statement is allowed per requirement version. The motivation for this restriction is that no reference in the literature or in industrial practice was found where multiple requirement life-cycles are associated with each requirement version. Multiple *configuration_element* objects shall be instantiated when there are multiple related requirement statements that applies to different system life-



associated_version
textual_requirement_ definition
description
The system shall perform X and Y
parent_definition
parent_definition
requirement_composition relationship
index
1
requirement_composition relationship
index
2
child_requirement
child_requirement
Requirement_occurrence
requirement_occurrence
requirement
definition
definition
associated_ version
textual_requirement_ definition
description
The system shall perform X
associated_ version
textual_requirement_ definition
description
The system shall perform Y
requirement
requirement
requirement_requirement_ class_association
requirement_requirement_ class_association
requirement_requirement_ class_association
class
class
class
Non-functional requirement
name
requirement_ class
requirement_ class
name
Functional requirement

Figure 7.7: Requirement classification example

cycles as indicated in Figure 7.6. This restriction is included to ensure consistent usage of version management structures for all requirement objects.

7.5.3 REQUIREMENT REPRESENTATION VS. THE REQUIREMENT COMPOSITION STRUCTURE

No implicit assumption is made on the suitability of a requirement captured using a particular requirement representation and its position in a requirement composition structure. For instance, a complex requirement could be expressed in a document and its immediate children requirements could be expressed in natural language, or vice versa.

7.5.4 REQUIREMENT CLASSIFICATION

The requirement classification part of the information model support classification of requirements for grouping of requirements with common characteristics. Classification is subject to a users interpretation of individual requirement statements and is not mandated. A requirement statement may be assigned to multiple requirement classes. Requirement classification is related to the *requirement_definition* entity. This is based on the assumption that classification for a particular classification schema, once performed, will remain stable over the life of a version of a requirement.

The use of the requirement classification structure is illustrated in Figure 7.7. Two requirement classes are identified in the figure: *functional* and *non-functional* requirements and the three *textual_requirement_definition* objects in Figure 7.7 are classified. Note that the model does not mandate classification information and that a single requirement may be classified to multiple classes. The assignment of a requirement to multiple classes may reflect the situation where unambiguous classification could not be made.

7.6 System Specific Requirement Information Model

This section introduces the information model structures for associating a requirement to a system and for capturing requirement relationships and properties local to a system. The entities presented in this section define the interface between the system architecture model presented in Chapter 6 and the static requirement structure introduced in Section 7.5.

The system specific requirement information model serves three purposes:

1. It provides the mechanism for assigning a requirement to a system specification
2. It provides the structure for capturing properties that are valid for a requirement when associated to a specific system or a set of systems.
3. It provides the structure for capturing relationships across requirements.

The information model portion is presented in Figure 7.8. The entities *requirement_instance*, *requirement_system_view_assignment, child_requirement_system_view_assignment* and *root_requirement_system_view_assignment* defines the mechanism for relating an individual requirement statement to a system. The *requirement_instance* entity cap-

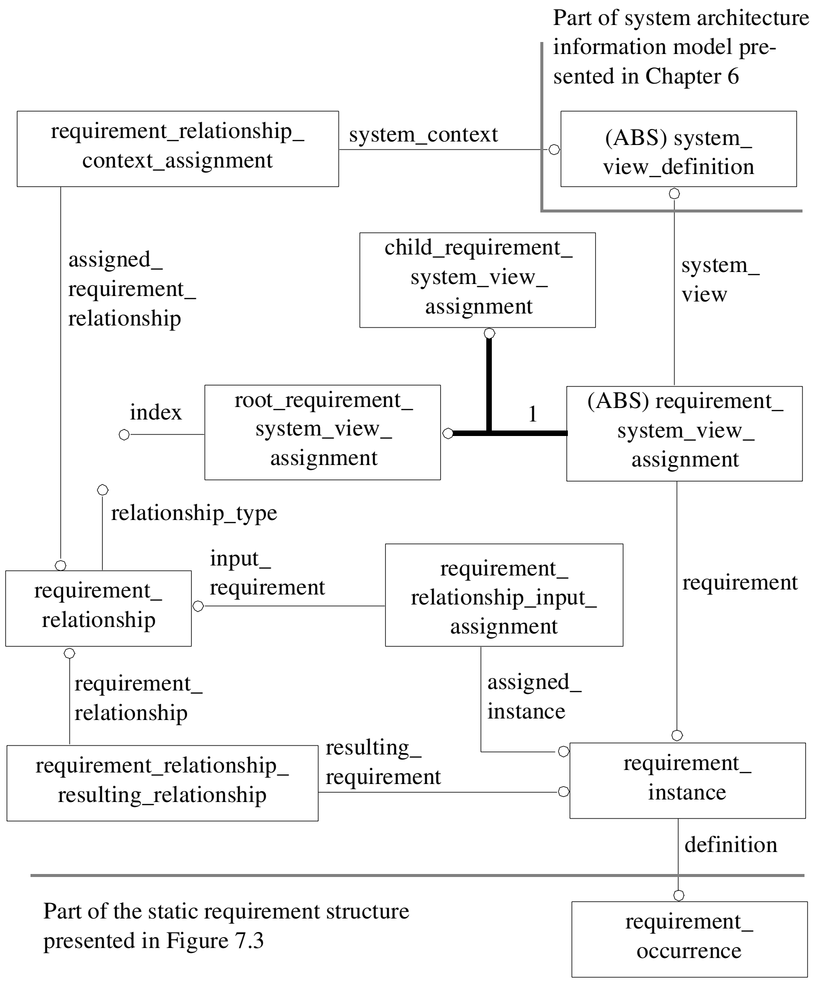

Part of system architecture information model presented in Chapter 6
requirement_relationship_context_assignment
system_context
(ABS) system_view_definition
assigned_requirement_relationship
child_requirement_system_view_assignment
system_view
index
root_requirement_system_view_assignment
1
(ABS) requirement_system_view_assignment
relationship_type
input_requirement
requirement_relationship
requirement_relationship_input_assignment
requirement
requirement_relationship
assigned_instance
resulting_requirement
requirement_relationship_resulting_relationship
requirement_instance
definition
Part of the static requirement structure presented in Figure 7.3
requirement_occurrence

Figure 7.8: System specific requirement information model

tures system specific requirement properties, and the entities *requirement_relationship*, *requirement_relationship_input_assignment*, *requirement_relationship_resulting_relationship* and *requirement_relationship_context_assignment* provide the structures for relating requirements to each other within the scope of a system life-cycle view or viewpoint.

The definitions of individual entities are presented below followed by a detailed discussion on how entities may be instantiated to represent real requirement data.

- *Requirement_instance* is the placeholder for capture of system specific requirement properties.
- *Requirement_system_view_assignment* is the abstract supertype for assigning a *requirement_instance* object to a *system_view* object. A requirement may only be assigned to a *system_view* object once. The subtypes *root_requirement_system_view_assignment* and *child_requirement_system_view_assignment* perform the assignment for root requirements and child requirements respectively. The definition and motivation for these two entities are presented further in Section 7.6.2.
- *Requirement_relationship* captures the existence a relationships between two or more requirements. The semantics of a *requirement_relationship* is defined by its *type* and *description* attributes. All requirements in the relationship must be assigned to the same *system_view* object.
- *Requirement_relationship_input_assignment* indicates that a *requirement_instance* object is somehow related to other requirement statements. Any number of *requirement_relationship_input_assignment* objects may be related to a *requirement_instance* object, but any *requirement_instance* object may only be assigned once to specific *requirement_relationship_input_assignment* object.
- *Requirement_relationship_resulting_relationship* indicates that a *requirement_instance* object is implied by a *requirement_relationship* object. Any number of *requirement_relationship_resulting_relationship* objects may be related to a *requirement_relationship* object, but a *requirement_instance* object may only be related once by a *requirement_relationship_resulting_relationship* object.
- *Requirement_relationship_context_assignment* relates a *requirement_relationship* object to a *system_view* object such that the *requirement_relationship* is valid in the *system_view*. If the assignment is not

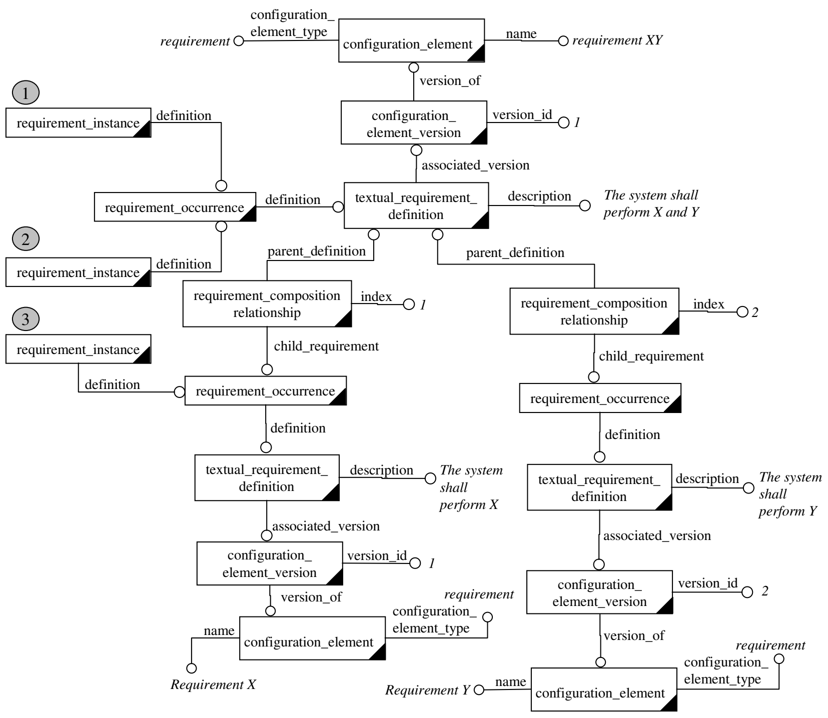


Figure 7.9: *Requirement_instance* objects relating a single *requirement_occurrence* object

present for a particular *system_view* object then the relationship shall not be considered within that view.

7.6.1 SYSTEM SPECIFIC REQUIREMENT REPRESENTATION

The purpose of the *requirement_instance* entity is to provide a placeholder where system specific properties for a requirement can be captured. In this sense the *requirement_instance* is the interface to the static requirement structure via its definition attribute to the *requirement_occurrence* entity and from there to the *requirement_definition* entity that captures the actual requirement statement. Any number of *requirement_instance* objects may relate to a *requirement_occurrence* object as illustrated in Figure 7.9. The *requirement_instance* objects labelled 1 and 2 in Figure 7.9 both refer to the same *requirement_occurrence* object. System specific properties may be assigned to each of these objects. For instance, the *requirement_instance* labelled 1 may be assigned properties indicating high priority and low risk, while the *requirement_instance* labelled 2 may be assigned a property indicating that the requirements is considered to be of low importance. Label 3 illustrates that *requirement_instance* objects may be assigned to any *requiremet_occurrence* object regardless of its position in a requirement composition structure.

7.6.2 ASSIGNMENT OF REQUIREMENTS TO SYSTEMS

The *requirement_system_view_assignment* provides the mechanism for assigning a *requirement_instance* object to a *system_view* object. The entity is abstract with two sub-types.

- *Root_requirement_system_view_assignment*, assigns a *requirement_instance* object that refers to a requirement which is the root of a requirement composition structure to a *system_view*. All properties of the *requirement_instance* object referenced and all of its child requirements are implicitly assigned as well. However, for composite requirements no requirement properties, except for those recorded on the top-level requirement are captured. The *root_requirement_system_view_assignment* also defines the top-level element in the presentation identifier of the requirement within the scope of the *system_view* via its *index* attribute.
- *Child_requirement_system_view_assignment*, assigns a *requirement_instance* object that refers to a *requirement_occurrence* object which is a child requirement in an arbitrary deep composition structure to a *system_view* object. The *child_requirement_system_view_assignment* is motivated as any requirement in a composition structure may have system specific properties assigned.

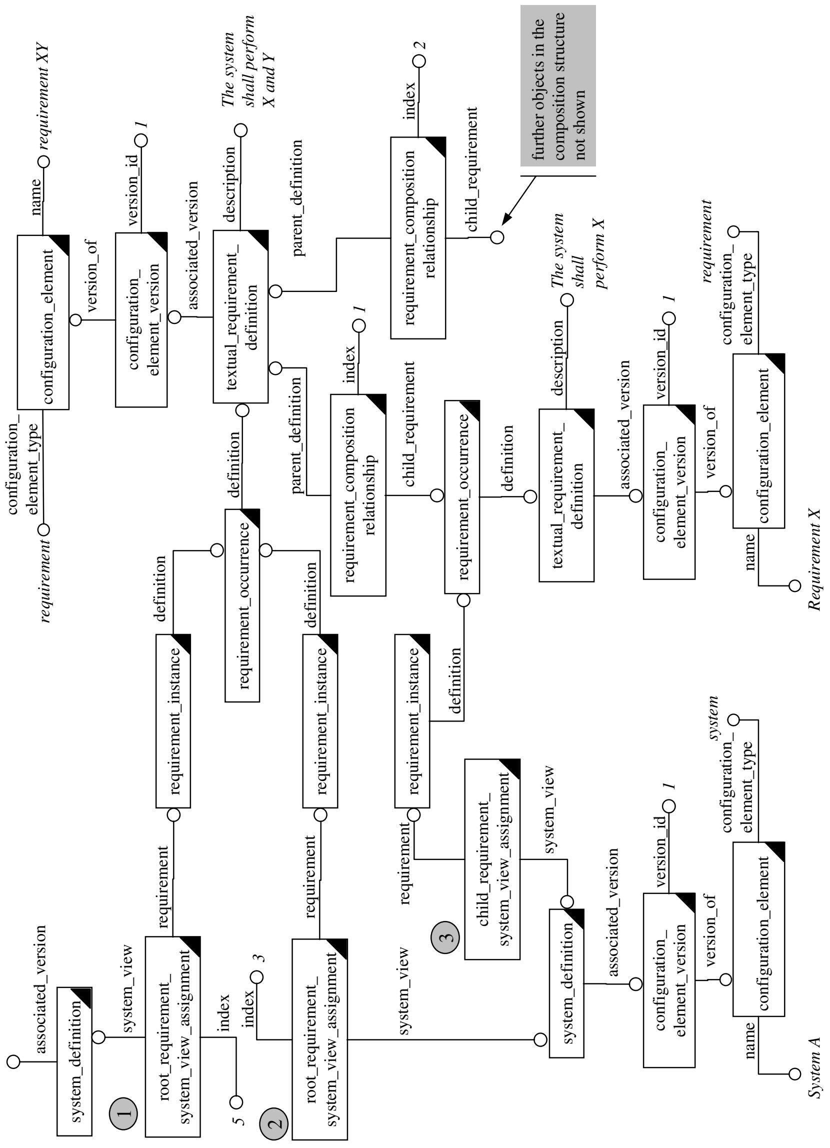

system_definition
associated_version
root_requirement_system_view_assignment
system_view
index
5
requirement
requirement_instance
definition
requirement_occurrence
configuration_element
configuration_element_type
requirement
name
requirement XY
configuration_element_version
version_of
version_id
1
textual_requirement_definition
associated_version
description
The system shall perform X and Y
parent_definition
requirement_composition_relationship
index
2
child_requirement
further objects in the composition structure not shown
root_requirement_system_view_assignment
index
3
system_view
child_requirement_system_view_assignment
system_view
requirement_occurrence
textual_requirement_definition
The system shall perform X
configuration_element
name
Requirement X
System A
system
1
3
2

Figure 7.10: Requirement assignment to system specifications

The entity *child_requirement_system_view_assignment* may only be used in cases where the root requirement of the composition structure is already assigned to the same *system_view* object via a r*oot_requirement_system_view_assignment* object.

The usage of the entities is further illustrated in Figure 7.10 that is an extension of Figure 7.9. Objects representing the system architecture part of the information model are present in the lower part of Figure 7.10. The *root_requirement_system_view_assignment* object at Label 1 in Figure 7.10 indicate that Requirement XY is assigned as top-level requirement number 5 to the *system_definition* object in the lower right hand corner.

The *root_requirement_system_view_assignment* object at label 2 in Figure 7.10 indicate that Requirement XY is also captured as requirement number 3 for a life-cycle view of version 1 of system A. If the properties for Requirement XY in the scope of the two systems are different then they are captured at different *requirement_instance* objects as in Figure 7.10. If the properties are identical for the two systems then they could be captured at a single requirment_instance object.

The *child_requirement_system_view_assignment* object at label 3 in Figure 7.10 illustrates how properties local to a *requirement_instance* object representing a requirement which is a child in a requirement composition structure are associated with a system. The *child_requirement_system_view_assignment* can only be used in cases where the top-level requirement of the requirement in the composition structure is already assigned to the same *system_view* object as the identified by the *system_view* attribute of the *child_requirement_system_view_assignment* object. For instance, if the *root_requirement_system_view_assignment* at label 2 did not exist then the *child_requirement_system_view_assignment* object at label 3 would not be valid.

In case the properties captured by the *requirement_instance* referred by the *child_requirement_system_view_assignment* at label 3 also applies to the *system_definition* at label 1 then this could be indicated by assigning a second *child_requirement_system_view_assignment* object to the *requirement_instance* object.

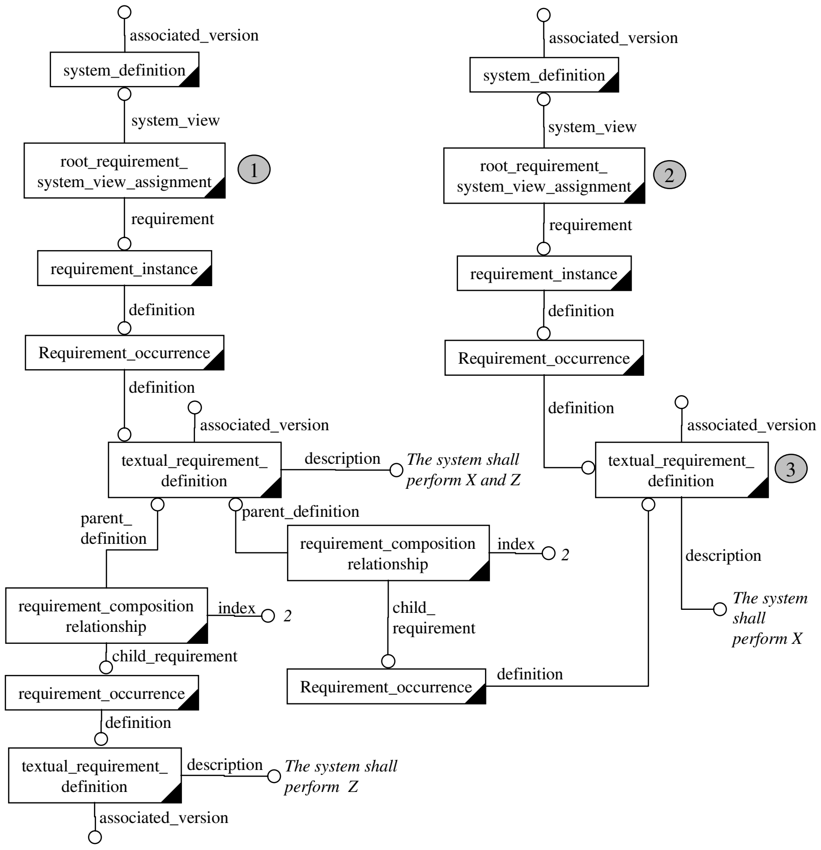


Figure 7.11: Assignment of a requirement as child requirement to the system_view indicated by label 1, and as parent requirement to the system_view indicated by label 2

A further example on requirement assignment to *system_view* objects is presented in Figure 7.11. Two requirements are assigned via the *root_ requirement_system_view_assignment* objects indicated by labels 1 and 2. The *textual_requirement_deifinition* object at label 3 in the figure is assigned to both systems. However, composition specific views can be defined with *requirement_occurrence* objects. A top-down traversal of from label 2 will not indicate that the requirement at label 3 is part of multiple composition structures.

7.6.3 REQUIREMENT RELATIONSHIP

A requirement relationship could indicate any kind of relationship between requirements. In the information model the attribute *relationship_type* of the *requirement_relationship* entity define the semantics of the relationship. Any textual string could be used to define the relationship. However, two types of requirement relationships were identified in the AP-233 working group and defined in the standard documentation:

1. Alternate: The requirements associated via *Requirement_relationship_input_assignment* objects to the *Requirement_relationship* are mutually exclusive or equivalent;
2. Derived: The requirements associated via *requirement_relationship_input_assignment* objects are used to derive additional *requirement_instance* objects. Requirements identified through the relationship are indicated using *resulting_relationship* objects.

The definitions of the above two relationships are proposed within the information model documentation. However, any other term could be used for identifying relationship type. The motivation for not defining more relationships is that consensus could not be reached for any more relationship types. Still agreement on the definition of a limited set of relationship terms appears appealing for effective data exchange between organisations. However, this must be agreed for each project where data exchange is considered.

7.6.4 REQUIREMENT RELATIONSHIP EXAMPLE

The information model Figure 7.12 illustrate how requirement relationships are captured in the information model. Two *requirements_instance* objects (labelled 3 and 4) are both assigned to a pair of *system_definition* objects (labelled 1 and 2). The *requirement_relationship* (label 5) of relationship type alternative indicate that the requirements are found to be mutually exclusive or equivalent. This relationship can be assumed to hold for all systems where both requirements are assigned. However, in a configuration managed environment it is important to capture for which systems the requirement relationship is valid. A relationship may be valid for

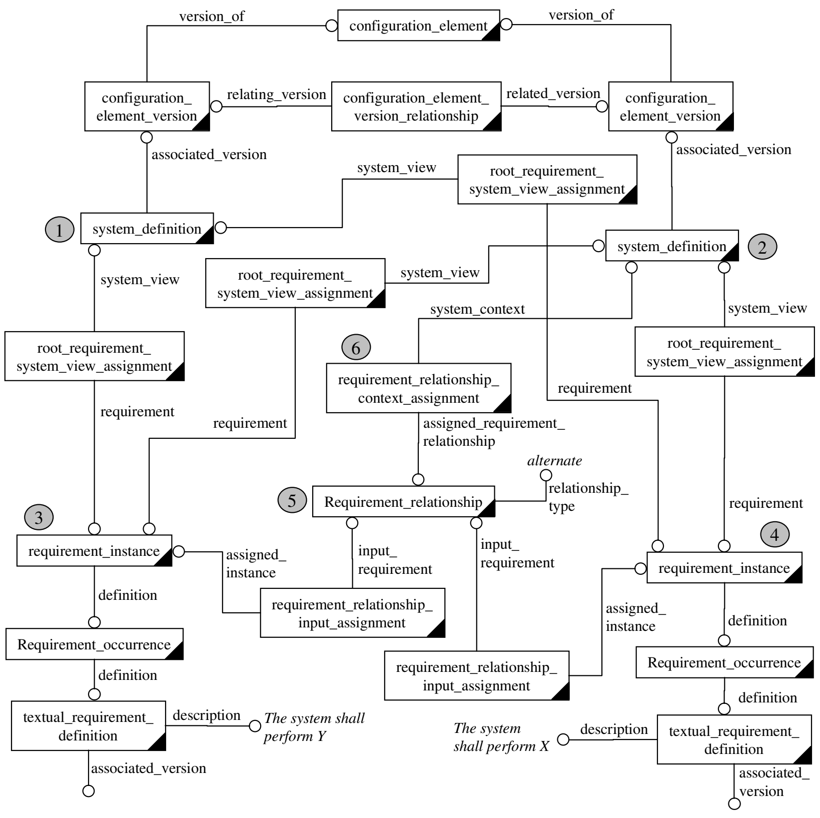


Figure 7.12: Alternate Requirement relationship example

a single or limited number version of a specification only — indicating that a relationship was not initially identified, or incorrectly captured. In Figure 7.12 the *requirement_relationship_context_assignment* object (label 6) indicate that the *requirement_relationship* is only valid in the context of the *system_definition* object at label 2. A second *requirement_relationship_context_assignment* object would be required to make the *requirement_relationship* valid for the *system_definition* object at label 1 in Figure 7.12.

7.7 Summary

In this chapter the information model structures for representing requirements and requirements assignment to system life-cycle views and viewpoints has been described and motivated. The structures are flexible such that a requirement statement may be part of multiple composition structures and requirement properties are captured independent of the actual requirement statement.

Requirements are not isolated from other specification elements of the information model. Chapter 11 introduces the model elements for establishing traceability relationships between requirements and other specification elements.

Chapter 8 Functional Architecture

This chapter presents the information model for representing system functional architecture and the methods considered when implementing the information model. The interface between the system architecture and the functional architecture parts of the information model is also presented.

8.1 Introduction

In IEEE-1220 [126] the term *functional architecture* is defined as:

> An arrangement of functions and their subfunctions and interfaces (internal and external) that defines the execution sequencing, conditions for control or data flows, and the performance requirements to satisfy the requirements baseline.

Note that the definition of function does not conform to the strict mathematical definition. Instead the word *function* shall be interpreted as an element of the overall the *functionality* that a system shall exhibit. The system functional architecture is considered to be an integral part of the system specification in IEEE-1220 and other Systems Engineering literature, e.g., [24] [101] [114] [139].

The functional architecture for a system is a result of an analysis of the functionality a system shall exhibit in each life-cycle phase. Traceability links between requirement statements and the elements of the functional architecture and from elements in the functional architecture to the physical items the functionality is allocated to may be captured. The information model elements for capturing relationships between the functional architecture model and other model elements is presented in Chapter 11.

A number of alternate terms have been proposed to functional architecture. Hatley and Pirbhai [62] uses the term 'Process model', in Oliver et al. [114] the term 'Behaviour' is used, 'Parallel composition' is used by Nissanke [108], and Harel and Politi [61] use the term 'Conceptual model'.

8.1.1 METHOD OVERVIEW

A large number of diagrammatic methods have been proposed to capture the functional architecture of a system. Some of these have been introduced as Systems Engineering methods from the outset while others been adapted from other domains.

Methods from the first category include work by Alford [2] [4] and Long [95] that led to the definition of Functional Flow Block Diagrams (FFBD) and Extended Functional Flow Block Diagrams (EFFBD) as implemented in tools like Holagent RDD-100 and Vitech CORE. The tool CORE described in [38] also implement similar capabilities.

Methods from the second category include Real-Time Structured Analysis methods defined by Hatley and Pirbhai [62] [63] and the related methods by Ward and Mellor [152] [153], approaches using activity charts and Statecharts by Harel [58] and Harel and Politi [61], Blanchard and Fabrycky [24], and Lewis and Wagenhals [91] promote the use of the IDEF0 method [5]. In addition, flowchart approaches originating from automatic control, e.g., as instantiated in the Simulink software package, are also in use by systems engineers.

The listed methods all share a number of high-level concepts but there is substantial heterogeneity in the detailed semantics. The aim when capturing different semantic concepts in the information model is to be fair in support for all methods identified as being relevant as outlined in Section 4.3.

The reviewed methods have been defined to different levels of formality. For instance, the semantics of Statecharts and activity charts as implemented in the Statemate Magnum tool has been defined in detail in, e.g., [60] [120]. For most other methods, e.g., Real-Time structured analysis [62] there exist only informal textual definitions of the methods and the semantics of their constituting elements.

Recently there has been a lot of interest in using object-oriented methods for capturing the functional architecture of a system [34] [97]. The inclusion of these in the information model falls outside the scope of the work presented in this thesis. This is, however, not an indication that we find object-oriented methods unsuitable for use in the Systems Engineering process.

8.2 Overview of Method Concepts

This section presents the criteria selected to compare engineering methods for capturing system functional architecture. The basic characterisation aspects are taken from Buede [25] where four aspects of functional architecture models are considered.

1. The *functional hierarchy* view describing the functions composition structure for a system and the external functions the system is interacting with.
2. The *function interaction* view describing the flow of data or items between individual functions.
3. The *processing* or *function instruction* view describing the algorithm for transformation function input to output, activation and termination conditions for individual functions and capture of system variant function properties.
4. The *control flow* or *behaviour* view describing valid sequences of function activation.

Note that individual methods need not support all functional architecture aspects listed.

In the following sections the four criteria are used to describe different semantic definitions of elements in a functional architecture specification.

8.3 Functional Hierarchy

The functional hierarchy aspect considers how functionality is represented and the mechanism for functional composition. Two types of functional elements can be identified in methods defined to capture functional architecture:

1. *Functions* represent decomposable nodes in the functional architecture. A function represents some kind of action or transformation performed by a system. A function may be composed of other functions or be a terminal node in the hierarchy. Alternate terms used to define the decomposable elements on the functional architecture include 'activity' [58] and 'process' [62].
2. A set of nodes representing *auxiliary functionality*. These are terminal nodes in the functional architecture with a defined semantics. Each method defines its own set of auxiliary functionality.

Formally a generic functional hierarchy model, discounting any functional interaction is a tree defined by the triplet:

$$FH = (F, \alpha, r)$$

where:

- F is non-empty, finite set of functions
- α is the finite set of auxiliary functionality
- r is the environment or context function in the functional architecture, i.e., the function that includes the system function and any functions or auxiliary functionality external to the system function.

The symbol W denotes the functional elements and is union of functions and auxiliary functionality. $W = \mathrm{F} \cup \alpha$ and $\mathrm{F} \cap \alpha = \emptyset$.

Some method specific examples for auxiliary functionality is presented below:

- In IDEF0, FFBD, EFFBD, CORE and behaviour diagrams the only functional element supported is functions [5] [38] [95], hence $\alpha = \varnothing$.
- In the definition of a data and control flow diagrams in [62] the types of functional elements include processes (functions), context processes (*cp*), data stores (*ds*) and cspec bars (*cb*), hence $\alpha = cp \cup ds \cup cb$. The set of auxiliary functionality is pairwise disjoint.
- In the Statemate implementation of activity charts [61] there types of functional elements include activities (functions), external activities (*ea*), data stores (*ds*), control activities (*ca*) and flow junction connectors (*jc*), hence $\alpha = ea \cup ds \cup ca \cup jc$. The set of auxiliary functionality is pairwise disjoint.

8.3.1 DEFINING THE SET OF AUXILIARY FUNCTIONAL ELEMENTS

The definition method specific functional elements are not homogeneous. A data store as defined by Hatley and Pirbhai [62] may only contain a single element at any time while a data store in Statemate [61] may contain multiple elements and the elements may be accessed as stacks or queues. Likewise the semantics of a *cspec bar* as defined by Hatley and Pirbhai is more restricted compared with that of a *control activity* in Statemate.

The non-functional requirements for the information model presented in Section 4.3 states that there shall exist representations in the information model that captures method specific concepts such that they can be recreated in the original method representation.

Following the guidelines defined in Chapter 4 for the methods identified as being of interest for the work presented in this thesis the set of auxiliary functional elements, α, supported in the information model shall be:

- Data stores
- External functions (terminators)
- Control activities (cspec bars)
- Flow junction connectors

8.3.2 FUNCTIONAL HIERARCHY DEFINED

This section presents a set of functions operating on elements in a functional hierarchy (FH). The functions are used later in this chapter to define rules for function composition and function interaction in the information model.

The function *children:* $F \rightarrow 2^W$ defines for each function its set of children (immediate subfunctions and auxiliary functionality). A function f, $f \in F$ is a *leaf function* if *children* $(f) = \varnothing$, otherwise the function is a *composite function.* The function *children**: $F \rightarrow 2^W$ computes the reflexive-transitive closure of children.

The function *parent:* $W \rightarrow F$ defines the immediate parent function for a functional element. A function f is a *parent* of a functional element w if $f =$ *parent* (w). If $f =$ *parent* (w) then $w \in$ *children* (f).

For each functional hierarchy model there exist a unique function $r \in F$ which has no parent, i.e., $\forall f \in F, \exists r \in F, r \notin children(f)$. This function r is the environment function or context view of the functional architecture model. Every functional element, except the environment function has exactly one parent.

Two more functions are defined for determining the set of parents for a functional element. The function $parent^{+}$: $W \rightarrow 2^F$ computes the transitive closure of the parent function for a functional element, i.e. it recursively traverses the parent functions of a functional element until the environment function is reached. There may not be any cycles in a functional hierarchy model, i.e., $f \notin parent^{+}(f)$. A function f is an ancestor of the functional element w if $f \in parent^{+}(w)$.

The function $parent^{*}$: $F \rightarrow 2^W$ computes the reflexive-transitive closure of parent for a functional element.

For a set of functional elements $X \subseteq W$ the least common ancestor function of X, denoted $lca(X)$ is defined to be the function x, $x \in W$ such that

- $X \subseteq children^{*}(x)$
- for every other $f \in W$, $X \subseteq children^{*}(f)$, it follows that $x \in children^{*}(f)$

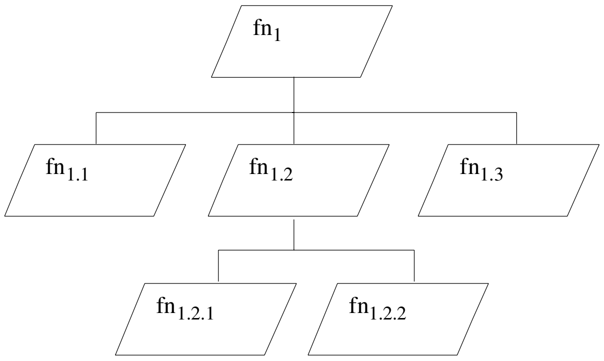


Figure 8.1: Functional hierarchy example

Example 8: The functional hierarchy model in Figure 8.1 is defined by $FH = (F, \alpha, r)$ where:

■

- $F = \{fn_1, fn_{1.1}, fn_{1.2}, fn_{1.3}, fn_{1.2.1}, fn_{1.2.2}\}$
- $\alpha = \{\emptyset\}$
- $r = fn_1$

The hierarchy structure is defined by the children function.

- $\{fn_{1.1}, fn_{1.2}, fn_{1.3}\} = children\ (fn_1)$
- $\{fn_{1.2.1}, fn_{1.2.2}\} = children\ (fn_{1.2})$

The following also applies for the example:

- $\{fn_1, fn_{1.2}\} = parent^+\ (fn_{1.2.1},)$
- $\{fn_1, fn_{1.2}, fn_{1.2.1}\} = parent^*\ (fn_{1.2.1})$
- $fn_1 = lca\ (fn_{1.2.1}, fn_{1.3})$

8.4 Functional Hierarchy Information Model

The model for capturing the functional hierarchy structure is presented in Figure 8.2 and similar to the requirement static structure representation. In the model a distinction between the definition, the use of a function is made with the entities *general_function_definition* and *function_instance*. A comparison with programming language structure gives that a *general_function_definition* object corresponds to the definition of a function, process or task body and a *function_instance* object corresponds to a function call.

The following entities are introduced to represent the functional hierarchy aspect of a functional architecture model.

- *General_function_definition*, the abstract supertype of *composite_function_definition* and *leaf_function_definition*. The *general_function_definition* captures version management and identification information common to the two subtypes. A *general_function_definition* may be referenced by any number of *function_instance* objects. Consequently a *general_function_definition* type object may be used in multiple functional hierarchy models and multiple *function_instance* objects may refer to a single *general_function_definition* object within the same functional hierarchy model.
- *Composite_function_definition*, defines a function which is composed of other functional elements. This is the only entity in the functional architecture model that can be broken down into further elements.
- *Leaf_function_definition*, defines a function definition which is a terminal node in the functional hierarchy structure. The *leaf_function_definition* may be tagged as a predefined function definition from a function library or a general function whose behaviour (algorithm) is described in text. The language used to capture the algorithm may be captured as well.
- *General_functionality_instance*, the abstract supertype of *function_instance*, *persistent_storage*, *fsm_model* and *io_split_join*. The *general_functionality_instance* captures attributes common to all its sub-types and is also important for the part of the model capturing functional interaction presented in Section 8.8. A *general_functionality_instance* object has at most one parent *composite_function_definition* object, i.e., it is referred to as *child_functionality* by at most one *function_composition_relationship* object.

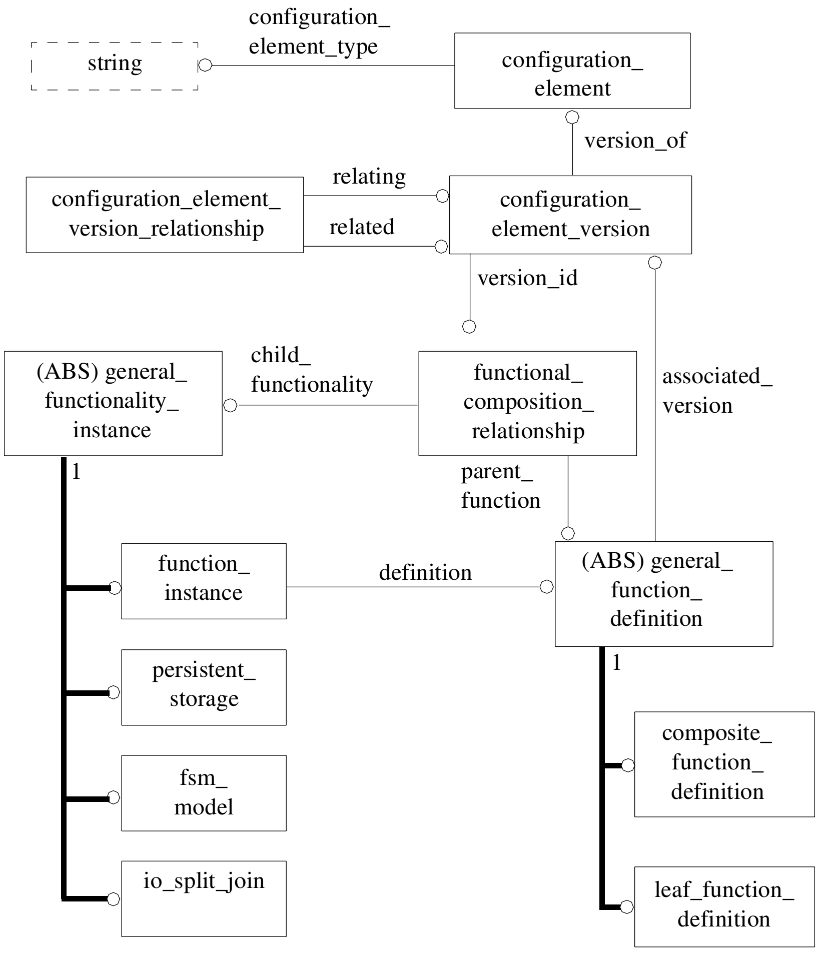

string
configuration_ element_type
configuration_ element
version_of
configuration_element_ version_relationship
relating
related
configuration_ element_version
version_id
(ABS) general_ functionality_ instance
child_ functionality
functional_ composition_ relationship
associated_ version
1
parent_ function
function_ instance
definition
(ABS) general_ function_ definition
persistent_ storage
1
fsm_ model
composite_ function_ definition
io_split_join
leaf_function_ definition

Figure 8.2: Functional hierarchy information model

- *Function_instance*, represent the use of a *general_function_definition* object within a functional hierarchy structure. A *function_instance* is related to exactly one *general_function_definition* via its *definition* attribute.
- *Persistent_storage*, represent a place where functional interaction items accumulate. It is a passive object as it can neither be activated

nor have its own thread of execution. A *persistent_storage* is a terminal in the function hierarchy structure. There are attributes for defining access method and the storage capacity. If known, the access method for accessing items from a *persistent_storage* can be defined to be a stack, queue or random access. In the first two cases reading an item from the *persistent_storage* is assumed to remove the item from the store. The definition of the *persistent_storage* entity is the same as that for a *store* in Statemate [61]. Hatley and Pirbhai [62] have a more restrictive definition as a store may only contain a single item. The storage capacity attribute shall be set to 1 when a data store with the Hatley and Pirbhai semantics is mapped onto a persistent storage object.

- *FSM_model*, is the entry point to a finite state machine. The *FSM_model* is a terminal in the functional hierarchy structure. A *FSM_model* may be used to define activation and deactivation conditions for functional elements and may also produce output based on some algorithm as any other function object. The definition of the *FSM_model* entity correspond to that of a *control activity* in Statemate [61], but also support the more restrictive definition for a *cspec bar* by Hatley and Pirbhai [62] and *control function* by Ward and Mellor [152].
- *Io_split_join*, is a support mechanism (function) for merging or splitting functional connections, see Section 8.6. The *io_split_join* entity is included in the information model to allow for correct recreation of the flow junction concept in the Statemate tool [61]. An *io_split_join* is always a terminal node in functional hierarchy structure. The *io_split_join* is not modelled as a *general_function_definition* type entity as that would have forced representation of configuration management information for each usage of the entity.
- *Functional_composition_relationship*, is the mechanism for assigning a *general_functionality_instance* type object as a child in a *composite_function_definition* object. A *general_functionality_instance* object may be referred to as *child_functionality* by at most one *Functional_composition_relationship* object.

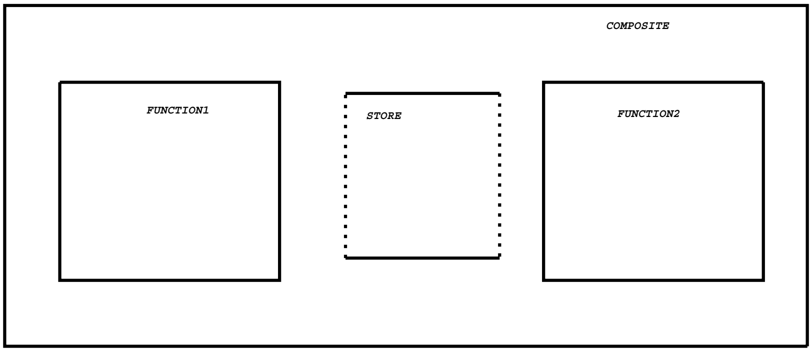


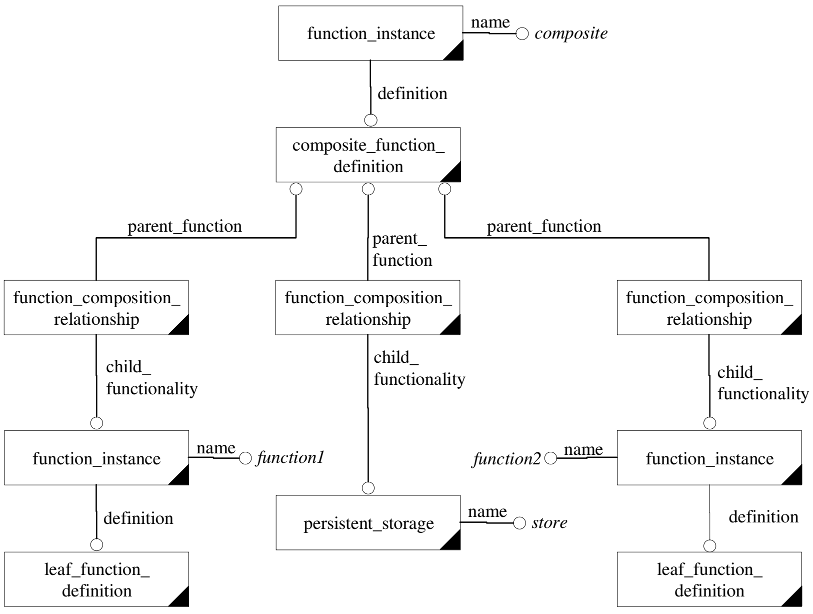
Figure 8.3: Statemate function hierarchy example

8.4.1 FUNCTIONAL HIERARCHY EXAMPLE

Figure 8.3 illustrates a simple functional hierarchy structure in the graphical notation defined by the tool Statemate [61]. In the figure the activity (function) named *composite* is composed of three functional elements: the activities named *function1* and *function2* and the data store named *store*. The corresponding representation for capturing the structure of the specification fragment in the information model is presented in Figure 8.4.

8.4.2 FUNCTIONAL HIERARCHY STRUCTURE

Intuitively a functional hierarchy structure as outlined in Section 8.3 form a tree, i.e., for each node in the functional architecture there is at most one parent function. With the introduction of *function_instance* and *general_function_definition* objects it is possible that a single *general_function_definition* object is providing the definition for multiple *function_instance* objects within the same or across different functional hierarchy structures. Conversely a *general_function_definition* object can be a child in multiple function hierarchy structures. Consequently, a function hierarchy model as captured in the information model is not a tree but a directed acyclic graph.



Note that version management information for *function_definition* objects are suppressed in the figure

Figure 8.4: Information model representation of the model fragment presented in Figure 8.3

In Figure 8.5 there are excerpts of three functional hierarchy structures instantiated. The three environment (root) functions are indicated by labels 1 - 3. The *function_instance* objects at labels 1 and 2 indicate environment functions for two separate functional hierarchy model sharing the same structure. The *function_instance* object at label 3 indicate the environment function for a functional hierarchy model which is sharing a subset of the functional structures from the *function_instance* objects at labels 1 and 2.

The *leaf_function_definition* object at label 4 in Figure 8.5 provide the definition for *function_instance* objects named *leaf2* and *leaf3*.

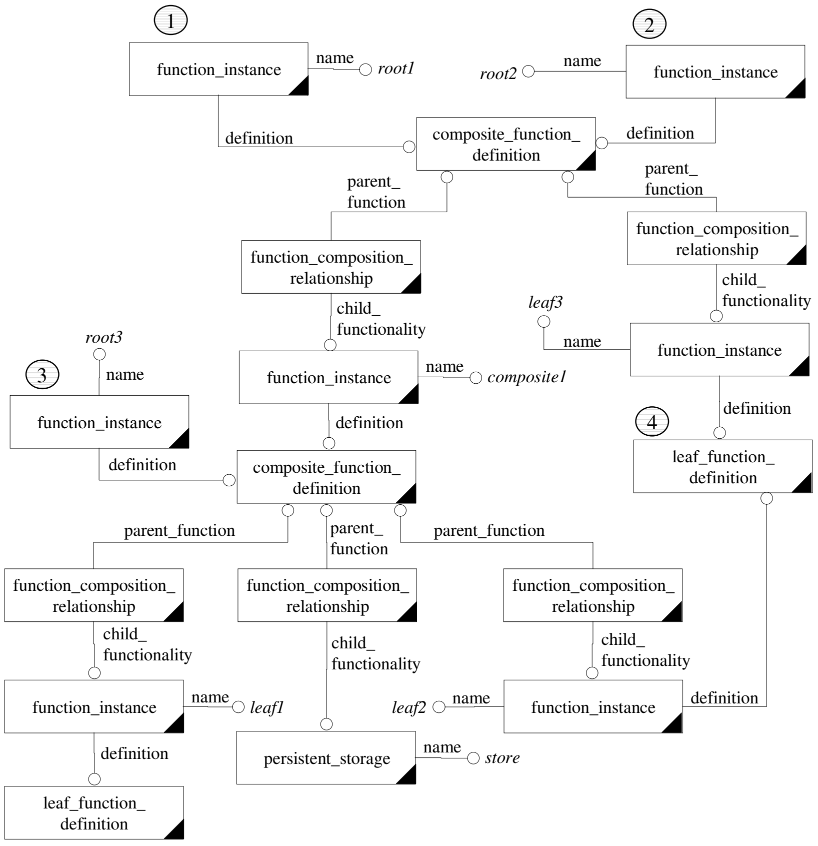


Figure 8.5: Complex function hierarchy examples

8.4.3 FUNCTIONAL HIERARCHY ELEMENTS — DISCUSSION

The choice of entities for capturing the functional hierarchy model has been heavily debated. Oliver [113] argues that there is no place for concepts like *persistent_storage* as it represents abstractions suitable for software specification only. The position held by Oliver is due to a more restricted view on methods relevant for Systems Engineering compared with the one taken for the creation of the information model. The motiva-

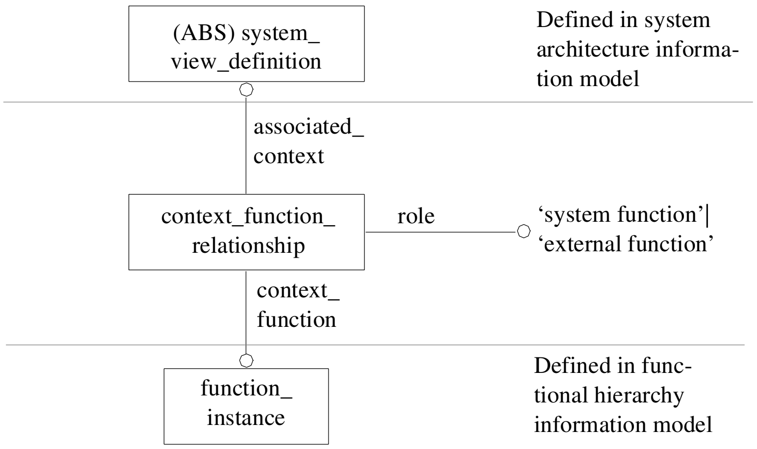

(ABS) system_ view_definition
Defined in system architecture information model
associated_ context
context_function_ relationship
role
'system function'| 'external function'
context_ function
function_ instance
Defined in functional hierarchy information model

Figure 8.6: Functional context information model

tion for including the persistent storage entity and thus include support for methods such as Statemate [61] and Hatley and Pirbhai [62] is that there are clear indications that these methods are used by systems engineers despite the fact they were not initially conceived as Systems Engineering methods. Support for these method in the information model is included in line with the information modelling requirements identified in Section 4.3.

8.5 Functional Context Information Model

This section describes the portion of the information model for representing the root function of a function hierarchy. The root function is represented by the *system_view* entity presented in Chapter 6 of the thesis.

The information model portion presented in Section 8.4 provides the capability to create functional hierarchy structure of arbitrary depth. When combined with the entities presented in this section a functional hierarchy model may be associated with a specific system life-cycle view or viewpoint. A function may either be assigned to a system as an external function (part of the environment) or a system function (part of the system

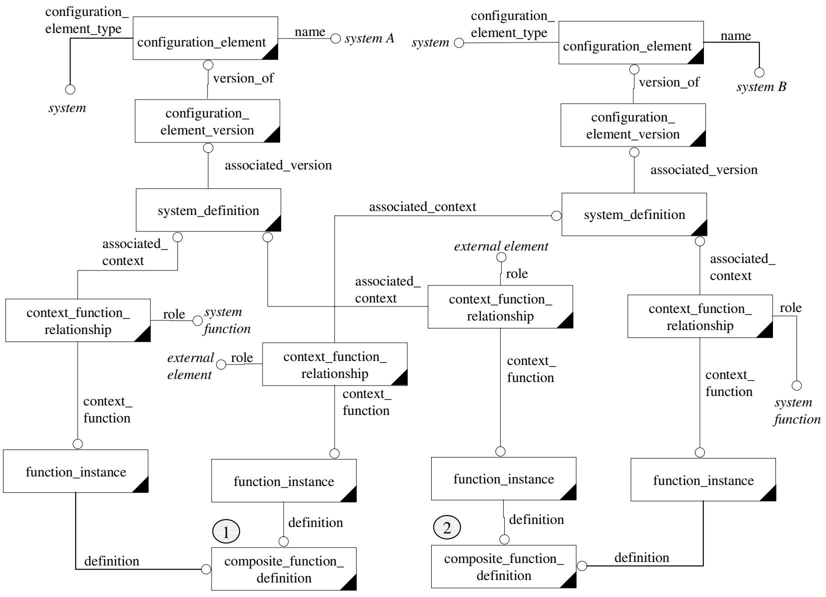


Figure 8.7: Functional context example instantiations

under specification). The information model for capture functional context data is presented in Figure 8.6.

- The entity *context_function_relationship* assign a *function_instance* object to a *system_view_definition* object either as an 'external function' or as a 'system function'. The *function_instance* object related may not be a child function in any composition. For any *system_view_definition* object there may be at most one 'system function' and any number of 'external element' objects assigned.

The construct allows for representation of external functions as composite structures, i.e., the auxiliary functionality external functions identified in Section 8.3.1 are represented in the same way as normal functions. This is less restrictive than what is allowed in Statemate [61], Hatley and Pirbhai [62] and Ward and Mellor [152] where external activities and terminators are leaves in the functional hierarchy structure.

The selected structure allow for representation of multiple perspectives on a function. A *function_instance* object may be the 'system function' for a particular system and an 'external function' for a number of other systems. Moreover, it allow for the association of the same instance of a functional hierarchy model to multiple versions of a system specification.

8.5.1 FUNCTIONAL CONTEXT EXAMPLE

An example instantiation of the functional context model is presented in Figure 8.7. In the figure the *composite_function_definition* object at label 1 represent the system function for the system named *system A* and the *composite_function_definition* object at label 2 is representing functionality external to the system. The situation is reversed for the system named system B in the figure.

8.6 Functional Interaction

The functional interaction aspect considers the semantics for the interaction between functional elements (W) in a functional architecture model. Functional interaction allows for representation of items (e.g., data, information, material or energy) communicated or exchanged between functional elements. The multiplicity of definitions for functional interaction semantics is illustrated by the functional interaction model proposed by Richter and Maffeo [128], Harel and Politi [61] and by Oliver et al. [114]. The three aspects of functional interaction illustrated in Figure 8.8 are considered in this thesis:

- The *function connection* aspect concerns the mechanisms for interconnection of individual functional elements indicating valid paths for communication and exchange of items.
- The *item representation* aspect concerns the representation of the items conveyed by the interaction are within the tool, i.e., by the use of data types.
- The *item temporal characteristic* aspect concerns the rules for item production and consumption.

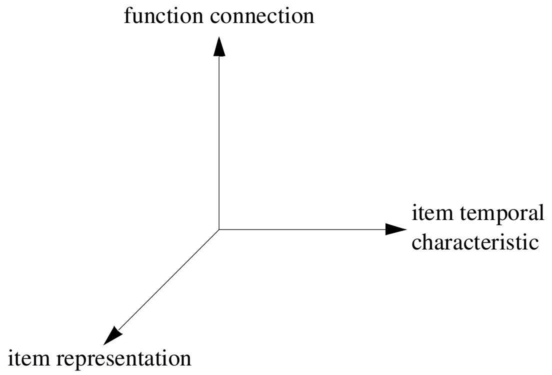


Figure 8.8: Three aspects of functional interaction

Each aspect is outlined in more detail below.

8.6.1 FUNCTION CONNECTIONS

A connection is a directed relation between a functional element producing an item, the source, and the functional element consuming the item, the destination. Multiple connections conveying the same item is used for cases where the functional interaction is bi-directional or there are multiple sources or destinations of the interaction.

8.6.2 TEMPORAL CHARACTERISTICS

Functional interaction is either *causal* or *non-causal*. Causal functional interaction from function *A* to function *B* imposes a partial activation order such that the activation of function *A* is a pre-condition for activation of function *B*. The item sent from function A to function B is a prerequisite for activating function B. Causal interaction is assumed in some formalisations of data flow diagrams, e.g., by Tau and Kung [147], and by Lee and Tang [88]. No specific activation order is defined under non-causal functional interaction.

8.6.3 ITEM REPRESENTATION

Functional interaction can also be classified according to the information content carried. A *non-value bearing* connection has no value content. In other words, each instance of a connection always conveys the same item. *Non-value bearing* items either represent *general-purpose signals* whose purpose is defined by the user or a *prompt* (command) that has a standard interpretation, e.g., 'start function'.

A *value bearing* connection carries an item that have a value content that represents, e.g., information, data, material or energy. The representation of *Value bearing* items may be further specialised in accordance with what is done in ordinary imperative programming languages, i.e., support for representing structured data types.

8.6.4 METHOD OVERVIEW

Several abstractions models are used to capture functional interaction information as illustrated below.

- EFFBD define two classes for capturing the temporal aspects of functional interaction: triggering (causal) or non-triggering (non-causal). Some data value is always assumed to be present for non-causal interaction. There is no support for specifying how the items exchanged are represented.
- The Hatley and Pirbhai method [62] [22] define two temporal characteristics for functional interaction: time continuous (non causal) and time transient (causal). There is limited support for representation of data types.
- Statemate [61] support a wide range of data types. Interaction is per default non-causal, i.e., the value of an item can always be read.
- In IDEF0 [5] there is no concept of data types and all interactions are defined to be causal.

The lack of standard definitions for functional interaction across methods means that loss of information is in many cases inevitable when a specification is moved from one tool to another. For instance, a specification captured with a rich set of data types cannot be maintained in an tool environment which does not support data types. There is also an impact on the information model in the sense that method specific semantics must be supported explicitly or modifications to a functional architecture model will not be detected when data is mapped to or from information model structures.

8.6.5 SPECIFICATION AND REAL WORLD ITEM REPRESENTATION

The selection of specific representations of for an item within the information model a may indicate a software oriented bias, i.e., assumptions can be made that there exist a direct correspondence between the representation of an item in a functional model and how the item is represented in the real world. Such assumptions are not implicit in the information model.

The item representation (type) selected for an item shall be viewed as an abstraction made in the context of a functional model. The nature of the abstraction depends on the purpose of the model and the capabilities of the tool used to capture the functional architecture of the system. Consequently specific representations are not being labelled as being suitable or unsuitable for Systems Engineering purposes.

8.7 Functional Interface and Abstraction

The basic functional hierarchy model from Section 8.3 is extended to represent functional interaction with elements for capturing connections between functions.

$$FHI = (F, \alpha, C, r)$$

Where F, α and r are defined as in Section 8.3 and

- C is the set of connections

Each connection carries an item specified with a specific item representation. As before the symbol *W* is used to denote the set of functional elements.

A connection $c, c \in C$ has one source and one destination functionality identified by the functions

source: $C \rightarrow W$
destination: $C \rightarrow W$

The item conveyed by the connection is communicated from the source object to the destination object. The set of input connections to a functionality $w, w \in W$ is denoted $i(w)$ and the set of output connections $o(w)$.

The rules for routing individual connections and the definition of the inputs and outputs of a function depend on method. Three relevant alternative semantics for functional connections has been identified:

- Explicit functional interfaces
- Implicit functional interfaces with data abstraction
- Implicit functional interface without data abstraction

The properties of each alternative are discussed below.

8.7.1 EXPLICIT FUNCTIONAL INTERFACES

Under this semantic the interfaces to composite and leaf functions are explicit. This means that functional interaction between functional elements not sharing the same parent will have to be routed through the interface of their parent functions until the least common ancestor function for the source and destination functional elements is reached. The following constraints apply for connection routing in methods supporting explicit interfaces:

Let $c \in C$, $f_1, f_2 \in W$, and $f_1 = \mathit{source}\ (c)$ and $f_2 = \mathit{destination}\ (c)$, where:
$f_1 = \text{parent}\ (f_2) \vee f_2 = \text{parent}\ (f_1) \vee f_1 \in \text{children}(\text{parent}(f_2))$

Multiple connection objects are required to capture functional interaction where the functional elements do not share a common parent function, as illustrated in Figure 8.9. Consequently there shall be a mechanism for relating inputs to outputs at different level of decomposition in a functional model. In many methods this is solved through the use of globally

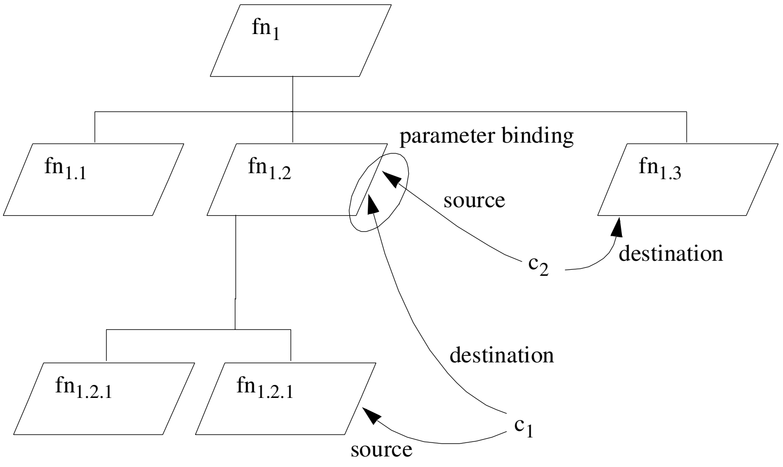


Figure 8.9: Functional interaction under explicit functional interfaces from $\mathrm{fn}_{1.2.1}$ to $\mathrm{fn}_{1.3}$

unique identifiers, e.g., via the connection name attribute that is common for all connections routing an item. An alternate approach is to identify related connection objects via explicit connection binding objects.

Data abstraction is enforced under this semantic — the input and output connections to a functionality *w* is defined by:

$$i(w) = \{\forall c, c \in C, destination(c) = w\}$$

$$o(w) = \{\forall c, c \in C, source(c) = w\}$$

Explicit functional interfaces for connection routing are, e.g., enforced by the structured analysis methods defined by Hatley and Pirbhai [62], Ward and Mellor [152], IDEF0 [5], Lotos [148] and by the research system POOSL by van der Putten and Voeten [150].

8.7.2 IMPLICIT INTERFACES WITH DATA ABSTRACTION

Under this semantics connections between functions at different levels in the function hierarchy are allowed without passing through the interface of any parent function.

Let $c, c \in C$ and $source\ (c) = f_1$ and $destination\ (c) = f_2$, $f_1, f_2 \in W$, then f_1 and f_2 shall satisfy the following constraint:
$parent^+(f_1) \cap parent^+(f_2) \neq \varnothing$.

I.e., f_1 and f_2 shall have at least one common ancestor for the connection to be valid. Figure 8.10 illustrates functional interaction functional under implicit functional interfaces. An item produced in function $fn_{1.2.1}$ is consumed in $fn_{1.3}$ without passing through the interface of $fn_{1.2}$. Had $fn_{1.2.1}$ or $fn_{1.3}$ had any child functions then they would not have been able to access the item conveyed by the connection.

The set of input and output connections for a functionality *w* is the same as defined for the explicit functional interface class, i.e., defined by

$$i(w) = \{\forall c, c \in C, destination(c) = w\}$$

$$o(w) = \{\forall c, c \in C, source(c) = w\}$$

Implicit interfaces for connection routing is enforced by the EFFBD method [95].

8.7.3 IMPLICIT INTERFACES WITHOUT DATA ABSTRACTION

Under this semantics connections between functions at different levels of decomposition are allowed without passing through the interface of any parent functions. Data abstraction is not enforced. A connection with its source or destination in a composite function *f* is accessible for reading and writing for all child functions of *f*.

For this class of functional interaction semantics the same routing rule applies for the Implicit interfaces with data abstraction class presented above.

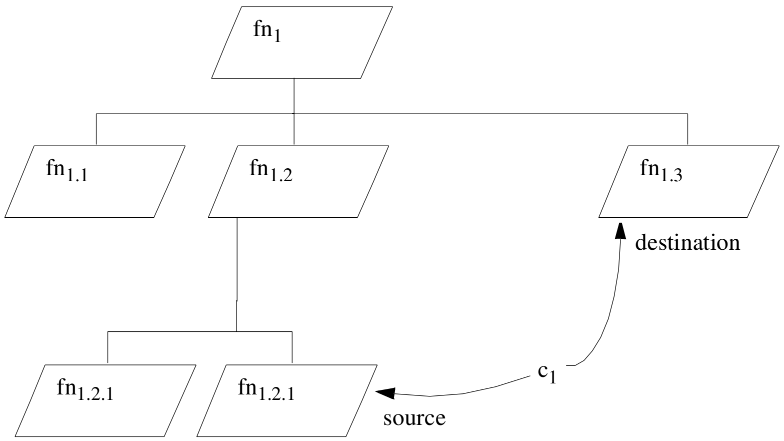


Figure 8.10: Functional interaction under implicit functional interfaces from $\text{fn}_{1.2.1}$ to $\text{fn}_{1.3}$

For a functional element *w* the set of inputs and outputs is calculated using the following formulas:

$$i(w) = \{\forall c, c \in C, destination(c) = parent^*(w)\}$$

$$o(w) = \{\forall c, c \in C, source(c) = parent^*(w)\}$$

Figure 8.11 illustrate functional interaction under implicit functional interfaces without data abstraction. Implicit interfaces are, for instance, supported by Statemate activity charts [61].

8.7.4 FUNCTIONAL INTERFACES IN THE INFORMATION MODEL

Maintaining the three models for functional interface in the information model representation is not ideal. It would require any tool interface based on the information model to include support for three different algorithms

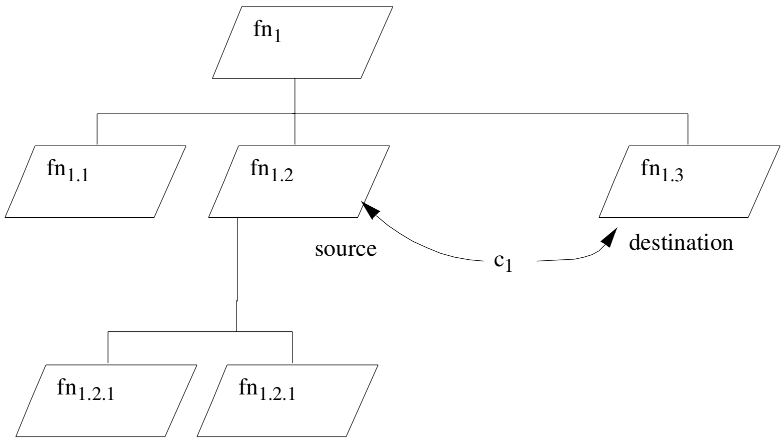


Figure 8.11: Functional interaction under implicit functional interfaces without data abstraction from $\mathrm{fn}_{1.2}$ or any of its children to $\mathrm{fn}_{1.3}$

for interpreting functional interaction. Instead the explicit functional interfaces semantics as presented in Section 8.7.1 are used exclusively in the information model.

The motivation for selecting the explicit functional interface representation as the single representation in the information model is that:

- The interface to each functional element is explicitly defined which is important in cases where a function can be used in multiple contexts. The only way to define interaction with a functional element is through its interface.
- Splitting and joining connections is performed explicitly at the interface of functions.

The selection of a single semantics for functional interaction within the information model lead to the introduction of complex mapping functions for methods supporting implicit interfaces as defined below.

Implicit interfaces with data abstraction

When a model containing a connection, c, with source(c) = f_1 and destination (c) = f_2 is exported onto the data model representation then connection fragments must be instantiated between all functions f up to the least common ancestor function of f_1 and f_2.

Conversely, when data is imported from the information model representation then the ultimate source and destination of an interaction must be established by a top down traversal of the set of connections. All connection objects traversed, but the ultimate source and destination function, are discarded in the tool internal representation.

Implicit interfaces without data abstraction

The same rules as for the *implicit interfaces with data abstraction* semantics apply. Depending on tool capability it may be possible to uniquely identify the set of functions producing or consuming a specific item when data is exported. In such cases connection objects may be created to identify those functions explicitly. However, this transformation cannot be reversed when data is imported from the information model representation.

8.8 Functional Interaction Information Model

The entities for the functional interaction part of the information model are presented in Figure 8.12. The entities introduced are documented below.

- *io_port*, is the abstract supertype for all entities that represent elements of the interface of a functional element. An *io_port* is one of the following *actual_io_port*, *formal_io_port*, *control_io_port* or *io_composition_port*. Interaction via an *io_port* element is uni-directional, i.e., an *io_port* is either an input or an output of a functional element. The representation of the item conveyed via the *io_port* is captured by the *data_instance* object identified by the *data* attribute. An *io_port* is said to be a producing port if can produce information to a *functional_link* object and a consuming port if it can consume infor-

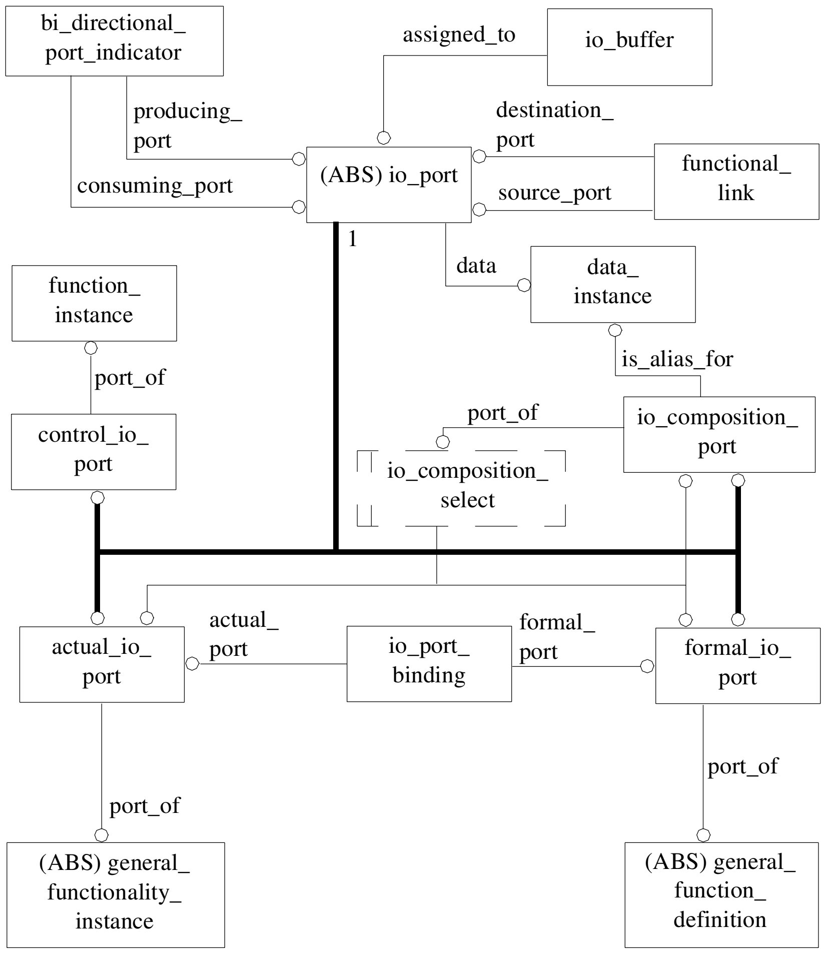


Figure 8.12: Functional interaction information model

mation from a *functional_link* object. A port is either a consuming port or a producing port.

- *functional_link*, represent a connection between a pair of *io_port* objects. The connection is directed from the *io_port* object indicated by the *source_port* attribute to the *io_port* object indicated by the *destination_port* attribute. The connection is valid iff the same *data_instance* object is referenced by the pair *io_port* objects references by the *functional_link* object. The connection established by a *functional_link* object is ideal. There are no delays or energy losses. By default a *functional_link* object realises non-causal interaction. The semantic can be modified by the use of *io_buffer* objects as outlined in Section 8.8.4.

- *actual_io_port*, represent an actual parameter to a functional element. As such the *actual_io_port* is associated to a *general_functionality_instance* object.
- *formal_io_port*, represent a formal parameter to a functional element and thus associated to a *general_function_definition* type object.
- *io_port_binding*, associates a *actual_io_port* object of a *function_instance* object with a compatible *formal_io_port* object of a *general_function_definition* object. At most one *io_port_binding* object may be applied to any *actual_io_port* object.
- *control_io_port*, is an *io_port* with a defined threshold for data driven function activation for the *function_instance* object the *control_io_port* object is assigned to. A *control_io_port* may define the condition for function activation only, or for function activation and deactivation.
- *io_composition_port*, is the mechanism for accessing a specific data element of a composite data type for reading or updating.
- *bi_directional_port_indicator*, relate a pair of *io_port* objects to indicate that they support bi-directional interaction for an item.
- *io_buffer* is a modifier to associate a first in first out buffer of unlimited size for items with an *io_port* object. The *io_buffer* modifies the semantics of the connection such items will be consumed when read, i.e., a connection becomes causal by the addition of an *io_buffer* object to the producing or consuming *io_port* object.

8.8.1 IO_PORT AND FUNCTION_LINK

In the information model the *io_port* entity is a supertype of *formal_io_ port* and *actual_io_port*. *io_port* support unidirectional interaction only — a port is either an input indicating that items are entering the assigned functional element or an output indicating that items are exiting the functional element. There are no restrictions on how many *functional_link* objects that may have their source or destination in a producing or consuming *io_port* object.

An *actual_io_port* has the role as producer of items if it is an output port and the role as consumer of items if it is an input port. Conversely a *formal_io_port* has the role as consumer of items if it is an output port and has the role as producer of items if it is an input port. The classification for formal ports may appear counterintuitive but is due to the fact that an output *formal_io_port* object consumes items for distribution outside the assigned *general_function_definition* object and an input *formal_io_port* produces items for distribution to the children functional elements of a *general_function_definition* type object.

Rules for correct connection of *function_link* objects are formed based on the definition of producer and consumer *io_ports*. A *functional_link* object realises a unidirectional connection from a producing *io_port* object to a consuming *io_port* object. In this sense a *functional_link* can be compared with an ideal diode. The constraints for connection routing outlined in Section 8.7.1 apply for *functional_link* objects.

8.8.2 FUNCTIONAL INTERACTION EXAMPLE

An instantiated information model fragment illustrating the connection routing principles in the information is presented in Figure 8.13. The instantiation include a *composite_function_definition* object with a single child *function_instance* object. There is one input and one output to the *composite_function_definition* object, labels 1 and 4 in the figure. Likewise there are two inputs to the child *function_instance* object in the figure. These are represented by the *actual_io_port* objects at label 2 and 3 in

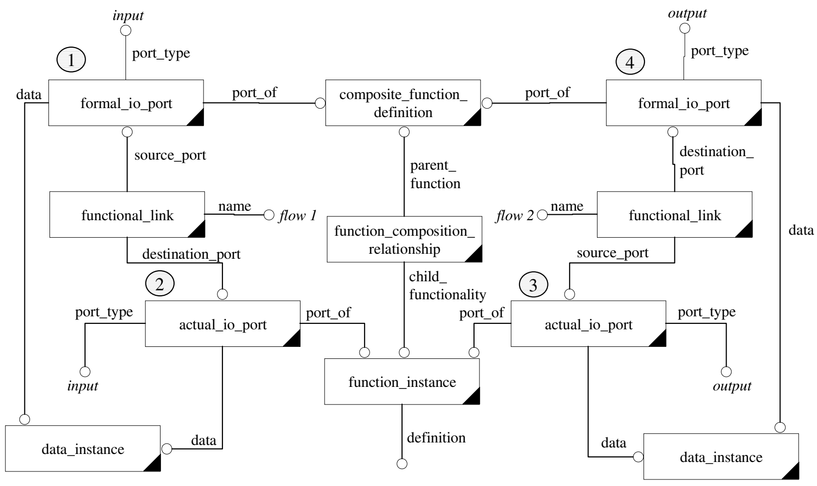


Figure 8.13: Functional interaction example

the figure. Two *functional_link* objects are used to establish the connection between the *io_port* objects. Both *functional_link* objects are valid as they each form a connection from a producing port to a consuming port.

8.8.3 PARAMETER BINDING

Each functional element in the information has an explicit interface defined by the assigned *io_port* objects. For a *function_instance* object it is in the form of *actual_io_port* objects and *formal_io_port* objects for a *general_function_definition* type object. The entity *io_port_binding* provide the mechanism for the association of an *actual_io_port* object of a *functional_instance* to a *formal_io_port* object of a *general_function_definition*.

The following prerequisites shall be fulfilled for a valid parameter binding:

1. The *role* attribute of the two port objects bound shall not have the same value, i.e., one of the ports shall be a producer of items and other

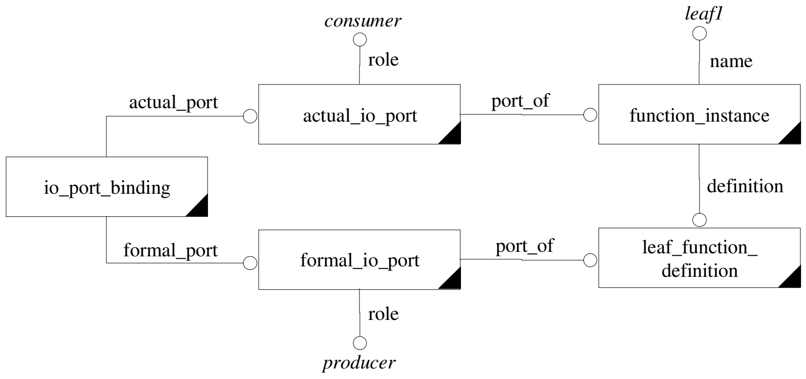


Figure 8.14: Parameter binding example

shall be a consumer.

2. The *function_instance* and *general_function_definition* type objects the *io_port* object involved in the parameter binding are connected to shall be related such that the *general_function_definition* object is identified by the *definition* attribute of the *function_instance* object.
3. The items carried by the bound *io_port* objects shall be of the same basic data type.

Figure 8.14 illustrates a valid parameter binding example.

8.8.4 CAUSAL INTERACTION AND FUNCTION ACTIVATION

The default behaviour of any combination of *functional_link* and *io_port* objects is that an item carrying some (may be undefined) value is available in the *consuming io_port*, i.e., the connection is non-causal. A connection may be modified to become causal by the association of an *io_buffer* object to an *io_port* producing or consuming data in a connection.

- If the *io_buffer* object is assigned to an *io_port* object producing items then the items will be buffered at the producing port. If more than one *functional_link* objects is connected to the port then any item produced on the port will be consumed by one of the functions connected to the *function_link*.

- If the *io_buffer* object is assigned to an *io_port* object consuming items then the items will be buffered in the consuming port allowing each consumer a local copy of the items produced.

Any number of items may accumulate in an *io_buffer* object. The semantics of the case where a *functional_link* object connections to two buffered *io_ports* is not defined and explicitly forbidden in the information model.

The information model support for representing activation conditions for a function is further specified in Section 8.9.

8.9 Function Instruction

The function instruction aspect defines how an individual function generates output from its inputs and the termination condition for a function. In [120] the internal behaviour of a function is characterised as follows:

1. Wait for activation and take input from the interface.
2. Internal processing, depending on tool support the processing algorithm may be expressed in a formal or informal language.
3. Output delay δ time units, where δ is a real number in the interval $0 \leq \delta \leq \infty$. δ is a temporal property of the function that is dependent on the system the function is associated with.
4. Produce output to the environment.

In the sequence above steps 1 and 4 are performed in conformance with the functional interaction semantics defined for each input and output to a function. Presence of input items, for all causal connections to a function is a prerequisite for function activation. Additional activation condition may be enforced by the introduction of clocks activating a function at regular intervals or activation may be controlled by an explicit behaviour component as outlined in Section 8.11. The association of a clock to a function makes the function time-discrete. If no clocks are assigned a function is assumed to be time-continuous.

The function instruction aspect also encompasses the function termination semantics. Here three alternatives can be distinguished.

1. A function is continuously active.
2. The activation of a function is controlled from an external source. Termination depends solely on the internal state of the function.
3. The activation and termination of a function is controlled from an external source.

External activation or deactivation conditions may be defined by the reception of a prompt. Alternatively function activation could be coupled to the item value of a value bearing connection as defined for the *control_io_port* entity in Section 8.8.

8.9.1 SYSTEM SPECIFIC FUNCTION PROPERTIES

The information model needs to handle the case where a single functional architecture model is used in multiple contexts. This is analogous to the system specific requirement information model presented in Section 7.6.

For instance, a supplier may provide a range of systems that provide the same functionality, but with different temporal performance. In this aspect it is important that functional properties such as output delay can be captured in the context of each individual system. Likewise the specified properties, but not the overall functionality of a system may change in the development process. This is in line for the design guideline for reuse of model fragments captured in Section 4.3.

8.10 Function Instruction Information Model

The function instruction information model contains entities for associating system variant information, e.g., temporal properties and requirement allocations, to individual functions and entities for capture of function specific properties. System variant information can be represented for each system viewpoint a functional architecture model is associated with. The entities in the model are presented in Figure 8.15 and documented below. The presentation is split into two parts. One for the realisation of the system variant structure of the functional architecture model and one for associating functional properties to model elements. Note that the entity for capture function algorithm (*leaf_function_definition*) is presented in Section 8.4.

8.10.1 SYSTEM VARIANT INFORMATION MODEL

The system variant functional model is based on the identification of distinct configurations of objects for capturing functional properties for a functional hierarchy model assigned to a specific system life-cycle view or system viewpoint. System variant properties of a functional architecture model are captured by the use of a placeholder entity — *functionality_instance_reference* — that is associated with a *general_functionality_instance* object. The entity *functionality_reference_configuration* defines the set of *general_functionality_instance* objects applicable for a functional architecture model when associated to a specific system view. Association of a *functionality_instance_reference* object to a *functionality_reference_configuration* is made via the entity *functionality_reference_configuration_relationship* that relates a parent and a child *functionality_instance_reference* object to a reference configuration.

The definition of the entities in the model is further presented below.

- *functionality_instance_reference*, is the placeholder for allocation of system variant information to individual elements of the functional architecture. Multiple *functionality_instance_reference* objects may be assigned to a single *general_functionality_instance* object. However, only one of them may be valid for any given system view.
- *functionality_reference_composition_relationship* defines a parent-child relationship between a pair of *functionality_instance_reference* objects. The parent - child relationship captured must match that formed for the functional hierarchy structure. The entity also identifies a *functionality_reference_configuration* for which the parent-child relationship is valid.
- *functionality_reference_configuration* acts as a collector of placeholder objects valid for a specific system view.

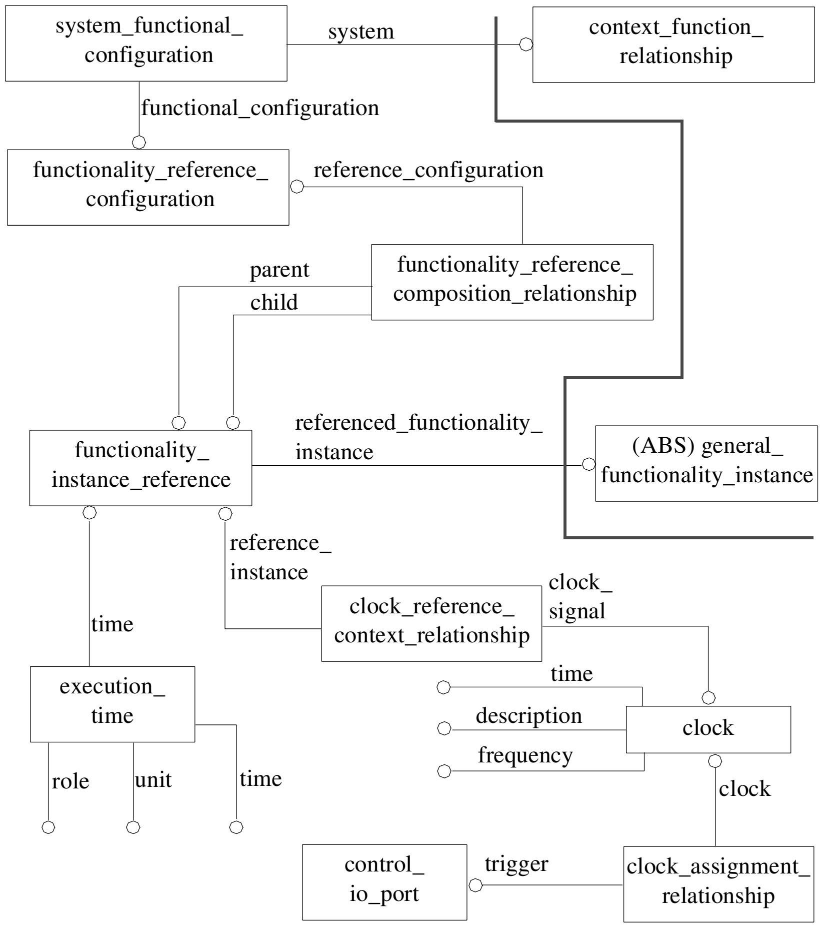


Figure 8.15: Function instruction information model

- *system_functional_configuration* is the association of a reference configuration to a specific system view. Multiple *system_functional_configuration* objects may be used to assign a reference configuration to multiple system life-cycle views or viewpoints sharing the same functional architecture.

8.10.2 FUNCTIONAL PROPERTIES INFORMATION MODEL

Two kinds of specific system variant functional properties are supported in addition to the general property assignment part of the information model outlined in Section 5.7:

- Clocks for periodic activation of functions.
- Capture of estimated function execution time.

The information model entities are presented in Figure 8.15.

Allocation of clocks for the periodic activation of a function within a functional architecture. The allocation of a clock to a function changes the function temporal characteristics from time-continuous to time-discrete.

- The entity *clock* represents a device that emits pulses at periodic intervals. A *clock* may be associated with one or more *control_io_port* objects by the use of *clock_assignment_relationship* objects. The *clock* association is also made to one or more *functionality_reference_instance* objects via *clock_reference_context_relationship* objects to indicate the configurations a specific clock assignment is valid for.
- The *execution_time* entity support capture of function execution time for a function in the functional architecture. Execution time may be captured for three cases, *normal*, *worst* and *best* case execution time.

8.11 Functional Behaviour

The behaviour aspect of functional architecture concerns the control flow that defines the valid sequence of function activation and termination. In many methods the behaviour model is implicitly defined by the causality relationships defined by functional connections while other include an explicit behaviour component.

Finite state machines in different forms have been proposed and used for capturing the behaviour aspect of the functional architecture, e.g., [61] [62] as have languages where the causality between functions is captured by explicit relationships as in Functional Flow Block Diagrams (FFBD) and Extended Functional Flow Block Diagrams (EFFBD) [4] [95] [96]. In the first case the output language of a finite state machine is used to control the status of individual functions. The control language may be formal or informal. Methods and languages using finite state machines for capturing the behaviour of a functional model have been proposed and studied in detail in, e.g., [58] [59] [60] [61] [120]. No detailed presentation of state based specification languages is made within this thesis.

In contrast with the large number of publications related to the use of finite state machines for capturing functional behaviour relatively little is published on methods using the second approach to capturing functional behaviour.

The introduction of a dedicated representation for capturing functional behaviour changes the assumption on function activation. In a model without a dedicated behaviour component a function is assumed to be active as soon as data is available to its inputs. When a functional behaviour representation is introduced an explicit activation via this representation is required to activate a function. This difference in model semantics must be maintained in the information model.

8.11.1 Causal chain based functional behaviour

Causality based functional behaviour methods are characterised by the arrangement of identified functions to threads. For each function in a thread there exist a finite set of functions that will have to be activated and terminated immediately before the activation of the function. Similarly there exist a finite set of functions that will be activated upon the termination of the function. Consequently ordering of functions in time is supported. Control structures, such as selection (if) statements and loops, are often introduced in causality based functional behaviour methods to increase model clarity. Methods and tools that support causality based functional behaviour includes

- Functional flow block diagrams (FFBD), as supported by the tools ViTech CORE, RDD-100 and documented in, e.g., [24] [95] [96] [112].

- Extended functional flow block diagrams (EFFDB), as supported by the tool ViTech CORE.
- Behaviour diagrams as supported by the tool RDD-100.
- The (BAE SYSTEMS) method CORE and supporting tools documented in [38].

The first three methods in the list above are related as they support the same set of control structures. Behaviour diagrams and EFFDB include notations for functional interaction, while FFBD does not. The difference between behaviour diagrams and EFFBD lies mainly in the fact that threads are arranged vertically in the first and horizontally in the second tool.

In contrast the BAE SYSTEMS method CORE is somewhat different in its support of control structures. An overview of the control structures supported by BAE SYSTEMS CORE and the FFBD family of methods are presented in Table 8.1 below.

Table 8.1: Control structure support per method

id	**Control structure**	**BAE-Systems CORE**	**FFBD**
1	*Thread selection based on function result (value)*	√	√
2	*Thread selection based on probability*		√
3	*Iteration based on constant*	√	√
4	*Iteration based on value evaluation*	√	√
5	*Iteration completion (loop exit conditions)*		√
6	*Concurrent execution*	√	√
7	*Forced termination of concurrent threads*		√

Table 8.1: Control structure support per method

id	Control structure	BAE-Systems CORE	FFBD
8	*Replication (multiple independent activation of a function)*		√
9	*Termination node*		√

The control structures in Table 8.1 is described in more detail below:

1. Thread selection based on function result. One thread out of *n, n>2* is selected based on a predefined fixed probability function. In FFBD terminology this structure is called “multi exit function”.
2. Thread selection based on probability. One thread out of *n, n>2* is selected based on a predefined fixed probability function.
3. Iteration based on constant. All functions enclosed by the iteration construct will be iterated a fixed number of times.
4. Iteration based on value evaluation. All functions enclosed by the iteration construct will be iterated until a fixed iteration condition evaluates to false.
5. Iteration completion. A special control instruction that immediately exits an iteration structure.
6. Concurrent execution. Defines n threads, where $n > 2$, which will execute in parallel and independent of each other. By default, all constituent threads in the control structure must terminate prior to the completion of the structure. Explicit synchronisation points precede and succeed concurrent execution structures.
7. Forced termination of a concurrent thread. A modifier that can be applied to any thread in a Concurrent execution structure. When the last function in a forced termination thread is executed all other threads in the concurrent execution are forced to terminate regardless of the state of the thread.
8. The replication construct defines that n instances of the enclosed functions will be executed. A dedicated control thread is provided for initialisation and termination of the replicated functions.

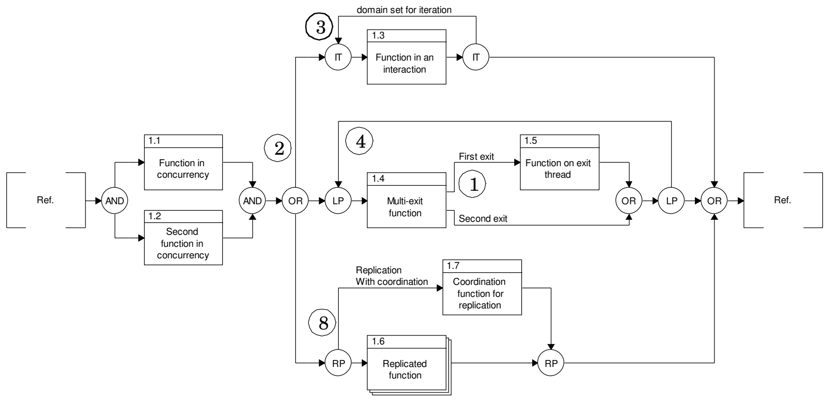


Figure 8.16: Illustration of FFBD control constructs

9. A termination node forces the instant termination of a function. Execution will continue in accordance with the control structure definition for the parent function of the function enclosing the termination node. In FFBD notation a termination node may be used in conjunction with thread selection based on function result to select a specific alternative based on the result of function.

A FFBD diagram illustrating the constructs in Table 8.1 is presented in Figure 8.16. The labels in the figure refer to the id column in Table 8.1.

8.11.2 CAUSAL CHAIN REPRESENTATION

There are multiple methods using causal chains to define functional behaviour. Each method use slightly different control structures. In order to provide a method independent representation the approach illustrated in Figure 8.17 and presented below is selected.

- The definition of a basic model for capturing causality, including selection, concurrency and replication. The similarity between Petri-nets [121] [106] and causal chain methods is exploited for the basic model.

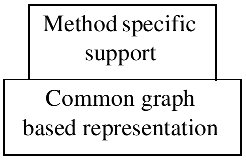


Figure 8.17: Layers for representing functional behaviour

- Definition of model elements to capture method specific constructs. The method specific model elements extend on the basic model to capture method specific constructs. Using this approach data exchange of common causal chain information is possible between tools (indirectly methods) supporting different causal chain methods.

8.11.3 PETRI-NET DEFINITION

Formally a Petri-net [108] is a sextuple $pn = (P, T, i, o, W, M)$ where

- P, commonly called places, is the set of potential conditions in a system. A place is drawn as a circle in graphical Petri-net notation.
- T, commonly called transitions, is the set of potential events in a system. A transition is drawn as a bar drawn at any inclination in graphical Petri-net notation.
- i, commonly called input arcs relates an condition (place) to an event (transition) such that the condition is a pre-condition to the event.
- o, commonly called output arcs relates an event (transition) to the post condition of that event.
- W, the weight of individual arcs. Arc weight is an alternative to having multiple input or output arcs between a place and a transition. The weight is represented by a natural number.
- M, is the time-variant marking function which associates a set of *tokens* with each place in the net.

The set of tokens allocated to places collectively capture the state in the Petri-net. Each change to the net state of the Petri-net is brought about by the *firing* of a transition. In order for a transition to fire it must be enabled.

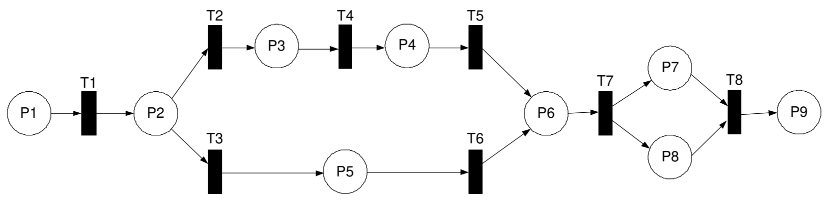


Figure 8.18: Example petri-net

A transition is enabled *iff* for there are at least W((p,t)) tokens in each place p connected via an input arc to the transition, where W((p,t)) is the weight of each input arc.

Firing of a transition alters the value of the marking function and obeys the following rules

1. Only enabled transitions may fire.
2. As a transition *t* fires $W((p_i, t))$ tokens are removed from each input place p_i of the transition *t*, and $W((p_o, t))$ are deposited at each output place p_o of the transition *t*. The value of the marking function elsewhere in the net is not affected by the transition.
3. Firing of a transition is normally assumed to be instantaneous.

An example Petri-net is presented in Figure 8.18. The net in the figure consist of 9 places ($P_1..P_9$), 8 transitions ($T_1..T_8$), 8 input arcs and 8 output arcs. The weight of individual arcs is not specified, which using the adopted norm implies the weight is 1, i.e., each arrow represents one input or output arc. No marking function is presented in the figure.

8.11.4 PETRI-NET FOR REPRESENTING CAUSAL CHAINS

In a causal chain all children functions of a function are ordered in threads in the chain. A Petri-net allow for capture the causality among functions by associating an individual function *f* with a place *p* such that the function *f* is activated when a token moved to place *p*. In a causal chain all children functions $f_1..f_n$ of a function f_0 are associated to at least on place *p*, but a place need not be associated with a function.

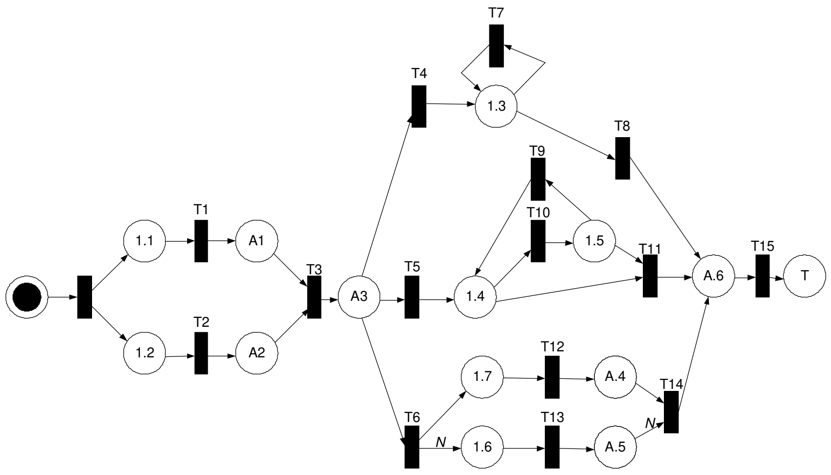


Figure 8.19: Petri-net representation of FFBD example in Figure 8.16

The transition firing rules for a transition is modified such that in addition of items 1..3 in Section 8.11.3 the following conditions applies:

- A transition *t* cannot fire as long as any one function associated with one of its input places p_i is active. The function must terminate prior to transition firing.
- A transition *t* will fire immediately when the number of tokens in each of its input places p_i exceeds the weight of the respective input arc ($W(p_i,t)$)) and no functions associated to any of the input places are active.

In a causal chain model a function will terminate when none of its children functions are active. In the Petri-net representation this corresponds to a marking function where all tokens are in a single place p_i with no input arcs to any transition. In addition, this place p_i shall not be associated with any function.

8.11.5 EXAMPLE USE OF PETRI-NET FOR CAUSAL CHAIN REPRESENTATION

Figure 8.19 captures the Petri-net representation of the FFBD model in Figure 8.16. The labels in inserted in the places in the figure relates to the function associated with the place. The label 1.2 indicates that function 1.2 in Figure 8.16 will be activated when a token is in the place. Labels starting with A are auxiliary places inserted to maintain the correctness of the causal chain. Labels A.1, A2 are inserted to correctly represent the cases where:

- Function 1.1 terminates before function 1.2, in this case the token in place 1.1 will move to place A.1.
- Alternatively, if function 1.2 terminates before function 1.1 the token in place 1.2 will move to place A.2.

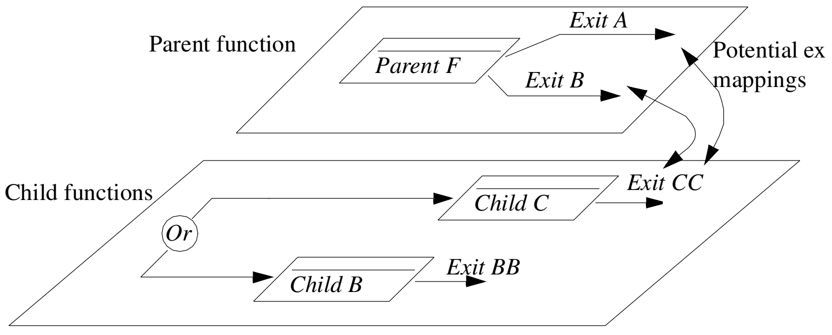
Note that transition T3 in the Figure 8.19 will not fire until there is a token in places A.1 and A.2.

The following elements in Figure 8.19 are also of interest for the Petri-net representation of FFBD structures.

- Place A.3 defines the initial branch in the thread selection based on probability structure (The selection structure terminates in place A.6). One of the transitions T4, T5 or T6 will fire when a token enters A.3.
- Transition T6 is the initial transition in the replication structure in Figure 8.16.
- The output arc from the transition to place 1.6 has weight n, indicating that n tokens with be deposited in place 1.6. Each token will start an instance of function 1.6. The functions will be deactivated one by one for each firing of transition T13.

8.11.6 RESTRICTIONS IN PETRI-NET SUPPORT

Petri-net allows for capture of basic causal chain control constructs such as selection, iterations in the graph, iteration completion, concurrency, replication and termination nodes. There is however no support for the following control constructs presented in Table 8.1:



The information model need to be able to unambiguously map exit condition to exits in the parent function. E.g., from *Exit BB* to either *Exit A* or *Exit B*

Figure 8.20: Mapping for multi exit functions

- Distinguishing between iteration based on condition and on fixed value
- Identification of loop exit conditions
- Forced termination of concurrent threads

Explicit support for these constructs needs to be included in the information model if method specific notations shall be maintained in data exchange. Moreover the information model needs to support mapping between a composite function having multiple exits and the internal exit conditions within the function as outlined in the FFBD in Figure 8.20

8.12 Constraints in Support of Functional Behaviour

Some of the methods identified as being of interest for use with the information model support languages for capturing the behaviour of functions and other components of a functional architecture model. This allow for simulation and in some cases for formal verification of a model captured in a tool.

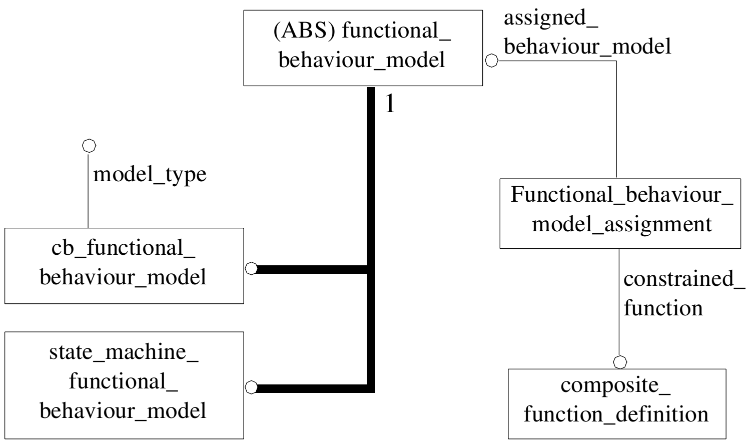

(ABS) functional_ behaviour_model
assigned_ behaviour_model
1
model_type
Functional_behaviour_ model_assignment
cb_functional_ behaviour_model
constrained_ function
state_machine_ functional_ behaviour_model
composite_ function_definition

Figure 8.21: Behaviour model information model elements

The information model does not include any definition of formally defined languages for capturing the algorithms of individual functions. Algorithms are represented as plain text. There are no facilities for facilitating machine analysis of the semantics of individual text elements beyond an attribute for capturing the language used in a specific textual element.

This design decision restricts the benefits of using the information model between tools with support for formal languages. The structure of the models will be carried over in the exchange, but automatic translation of algorithms is not supported. The motivation for making this restriction is primarily the amount of work required to define a formal language and the cost to include support in tool interfaces.

8.13 Behaviour Model Information Model

The introduction of an explicit behaviour component for a function f_0 changes the default behaviour for its children functions f_1 to f_n such that an explicit activation is required to activate each child function. This is discussed in Section 8.11. In the information model this change of activation semantics is captured by the entity *functional_behaviour_model*.

The association of a *functional_behaviour_model* object to a *composite_function_definition* object is made by an object of type *functional_behaviour_model_assignment*. A behaviour model can be assigned to exactly one *composite_function_defintion* object, and a *composite_function_defintion* object may have at most one behaviour model assigned. Two kinds of *functional_behaviour_model* are defined:

- *cb_functional_behaviour_model*: defining that the functional behaviour of the associated function is defined by a causal chain as outlined in Section 8.11.1. The *model_type* attribute allows tools to identify the original causal chain model representation, e.g., FFBD, Behaviour diagram, to facilitate interpretation of the data.
- *state_machine_functional_behaviour_model*: defining that the functional behaviour of the associated function is defined by state machines, either in Moore, Mealy or Statechart notation.

The behaviour model EXPRESS-G information model elements are presented in Figure 8.21.

8.14 State-Based Functional Behaviour

The state-based functional behaviour information model contains entities for representing states, aggregation of states and state transitions. This model is designed to support the representation of Mealy and Moore state machines as well as Statecharts.

8.14.1 STATE MACHINE INFORMATION MODEL

The state machine information model is presented in Figure 8.22. The main entities in the model are:

- *Functional_state_context,* provides an enclosure for the state machines. The attribute *state_machine_type* allow for capture the type of the enclosed state machine: *Mealy*, *Moore*, *Mealy and Moore* (Combination of Mealy and Moore state machine), or *Statechart*. The *original_representation* attribute allow for capture of the presentation format in the original tool, i.e., *graphical* or *tabular*. This attribute is

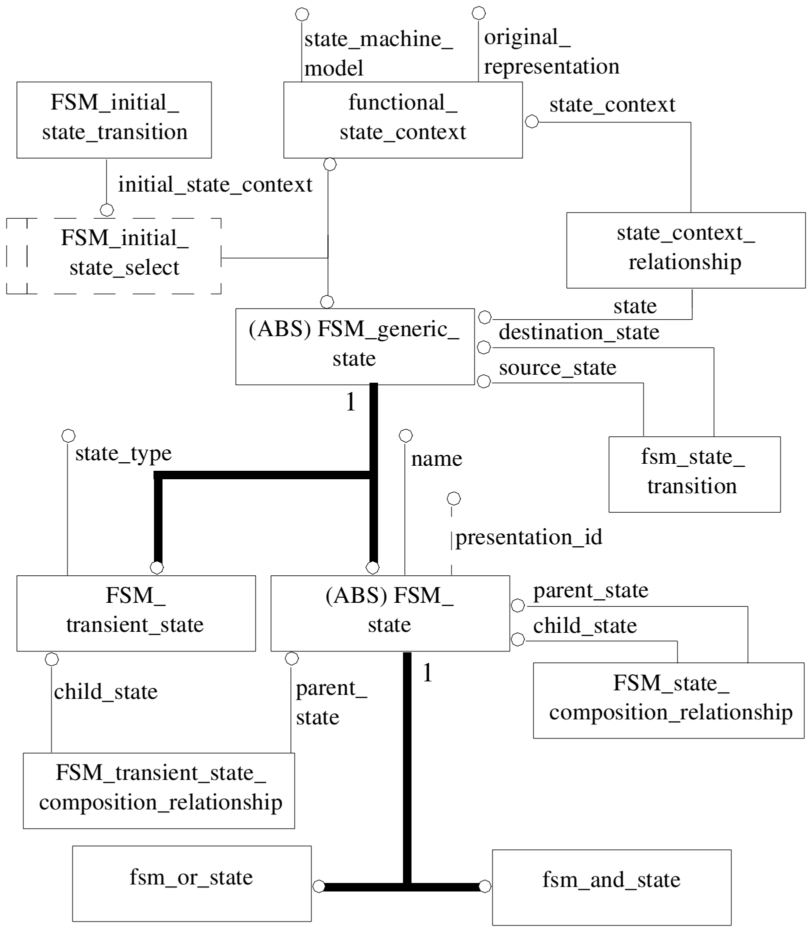


Figure 8.22: State machine information model

essential for recreation of the original presentation format of a state machine after data exchange.

Multiple entities are used to define state types

- An *FSM_generic_state* is an abstract entities representing one of an *FSM_state* or an *FSM_transient_state*. An *FSM_generic_state* type object may be an element of a *functional_state_context*. An *FSM_* *generic_state* type object may be the initial state in a state machine.

- An *FSM_State* is an abstract entity representing one of *FSM_or_state* and *FSM_and_state*. An *FSM_state* can be decompositioned and be the source or destination of any number of *FSM_state_transition* objects.
- An *FSM_or_state* is an *FSM_state,* potentially composed of other *FSM_state* objects, with the discriminator that when entered via a state transition then the state and all of its parent states, but none of its sibling states are entered
- An *FSM_and_state* is a state decomposed into at least two *FSM_state* objects with the discriminators that when entered via a state transition then all of its parent states and all immediate children states are entered.
- An *FSM_transient_state* is an *FSM_generic_state* with the discriminator that performs a predefined activity terminates instantaneously when entered. An *FSM_transient_state* cannot be decomposed. *Transient* states are supported by, e.g., the Statemate implementation of Statecharts [59] and the value of the *state_type* attribute of a *transient_state* is one of: *history, deep history, condition and selection*. The semantic of each value is defined in [60].

Transitions between states may be of two types:

- *FSM_initial_state_transition* identifies the state that will be entered when a state machine is first activated. An *FSM_initial_state_transition* has no source state, and one destination state. At most one *FSM_initial_state_transition* objects may be assigned for each composite state.
- *FSM_state_transition* defines a state-transition between two *FSM_generic_state* objects. An *FSM_state_transition* may be guarded by an arbitrary condition that must be fulfilled prior to the execution of the state-transition.

Both types of state-transitions may have actions associated that executes when the transition fires. The actions associated with a transition are captured as text with an associated attribute indicating the encoding language.

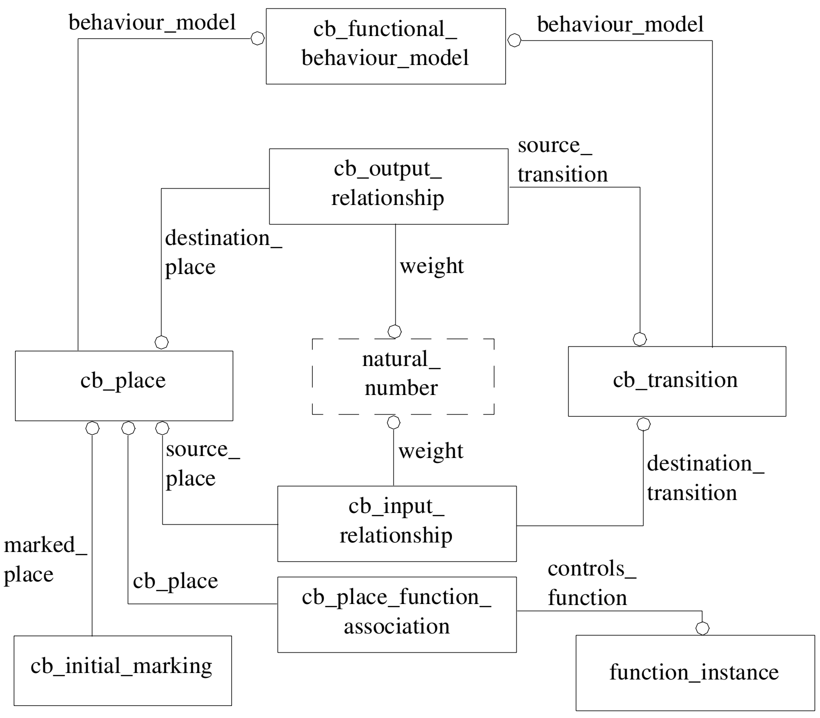


Figure 8.23: Basic causal chain information model

8.15 Causality-Based Functional Behaviour Information Model

The principle for the representation of causal chain models in the information model has been outlined in sections 8.11.1 to 8.11.6. The information model for a basic causal chain is presented first followed by presentation of information model elements defined to support representation of method specific concepts.

8.15.1 BASIC CAUSAL CHAIN INFORMATION MODEL

The basic causal chain information model is very similar to the one outlined in Section 8.11.5. The Petri-net building blocks are represented in the information model as defined in Table 8.1 and the information model is presented in Figure 8.23.

All *cb_place* and *cb_transition* objects shall reference to exactly one behaviour model with their *behaviour_model* attribute. A *cb_place* and a *cb_transition* object may only be related if they reference the same behaviour model.

Table 8.1: Information model mapping to example in Section 8.11.5

Petri-net element	**Information model entity**	**Comment**
Place	*cb_place*	
Transition	*cb_transition*	*A textual attribute allow for capture of any logical condition that shall be fulfilled prior to the firing of the transition.*
Input arc	*cb_input_ relationship*	-
Output arc	*cb_output_ relationship*	-
Multiplicity	-	*Captured with attribute weight for entities cb_input_relationship and cb_output_relationship*
Initial marking	*cb_initial_ marking*	*Captured for each place which shall have a tokens assigned.*
-	*cb_place_ function_ association*	*Relates a place in the causal chain model with the function_ instance object whose activation is controlled by the place.*

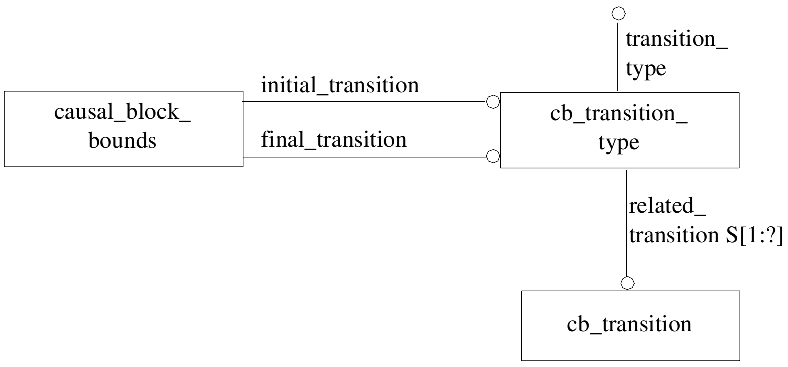


Figure 8.24: Identification of loop type

8.15.2 METHOD SPECIFIC CAUSAL CHAIN MODEL

Four cases where additional information is required to adequately represent method specific constructs in causal chain models were identified in Section 8.11.6. For each of these cases the basic Petri-net model need to be extended with additional entities.

Definition of iteration type

The problem with iteration types is that the Petri-net representation for iteration based on constant and iteration based on condition is identical. A specific tag is required to separate the two cases. The tag cannot be captured directly in the *cb_transition* entity as not all transitions are part of the bounds of a control structure. Instead the entities *cb_transition_type* and *causal_block_bounds* are introduced as presented in Figure 8.24.

- The entity *cb_transition_type* allow for tagging one or more *cb_transition* object being a initial — or final — node in, e.g., an iteration. The *transition_type* attribute captures the type of the tag, e.g, *iteration based on constant* or *iteration based on condition*. The range of tags can be extended to include all control structures identified for causal chains such that the identification of the initial and final element in a control structure is simplified.

- The entity *causal_block_bounds* relates the initial and final elements in a control structure as captured by *cb_transition_type* objects. The value of the *transition_type* attribute of the related *cb_transition_relationship* objects must be identical for valid usage of a *causal_block_bounds* object, i.e., the initial element of a loop may only be related to the final element of a loop.

Thread termination

The FFBD thread termination concept cannot be captured directly in a Petri-net. The arrival of a token to a place cannot influence the behaviour of any other token. Consequently the thread termination concept must be represented outside the Petri-net such that a tool can build the causal chain structure including concurrent threads and then add a modifier that couples the completion of the thread with the immediate termination of all threads in the concurrent execution structure.

Loop exit

Representing a loop exit control is not possible directly in the Petri-net representation as the loop exit will be represented by a normal transition followed by a normal place. Additional information is required to indicate that a particular place object in the Petri-net representation represents the exit from an iteration control structure.

Information model representation for thread termination and loop exit

The solution to the loop exit and thread termination issues can be solved using a common extension to the basic Petri-net model. The solution is to associate a specific tag to a *cb_place* object to indicate that there is additional method specific information related to the object. The information model solution is presented in Figure 8.25.

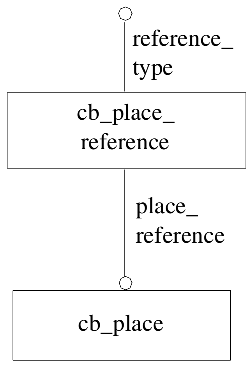


Figure 8.25: Information model excerpt for thread termination and loop exit

The entity *cb_place_reference* is used to provide an indication that method specific information is associated with a *cb_place* object. The semantics of the modification is captured by the *reference_type* attribute. The attribute value may be one of:

- Loop exit: indicating that the *cb_place* represents an exit from an iteration control structure.
- Thread termination: indicating that when a token is placed in the *cb_ place* all other threads concurrently executing threads shall terminate. The thread termination value may only be used in the last *cb_place* object in a thread.

A *cb_place_reference* object may not be associated with a *cb_place* object that controls the activation of a function, i.e., a *cb_place* may not both be a control structure and control the activation of a function.

Function exit condition mapping

Function exit condition mapping as outlined in Section 8.11.6 is not primarily related to limitations in the Petri-nets representation but to the fact that exit conditions at decomposition level *n* in the functional hierarchy must be correctly mapped to completion alternatives at level *n-1*. The problem is similar to that of functional interaction parameter mapping —

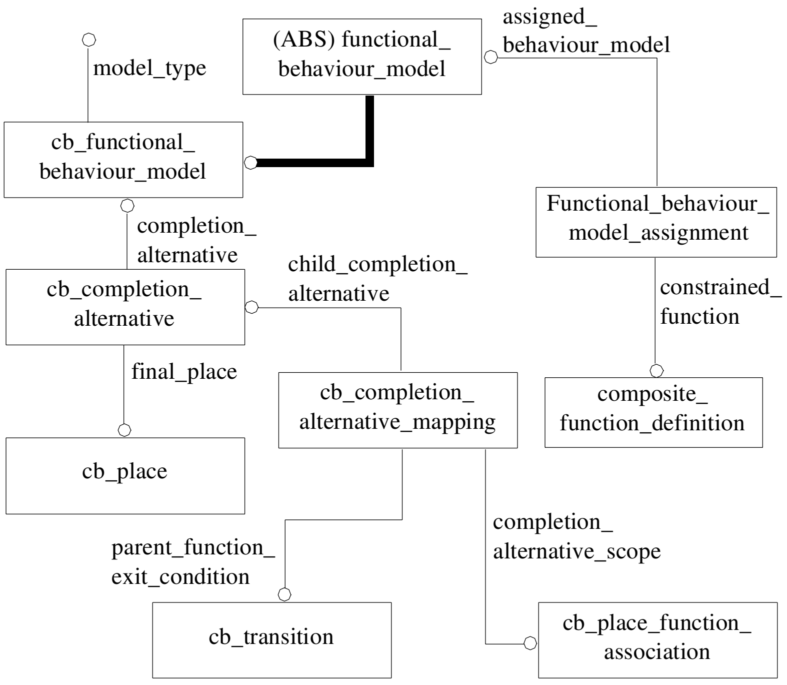
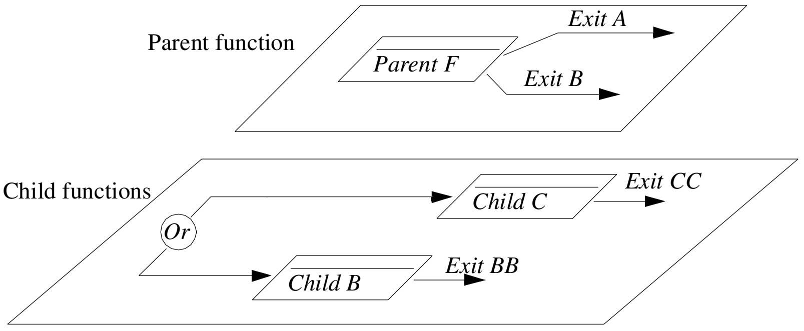

(ABS) functional_ behaviour_model
assigned_ behaviour_model
model_type
cb_functional_ behaviour_model
Functional_behaviour_ model_assignment
completion_ alternative
cb_completion_ alternative
child_completion_ alternative
constrained_ function
final_place
cb_completion_ alternative_mapping
composite_ function_definition
cb_place
completion_ alternative_scope
parent_function_ exit_condition
cb_transition
cb_place_function_ association

Figure 8.26: Information model excerpt for exit condition mapping

an *actual_io_port* object must be explicitly mapped to a *formal_io_port* object. Similarly, a *cb_place* object representing the end of a causal chain within a *composite_function_definition* at composition level *n* must be explicitly mapped to a *cb_transition* object which define the correct execution alternative that shall be selected at level *n-1* (the immediate parent level). Entities from both the functional hierarchy and causal behaviour parts of the information model are involved to capture the correct mapping alternative as indicated in Figure 8.26. The definition of the entities related to exit condition mapping in the figure are presented below:

- *cb_completion_alternative* defines that a *cb_place* object is a exit condition for a causal chain model identified by a *cb_functional_behaviour_model*. The *cb_completion_alternative* will execute when



FFBD excerpt where *exit BB* shall be mapped to *Exit B* and *Exit CC* shall be mapped to *Exit A*

Figure 8.27: Exit condition mapping example

a token is placed in the *cb_place* object referenced by the *final_place* attribute.

- *cb_completion_alternative_mapping* defines the mapping from a *cb_completion_alternative* to the appropriate output cb_transition object for the parent function. This is done by identifying the *cb_transition* object that shall fire when the *cb_completion_alternative* is executed.

Example

The FFBD model fragment in Figure 8.27 is used to illustrate how exit condition mapping is represented in the information model. The equivalent information model instantiation for Figure 8.27 is presented in Figure 8.28. It shall be noted that no less than 43 objects are required to represent what appears to be only 8 distinct elements in the diagram in Figure 8.27. The difference in the number of objects instantiated is primarily due to the fact that the instantiated objects in Figure 8.28 represents the elements required to represent a model in a storage format as compared to the user level presentation in Figure 8.27. Still, it is expected that any tool proprietary representation of the model fragment could be substantially smaller as the storage representation can be tailored to the actual semantics of the tool.

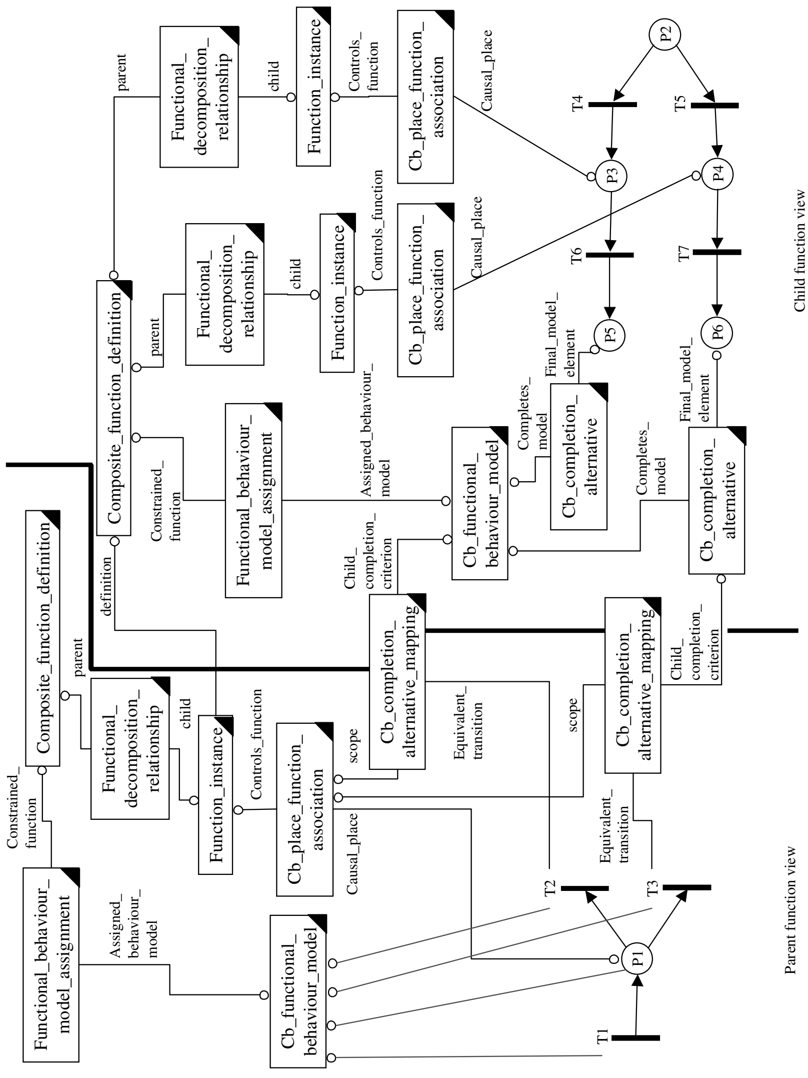


Figure 8.28: Information model instantiation of the FFBD in Figure 8.27

8.16 Summary

The information model support for representing functional architectures has been presented in this chapter. Four aspects of functional architecture is supported:

- Functional hierarchy
- Functional interaction
- Function instruction
- Functional behaviour

Each aspect contain support for representing relevant semantic variants identified. Method specific semantics is representing through a combination of the capabilities offered in each aspect. As a result it is possible to map design data from multiple methods to the information model, without loss of any information. The application of complex mapping functions is required in certain cases.

The diversity of semantics in methods in use for capture system functional architecture suggests that data exchange between tools supporting different methods may result in large parts of the data being modified or lost in the receiving tool. This suggests that the capabilities of functional architecture specification tools used in a data exchange environment must be well understood, and functional architecture models adapted to identified constraints prior to any data exchanges.

Chapter 9
Physical Architecture

This chapter outlines the scope for the system physical architecture model part of the information model and presents the corresponding representations in the information model.

9.1 Physical Architecture

In this thesis the term *system physical architecture* is used to refer to a high level logical view on the overall products, services or processes that comprise a system. The usage of the term in this thesis corresponds to that of Sage and Rouse [132], Lewis [90] and Buede [25].

An element or component in the physical architecture does not represent specifications at an abstraction suitable for product realisation, but represent a requirement view for the realisation or implementation process phase. Moreover, a physical architecture component need not be a manufacturable product, but could be a human operator that needs to be properly trained to interact with the system.

9.1.1 PHYSICAL ARCHITECTURE MODEL SCOPE

The physical architecture part of the information model captures the physical or logical components of a system at a high level of abstraction. Components in the physical architecture represent a requirements view on a system and do not need to completely mirror the set of products and services later selected for realising the requirements expressed. The purpose of the physical architecture is to allow identification of key characteristics of the components, component interaction and interfaces. Two kinds of components are identified:

- Components whose main characteristic is the processing of things (e.g., information, data, material, energy).
- Components whose main characteristic are the transfer of things (e.g., information, data, material, energy). Transfer may not be ideal, i.e., delays or loss of energy may be a characteristic of the component.

The level of abstraction selected for representation of physical components correspond to that of module charts in Statemate [61] and system component entity type in the Core tool [89].

No attempts are made to identify sets of prescriptive properties for a physical component. This is due to the wide range of potential properties that may be captured for a component. Instead any number of properties may be assigned to a component using the property models outlined in Section 5.7.

Since the focus is on a logical view on system components there is no support for detailed property representations such as component shape and engineering analysis properties. Representations suitable for this level of abstraction are available in product oriented STEP Application Protocols, e.g., AP-203 and AP-209.

9.1.2 REUSE OF DESIGN FRAGMENTS

A physical architecture captured is essentially system independent. A set of components identified may be applicable to multiple systems, system versions and viewpoints. On the other hand, the set of requirements and functional architecture elements that the set of components shall realise is

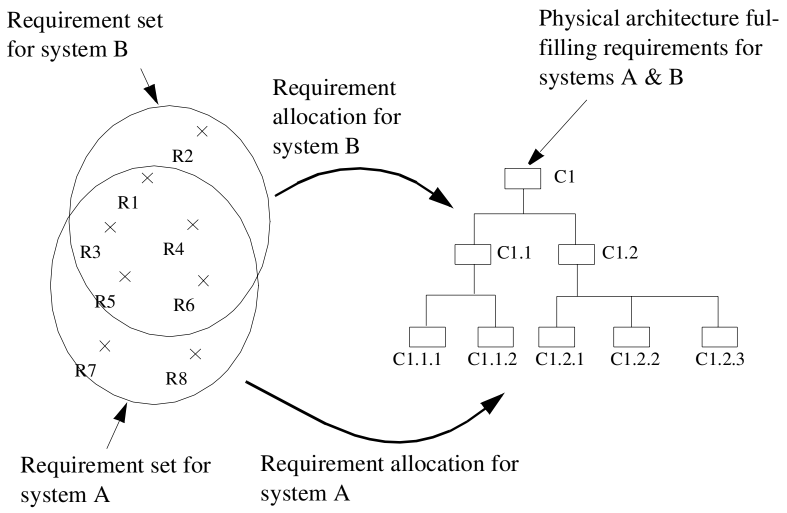

Requirement set for system B
Physical architecture fulfilling requirements for systems A & B
Requirement allocation for system B
R1
R2
R3
R4
R5
R6
R7
R8
C1
C1.1
C1.2
C1.1.1
C1.1.2
C1.2.1
C1.2.2
C1.2.3
Requirement set for system A
Requirement allocation for system A

Figure 9.1: Requirements and physical architecture

system dependent. Consequently, allocation relationships that link other specification elements to a component in the physical architecture, e.g., for requirements or functions, shall be represented such that allocation is represented locally for each system. The relationship between requirements and physical architecture is illustrated in Figure 9.1 where a physical architecture model is common to two systems despite the requirements for systems are not identical.

9.2 Physical Architecture and System Architecture

In the information model there are two related structures for capturing structure related information on a system.

- The system architecture model define the system composition structure
- The physical architecture define the components of individual systems

One difference in the two architecture views is that the system composition structure is capturing relationships between conceptual objects. The system internal architecture is the collection of specification elements assigned to the life-cycle view of a system. The physical architecture on the other hand is the set of physical or logical components identified for the realisation of a single system component. Moreover, the system architecture need not contain any physical architecture elements to be complete.

9.3 Physical Architecture Information Model Structure

The structure of the physical architecture model is similar to that of the functional architecture model. As with the functional architecture there is a system invariant layer for capturing components, component composition, their interfaces. A physical composition structure may not contain any cycles. The system invariant layer may be valid for any number of systems, system versions or viewpoints and capture any number of properties that apply for all usages of a component.

Strict interfaces are enforced for components similar to the functional interaction model. The interfaces of components are explicit and connections between components may only be captured via the interface.

There is also a system specific layer which extends on the invariant layer to capture properties that are system dependent, e.g., weight or availability data, and allocation data, i.e., requirement and functional allocation data as discussed in chapter 11.

9.4 System Invariant Physical Architecture Layer

The information model for capturing the system invariant physical architecture layer is similar to those for capturing requirement and functional architecture information. The structures for capturing version management information are also identical with the exception that *configuration_element* objects representing physical architecture objects shall have the *configuration_element_type* attribute set to the string 'physical architecture element'. This constraint is enforced to ensure that only physical components can be used to build a physical composition structure.

The information model for capturing the system invariant physical architecture of a system is presented in Figure 9.2. Definitions of individual entities are presented below.

- *general_physical_definition*, an abstract supertype of *physical_node_definition* and *physical_link_definition*. The *general_physical_definition* provides access to version management and identification information common to the two subtypes. A *general_physical_definition* type object may be referenced by any number of *physical_instance* objects. Consequently a *general_physical_definition* type object may be used in multiple physical architecture models and multiple *physical_instance* objects may refer to a single *general_physical_definition* objects.
- *physical_node_definition*, represents a physical or logical element whose main purpose is to process things (e.g., data, material or energy).
- *physical_link_definition*, represents a physical or logical element whose main purpose is to transfer things (e.g., data, material or energy).
- *physical_instance*, represents the use of a *general_physical_definition* object within a physical architecture. A *physical_instance* is related to exactly one *general_physical_definition* via its *definition* attribute.
- *physical_composition_relationship*, is the mechanism for relating a *general_physical_definition* object to one of its immediate child components (represented by a *physical_instance* object).
- *physical_port*, represents an element of the interface of an object in the physical architecture model. It is an abstract supertype for the entities *actual_physical_port* and *formal_physical_port*. A *physical_port* does not capture any detail on the nature of the communication over the interface.
- *actual_physical_port*, is an element in the interface of a *physical_instance* object.

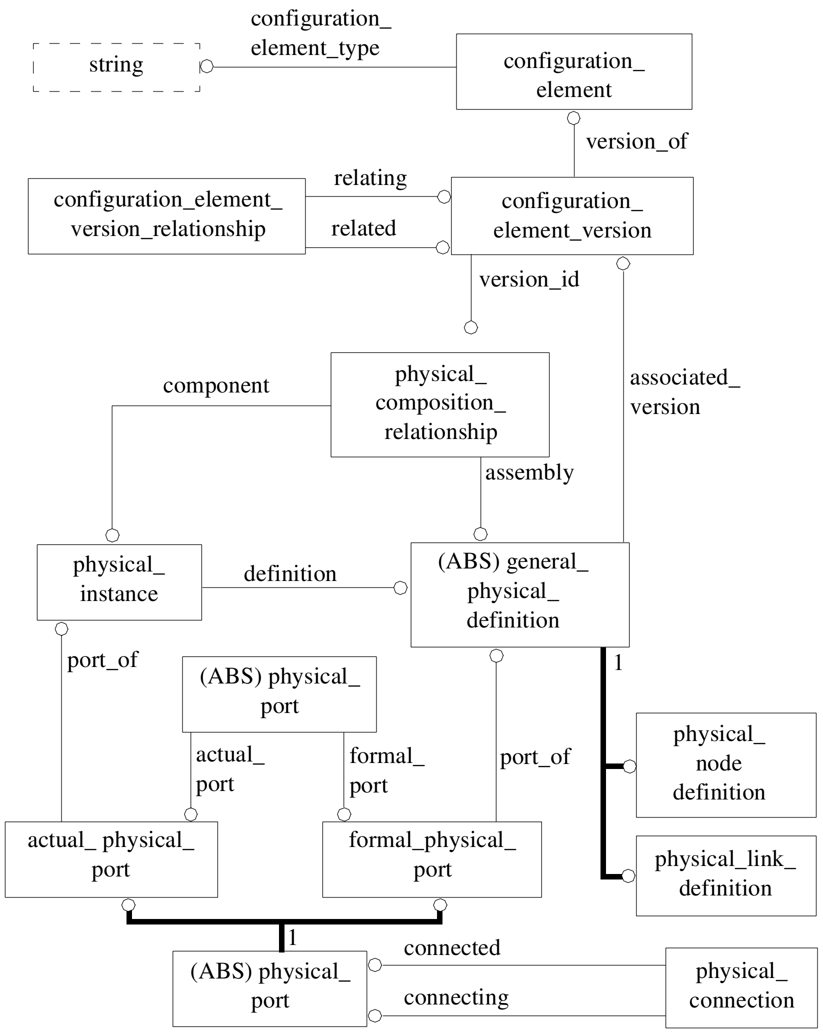


Figure 9.2: System invariant physical architecture model

- *formal_physical_port*, is an element in the interface of a *general_physical_definition* object.
- *physical_binding*, is the mechanism for indicating an ideal connection between an *actual_physical_port* and an *formal_physical_port*. There

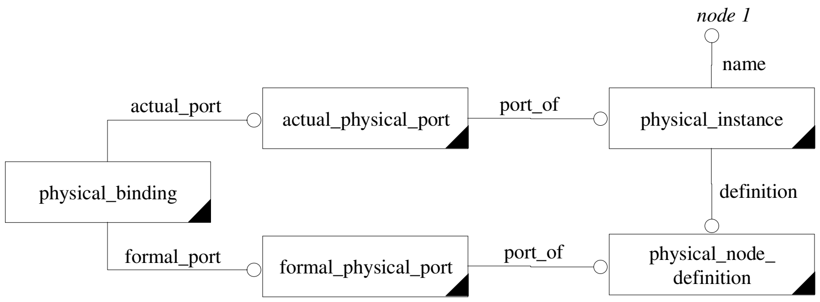


Figure 9.3: Physical_binding example.

are no time delays or energy losses associated with a *physical_binding*. The *actual_io_port* in the relationship is associated with a *physical_instance* object one level higher in the physical composition structure compared with the *general_physical_definition* object related to the *formal_physical_port* in the relationship. Just as for parameter binding for functional elements the *physical_instance* object the *actual_port* object in the relationship is associated with shall refer to the *general_physical_definition* object with its definition attribute the *formal_physical_port* object in the relationship is associated with. The usage of *physical_binding* object is illustrated in Figure 9.3.

- *physical_connection* is the mechanism for indicating connection between a pair of *physical_port* objects at the same level of decomposition in a physical architecture model. Any number of *physical_connection* objects may be associated to a *physical_port* object. The *physical_connection* represent an ideal connection. There are no time delays or energy losses associated with a *physical_connection*.

9.4.1 PHYSICAL COMPOSITION EXAMPLE

This section presents a small example physical composition structure. The components in the example presented in Figure 9.4 represent a trivial sensor system with an integrated processing unit. The composition consists of one CPU component and a sensor assembly containing two sensor compo-

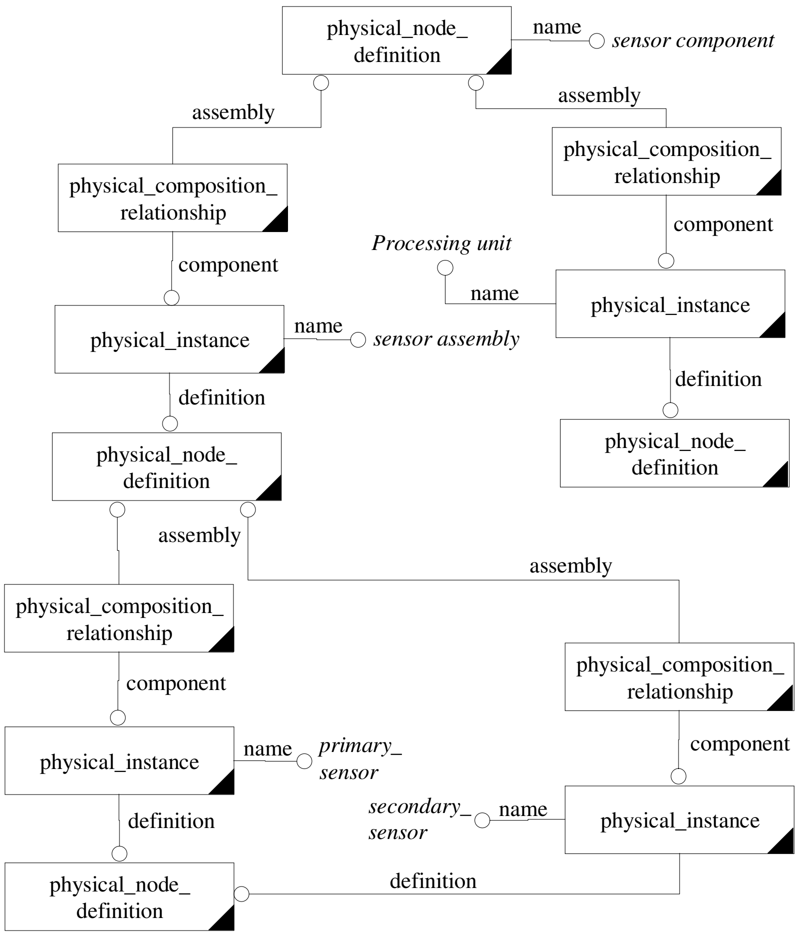


Figure 9.4: Physical composition example for a trivial sensor system

nents. The sensors are considered being identical, so the *physical_instance* objects representing the sensors refers to the same *physical_node_definition* object. Note that all version management information is suppressed in Figure 9.4.

9.5 System Variant Physical Architecture Layer

The system variant physical architecture layer provides the structures for associating a physical composition structure to entities in the system architecture model, for capturing system specific properties of individual physical components and for supporting allocation of requirements and functions onto a physical component.

A specific property and allocation entity, *physical_instance_reference*, is introduced for capturing system specific properties of *physical_instance* objects in a physical composition structure. *physical_instance_reference* objects are related to form a tree structure via *physical_instance_relationship* objects such that each component in the physical architecture is potentially associated with its own unique property placeholder.

A *physical_reference_configuration* relate to all property placeholder objects relevant for a static physical composition model. There may be at most one *physical_reference_configuration* for a static physical composition model and a *physical_reference_configuration* may be valid for any number of systems.

The system variant physical architecture information model part in presented in Figure 9.5. This portion of the information model connects the system architecture part of the model with the system invariant model. The following entities are introduced:

- *context_physical_relationship*, is the mechanism for associating a top-node *physical_instance* object to a system represented by a *system_view_definition* object. The assigned *physical_instance* object either represents an element of the system or an element in the environment of the system.
- *physical_instance_reference*, is a placeholder for capturing system specific properties and allocation relationships for *physical_instance* objects. Any number of *physical_instance_reference* objects may be associated to a *physical_instance* object. Each would capture its own set of system specific properties.
- *physical_reference_relationship*, is the mechanism for representing a parent child relationship between a pair of *physical_instance_refer-*

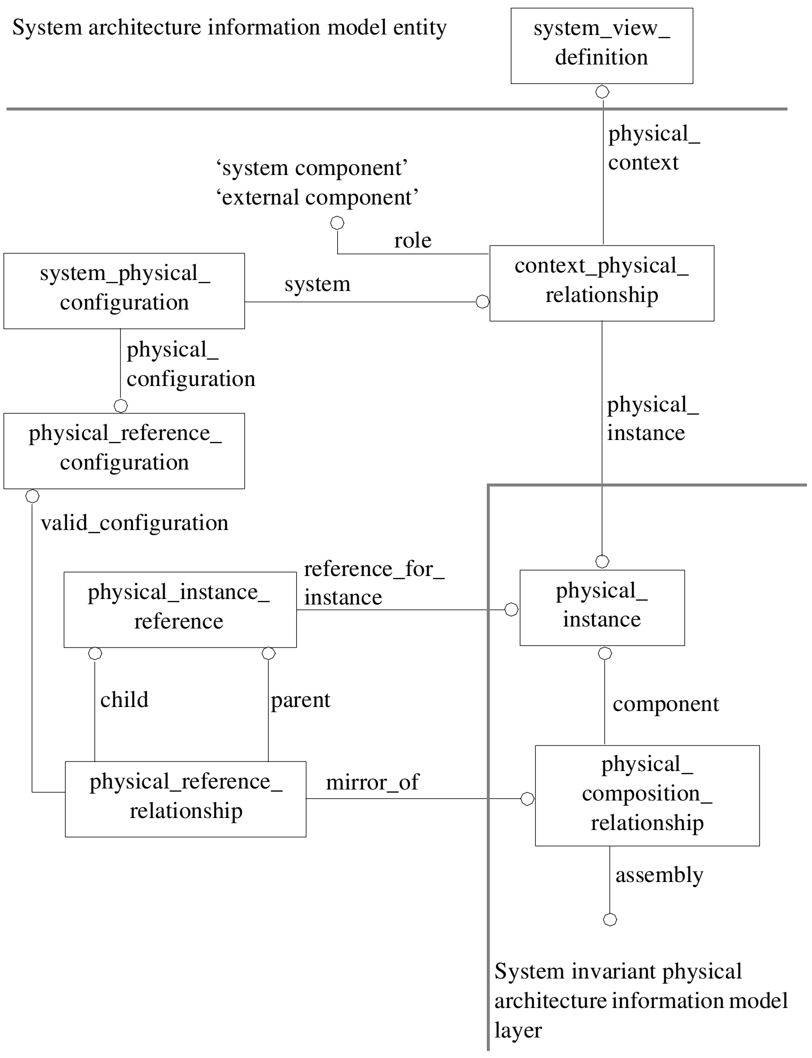


Figure 9.5: System variant physical architecture information model

ence objects. The composition relationship realised by a *physical_reference_relationship* shall correspond that defined in the system invariant physical architecture layer. A *physical_reference_relation-*

ship is part of exactly one *physical_reference_configuration* via its *valid_configuration* attribute.
- *physical_reference_configuration*, represent a set of *physical_reference_relationship* objects for a physical architecture for a system or a set of systems. The purpose is to identify the *physical_instance_reference* objects that are valid for a particular system.
- *system_physical_configuration*, is the mechanism for assigning the set of system specific properties associated *physical_reference_configuration* object to a *system_view_definition* type object.

9.5.1 EXAMPLE

This section illustrates how system variant layer and system invariant layer of the physical architecture information are combined to capture properties and allocation data for a physical model. The model excerpt presented in Figure 9.6 extends on the structure presented in Figure 9.4 to include:

- The association of the physical composition structure to a *system_view_definition* type object. This is performed by the *context_physical_relationship* object at label 1 in Figure 9.6.
- A set of *physical_instance_reference* objects (label 2) to capture properties local to a specific system. No property assignment or allocation data is present in the figure.
- The use of *physical_reference_relationship* objects to form a system specific *physical_reference_configuration*, which in turn may serve as the configuration for a specific system view. This is indicated by the *system_physical_configuration* entity in Figure 9.6. A *physical_reference_configuration* can be assigned to multiple *context_physical_relationship* objects, i.e., the same configuration may be assigned to multiple system view or viewpoints.

In case a system invariant physical architecture structure is assigned to multiple systems with divergent property or allocation assignments then there should be specific *physical_reference_configuration* objects defined for each system. Note that *physical_instance_reference* objects may be

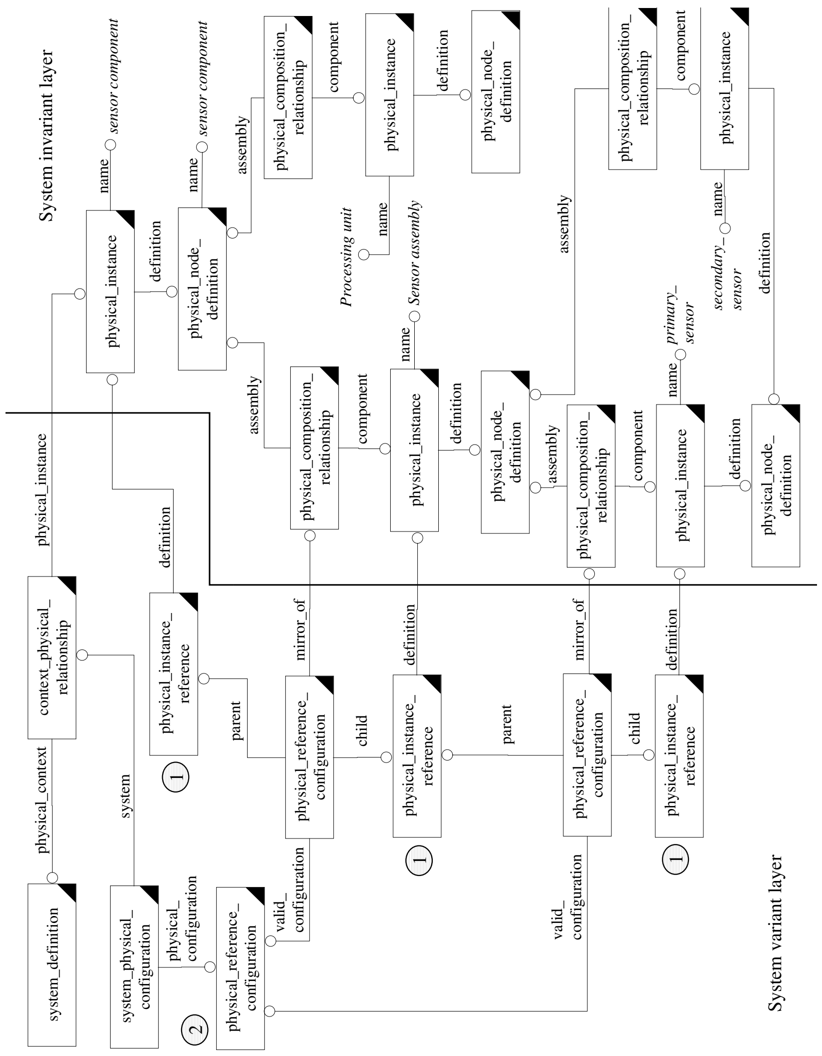


Figure 9.6: System invariant and system variant physical architecture models combined

part of multiple *physical_reference_configuration*. This would be the case if the same property and allocation patters would apply for more than one system.

9.6 Summary

The physical architecture part of the information model supports representation of a physical or logical architecture for a system. System components, their interfaces and how they interact may be captured at a high level of abstraction.

The system variant part of the physical architecture model is further used for allocation of functional and requirement elements onto elements in the physical architecture. This part of the information model is presented in Chapter 11.

Chapter 10 Verification and Validation

This chapter cover the information model support for representing verification and validation statements and how they are related to the system architecture part of the information model.

10.1 Verification and Validation

Verification and validation is a key Systems Engineering activity. Capture of verification and validation procedures that verify that the required capabilities are met are just as important as the requirements themselves [24] [100] [142]. Within this text we use the terms verification and validation as used by Storey in [143]:

- Verification, is the process of determining that a system, component or module, meets its specification
- Validation is the process of determining that a system is appropriate for its purpose.

Identification of verification activities and the definition of procedures for verification and validation (test cases) are performed as an integral part of the system engineering process [24] [100]. We use the term *verification and validation plan* to refer to a collection of information related to verification and validation activities and the term *verification and validation plan item* to refer to a specific element in a verification and validation plan.

In theory verification and validation plans should be defined such that each requirement is verified, i.e., verification activities shall be performed to verify conformance with each requirement statement. However, for a complex system it could be difficult or even impossible to verify and validate all requirements over all system life-cycles. Verification is typically carried out at multiple levels in a system, i.e., verification activities are performed at all levels in the system breakdown structure.

The following assumptions where made for the verification and validation part of the information model.

1. Verification and validation plan items can be captured in any representation supported by the information model.
2. Verification and validation plans can be of any complexity. For customer validation a system demonstration may be sufficient, but verification may also include any number of complex steps or components captured by verification and validation plan items.
3. A verification and validation plan item can verify any number of requirements, or other specification elements in the information model, or a complete system specification.
4. A verification and validation plan item may be applicable to multiple systems, system versions or system viewpoints.

The mechanism for capturing traceability links between requirements and verification and verification plan items is presented in Chapter 11.

10.2 Verification and Validation Information Model

The same static representation structures are selected for product requirements and verification and validation data. The assignment mechanism used to determine whether a requirement item captures verification or validation information or system requirement information. The motivation for this structure is that both product requirements and verification and

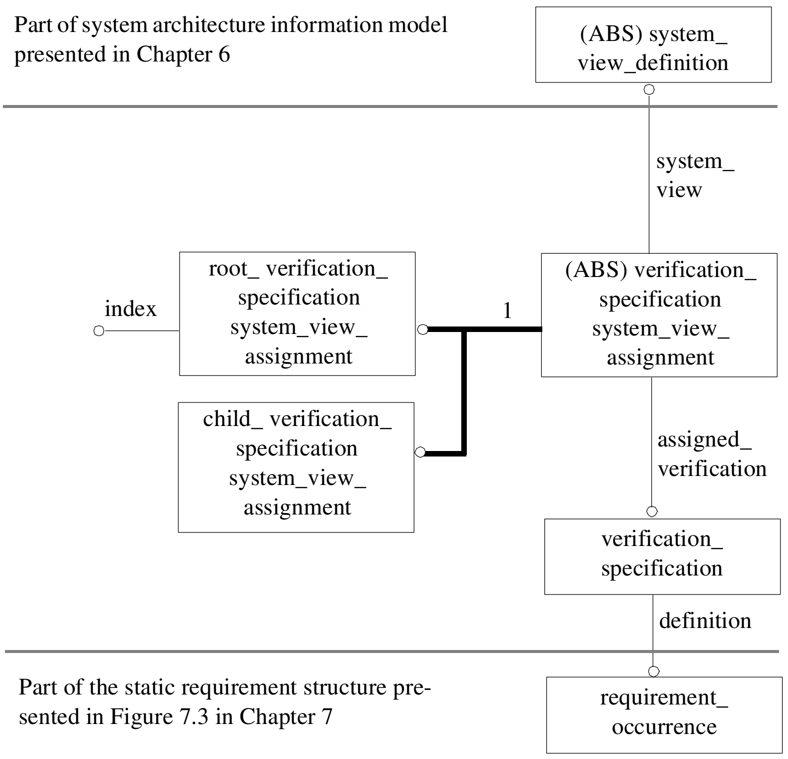

Part of system architecture information model presented in Chapter 6
(ABS) system_ view_definition
system_ view
root_ verification_ specification system_view_ assignment
index
1
(ABS) verification_ specification system_view_ assignment
child_ verification_ specification system_view_ assignment
assigned_ verification
verification_ specification
definition
Part of the static requirement structure presented in Figure 7.3 in Chapter 7
requirement_ occurrence

Figure 10.1: Verification and validation information model

validation requirements share a common static structure. A product requirement is a statement of a capability that may be broken down into more basic components. The same is true for verification or validation plan items. The actual statements captured by requirements and verification and validation plans and properties captured can be expected to be fundamentally different, but the representations for capturing the information are identical. A detailed discussion on the system variant requirement representation model that motivates the selected structure is presented in Section 7.6.

The information model for capturing verification and validation plans for a system is presented in Figure 10.1. The following entities are introduced:

- *Verification_specification* represent a verification and validation plan. It is the placeholder for capture of system specific verification and validation properties. The entity corresponds to the *requirement_instance* entity in the system variant requirement representation model presented in Section 7.6.
- *verification_specification_system_view_assignment* is the abstract supertype for assigning a *verification_specification* object to a *system_view* object. A verification_specification object may only be assigned once to a *system_view* object. The subtypes *root_ verification_specification_system_view_assignment* and *child_ verification_specification_system_view_assignment* perform the assignment for root and child validation and verification plans respectively.

Note that the entities introduced mirror those for representing requirements as presented in Section 7.6.

10.3 Discussion

The verification and validation model is severely limited in the sense that it does neither allow for the representation of the product that shall be verified or validated nor does it allow for the capture of test results. This restriction is made on purpose as the information model does not include support for the detailed product data required to represent the verification configuration. Inclusion of this functionality may be performed as a future extension of the information model through the combination of the information model presented herein, production oriented STEP application protocols and product instance oriented information models from the PLCS project.

10.4 Summary

Planning for verification and validation activities is an important activity in the Systems Engineering process. The information model for specific verification and validation items presented in this chapter can capture items in multiple representations and can relate them to multiple system life-cycle views, system versions or viewpoints. The mechanisms for relating a verification and validation plan to other specification elements are presented in Chapter 11.

Chapter 11 Traceability

Design and decision traceability, i.e., the capture of justifications, motivations for decisions made is of outmost importance for projects developing complex systems. The risk for repeating analysis activities or for taking conflicting design decisions is lowered substantially if design decisions and trade-off studies are properly documented, motivated and available to project participants.

In chapters 6 - 10 specific parts of the information model has been presented in isolation. In this chapter the structures in the information model for capturing the relationship between specification elements and between elements and the engineering process is introduced and motivated.

11.1 Traceability

There are multiple definitions for the term traceability. The term as used within this thesis corresponds with that definition presented in the EIA-632 standard [102].

> Traceability: The ability to identify the relationship between various artefacts of the development process, i.e., the lineage of requirements, the relationship between a design decision and the

affected requirements and design features, the assignment of requirements to design features, the relationship of text results to the original source of requirements.

Maintaining traceability links throughout the engineering process is resource demanding but do provide the groundwork for understanding, e.g., how changes in one system specification view influence other views. The consequences of a modification may be defined easily if traceability links are captured correctly and maintained properly.

Support for capturing traceability can be implemented at different levels of granularity [98]:

- Coarse-grained traceability allow for capture that there exist a relationship between some information residing in compound objects. The precise elements that are involved in each compound cannot be established.
- Fine-grained traceability allow for capture of relationships directly with the relevant objects.

The information model has been designed with the intention to support fine-grained traceability between elements.

11.2 Traceability Dimensions

There are multiple aspects of traceability that shall be considered. In the information model five dimensions of traceability can be identified:

1. *Specification element history traceability*, refers to the capability to capture how a specification element evolves over time, e.g., version and variant management.
2. *Specification element traceability*, refers to the capability to relate specification elements of different types to each other. This dimension of traceability includes, e.g., allocation of requirements to functional and physical specification elements and the allocation of functional elements to physical specification elements.
3. *System composition and viewpoint traceability*, refer to the capability

to relate specification elements at different levels of detail, e.g., traceability between user and system requirements on a system, and to capture how a subset of the requirements on a system is related to the requirements on one of its subsystems. The purpose of this dimension of traceability is to ensure that requirements captured at a specific level of abstraction are properly reflected in other life-cycle representations or subsystem specifications.

4. *Engineering process traceability*, refer to the capability to relate elements of the system specification to activities performed in the engineering process. The purpose is to capture where the elements were create or referenced. This aspect of traceability can also be used to capture design decisions, change management and trade-off analysis data.
5. *Commonality traceability*, refer to the capability to identify specification elements common to multiple system specifications. The benefit of this aspect of traceability is a capability to identify the occurrences of a specific specification element across multiple system specifications.

The details of each traceability dimension presented in detail in the rest of this paper.

11.3 Specification Element History

This traceability aspect capture how a specification element evolves over time and how different specification elements relate. The purpose is to capture version history and identification of, e.g., variant and alternate specification elements. In the information model this aspect is captured by a set of entities common to all concepts under configuration control. The following entities are defined for capturing specification element history traceability: *configuration_element*, *configuration_element_version*, *configuration_element_relationship* and *configuration_element_version_relationship*. Of these entities the two capturing relationships are of interest for capturing specification element history traceability. The information model entities for capturing specification element history is available

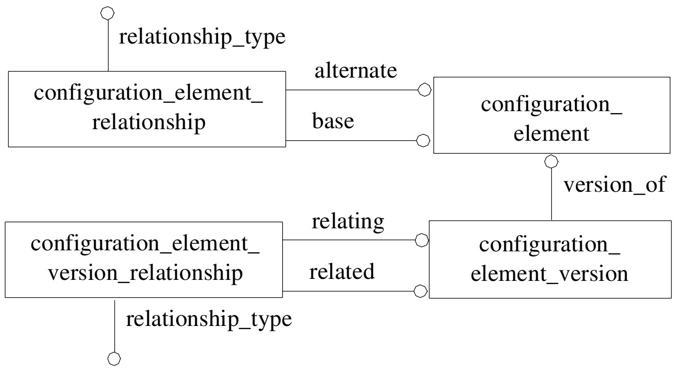

relationship_type
configuration_element_ relationship
alternate
base
configuration_ element
version_of
relating
related
configuration_element_ version_relationship
configuration_ element_version
relationship_type

Figure 11.1: Specification element history information model

in Figure 11.1. Note that the entity *configuration_element_version* is used by all entities with a *_definition* suffix to support representation of relationships between individual versions.

The *configuration_element_relationship* captures any logical relationship between a pair of *configuration_element* objects. The semantics of the relationship is defined by the *relationship_type* attribute. In the information model the following attribute values are defined explicitly defined:

1. 'Variant', the *configuration_element* object identified by the alternate attribute has a substantial degree of commonality with the *configuration_element* object identified by the base attribute.
2. 'Alternative', the *configuration_element* object identified by the alternate attribute may be replaced with the *configuration_element* object identified by the base attribute.

Additional attribute values may be defined to capture additional concepts.

The second aspect of specification element traceability is that concerning how versions of a *configuration_elements* relate to each other. This aspect is captured by the entity *configuration_element_version_relationship*. Three kinds of relationships are supported:

1. 'Revision', a revision relationship indicates a relationship between two revisions, major versions, of a *configuration_element* object.
2. 'Workspace revision', a workspace revision relationship indicates a relationship to or from a minor version of a *configuration_element* object.
3. 'Alternative', an alternative relationship indicates that a pair of *configuration_element_version* objects related are alternate, i.e., it is judged that the two version objects can be used interchangeably within a specification.

The version history structures built by *configuration_element_version* and *configuration_element_version_relationship* objects form a directed acyclic graph.

11.4 Specification Element Traceability

This aspect of traceability concerns the relationships between specification elements objects for a system specification. The intention is to support capture of how different specification elements relate to each other, e.g., that requirement is traced to or allocated to a functional or physical elements. The importance of specification element traceability is underlined in Systems Engineering literature, e.g., Martin [101] and Stevens et al. [142], as well as in Systems Engineering standards, e.g., [126] and EIA-632 [102].

Three types of allocation relationships can be identified in the information model:

- Requirement allocation to functional and physical elements, indicating that the functional or physical element shall fulfil the requirement.
- Functional allocation to physical elements, indicating that the physical element shall realise the functionality specified by the functional element.
- Verification and validation allocation to requirements, functional and physical elements. This allocation relationship indicate that a require-

ment, functional or physical element shall be verified or validated against a specific verification and validation object defined.

The characteristics of each allocation relationship and the implementations in the information model are discussed below.

11.4.1 REQUIREMENT ALLOCATION

A requirement allocation relationship is a binary relationship that relates a requirement to a specification element indicating that the requirement shall be fulfilled by the element. This view on requirement allocation implies that a set of requirements is guiding the development of the functional or physical architecture. Alternatively the relationship capture that the specification element fulfils the requirement. This view on requirement allocation indicates that a functional or physical architecture model is used to guide the identification of requirements.

Requirement allocation may also be made relative to pairs of elements, e.g., for capturing the temporal requirements on functions. A requirement may state that a pair of functions shall terminate within a specific interval or that there shall be a minimal or maximal amount time elapsed from the activation of one function to the termination of another.

The requirement being allocated may be leaf requirements or composite requirements. If a composite requirement *r* is allocated to a functional or physical element then all child requirements to *r* are allocated.

Likewise, a requirement may be allocated to a composite or leaf objects of the functional or physical architecture of a system. Allocation to a composite object implies that no guidance is given in the specification on how the requirements shall be fulfilled by child objects in the composition structure.

Requirement allocation is system specific. The fact that a requirement is allocated to, e.g., a functional element within one system does not imply that the allocation holds for all systems to which the requirement and functional element are assigned. Consequently requirement allocation is performed on the system variant views of the requirement, functional and physical architecture information models.

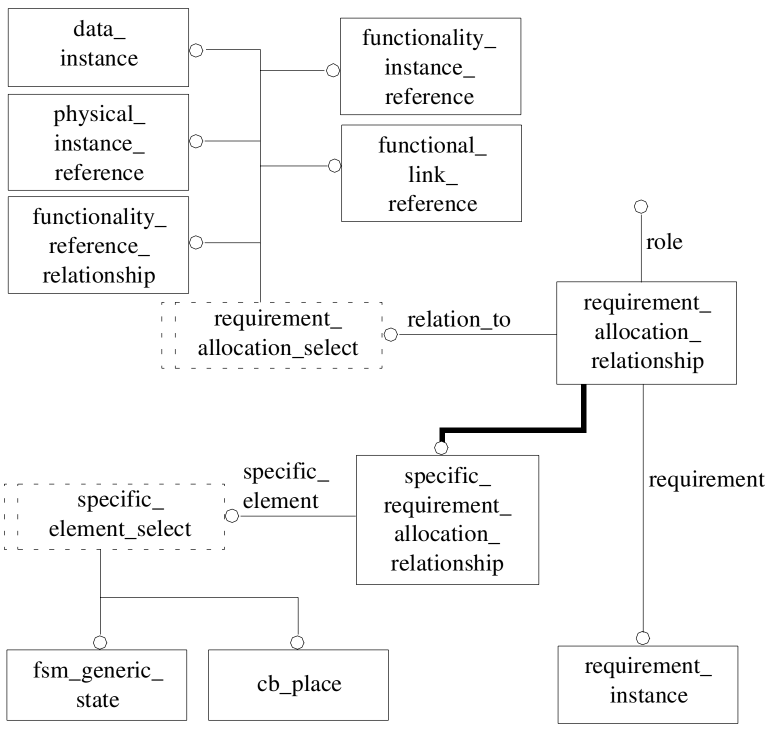

data_ instance
physical_ instance_ reference
functionality_ reference_ relationship
functionality_ instance_ reference
functional_ link_ reference
requirement_ allocation_select
relation_to
role
requirement_ allocation_ relationship
specific_ element
specific_ element_select
specific_ requirement_ allocation_ relationship
requirement
fsm_generic_ state
cb_place
requirement_ instance

Figure 11.2: Requirement allocation information model entities

The requirement allocation part of the information model is presented in Figure 11.2. The following entities are introduced to capture requirement allocation information:

- *requirement_allocation_relationship*, is the mechanism for relating a *requirement_instance* object to an element in the functional or physical architecture model. The *role* attribute is used to indicate whether the relationship captures allocation of a requirement to the functional or physical component or fulfilment of a requirement by the functional or physical element. The *requirement_allocation_relationship* entity may only be used to relate objects assigned to the same system specification.

- *specific_requirement_allocation_relationship*, is a subtype of the *requirement_allocation_relationship* entity and the mechanism for allocating requirements to elements capturing functional behaviour within a function, i.e., a *fsm_generic_state* object within the scope of a finite state machine (*fsm_model*) or a *cb_place* object within a causal behaviour model.

The *requirement_allocation_relationship* entity support the alternatives for requirement allocation to objects of type:

- *data_instance*, for allocation of requirements to objects used to represent the items in functional interaction.
- *physical_instance_reference*, for allocation of requirements to objects representing elements in the physical architecture description of a system.
- *functionality_instance_reference*, for allocation of requirements to objects representing elements in the functional architecture description of a system.
- *functionality_reference_relationship*, for allocation of, e.g., temporal requirements, to a pair of *functionality_instance_reference* objects.
- *functional_link_reference*, for allocation of requirements to a *functional_link* object.

Requirement allocation example

The information model part for requirement allocation is illustrated in Figure 11.3 where a requirement is allocated to a function within the functional architecture description for a system. The *requirement_allocation_relationship* object performing the allocation is indicated by label 1 in the figure.

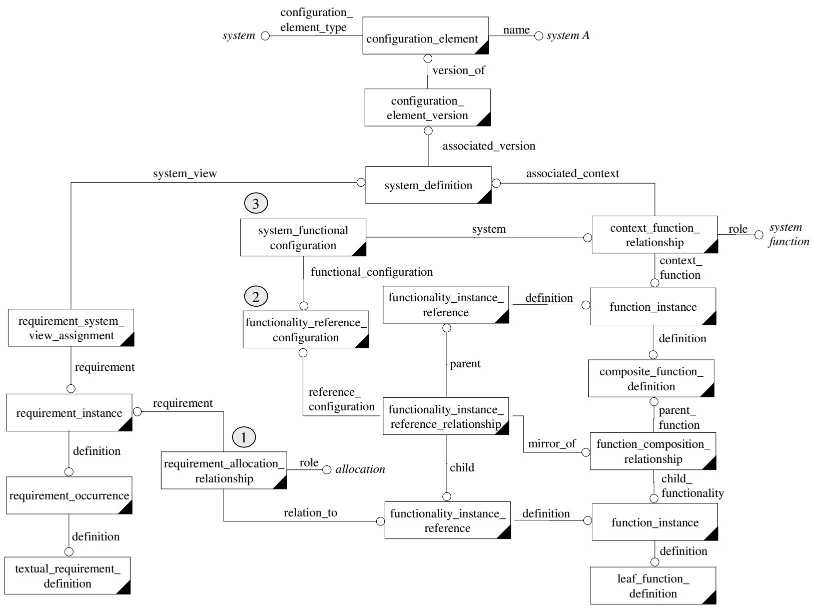

configuration_ element_type
system
configuration_element
name
system A
version_of
configuration_ element_version
associated_version
system_view
system_definition
associated_context
3
system_functional configuration
system
context_function_ relationship
role
system function
context_ function
functional_configuration
2
functionality_reference_ configuration
functionality_instance_ reference
definition
function_instance
requirement_system_ view_assignment
requirement
definition
composite_function_ definition
parent
reference_ configuration
requirement_instance
requirement
functionality_instance_ reference_relationship
parent_ function
1
mirror_of
function_composition_ relationship
definition
requirement_allocation_ relationship
role
allocation
child
child_ functionality
requirement_occurrence
relation_to
functionality_instance_ reference
definition
function_instance
definition
definition
textual_requirement_ definition
leaf_function_ definition

Figure 11.3: Requirement allocation example

Note that the allocation illustrated in Figure 11.3 is only valid for a specific *functionality_reference_instance* object. The mechanism for association allocation information to a specific system is through the *functionality_reference_configuration* object at label 2 and the *system_functional_configuration* at label 3 in the figure. If the functional architecture part of the example is part of the functional architecture of another system specification then there may be a different set of allocations made.

11.4.2 FUNCTIONAL ALLOCATION

Functional allocation relates elements in the functional architecture description of a system to elements in the physical architecture description of the same system. A functional allocation relationship is a binary relationship either relates a functional element to a physical element or a functional connection element to a physical element. The allocation indicate

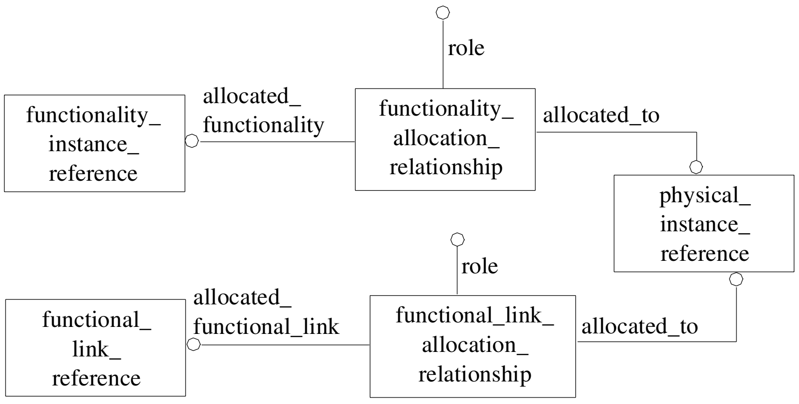

role
functionality_instance_reference
allocated_functionality
functionality_allocation_relationship
allocated_to
physical_instance_reference
role
functional_link_reference
allocated_functional_link
functional_link_allocation_relationship
allocated_to

Figure 11.4: Functional allocation part of information model

that the physical element shall display the functionality defined by the functional element or the physical element has been found to exhibit the functionality indicated by the functional element in the allocation. Functional allocation is, just as requirement allocation, system specific.

The functional allocation part of the information is presented in Figure 11.4 and the entities introduced are presented below.

- *functionality_allocation_relationship*, is the mechanism for allocating a functional element onto a physical element of a system specification.
- *functional_link_allocation_relationship*, is the mechanism for relating a *functional_link* to the *physical_instance* object that shall enable the interaction captured by the *functional_link*. The entities related are of type *physical_instance_reference* and *functional_link_reference* as the allocation is system specific.

11.4.3 VERIFICATION AND VALIDATION ALLOCATION

Verification and validation allocation relate verification and validation plan elements to the requirement, functional or physical architecture elements that the verification and validation plan objects shall verify or validate.

Like for requirement and functional allocation the he elements related shall be assigned to the same system specification. The structure of the verification and validation plan allocation model is identical to that of requirement allocation and not presented in further detail here.

11.5 System Composition and Viewpoint Traceability

The System composition and viewpoint traceability dimension consider relationships between specification elements assigned to different systems, system life-cycle views and viewpoints. The purpose is to ensure that the different views are consistent and relationships between individual elements in different views are captured explicitly. Note that the purpose of this traceability dimension is to capture relationships between views on the same system or between systems that are related via the system composition structure. This is in contrast with the specification element traceability dimension outlined in Section 11.4 which provide for traceability within a single system life-cycle view or viewpoint.

In the information model this traceability dimension is captured using two mechanisms — coarse level traceability and explicit traceability relationships. The two aspects are further outlined below.

11.5.1 COARSE LEVEL TRACEABILITY

Coarse level traceability support is enabled through the structures for capturing system internal and external architecture as outlined in Chapter 6. These include relationships for defining system-subsystem, viewpoint-system and viewpoint-viewpoint relationships. These relationships provide a coarse granularity traceability mechanism. For instance, a modification in a viewpoint could potentially have an impact to other specification views on the system. However, the exact nature of the impact on individual specification elements cannot be determined from these relationships.

11.5.2 EXPLICIT TRACEABILITY RELATIONSHIPS

The second traceability aspect supported within this dimension is through explicit traceability relationships between specification elements assigned to different system views. In the information model this is only supported for requirements objects. The motivation for this restriction is that traceability for functional and physical elements are captured implicitly via relationships captured on *system_view_definition* type objects. This is due to the fact there is at most one functional architecture model and physical architecture model respectively for each *system_view_definition* object.

The intention with the support for explicit traceability relationships in the model is to allow the capture of relationships between requirements captured with different system life-cycle views or different systems. The relationship shall be used to provide traceability to answer the following questions:

- What is the relationship between a pair of requirements captured in different system viewpoints?
- How is a pair of requirements captured in different system life-cycle views defined at different levels of abstractions related?
- What is the relationship between a requirement captured for a system and a requirement on one of its subsystems?

11.6 Engineering Process Traceability

The engineering process traceability dimension considers the relationship between engineering activities and data, i.e., capture of information relating to the context where individual specification elements were captured, references or modified. In the information model this dimension considers three aspects:

- Establishment of links between engineering activities and Systems Engineering data for capture of the data that was used as input or generated as output as a result of work in the engineering process.
- Capture of change management information in the form or engineering change requests and engineering change orders

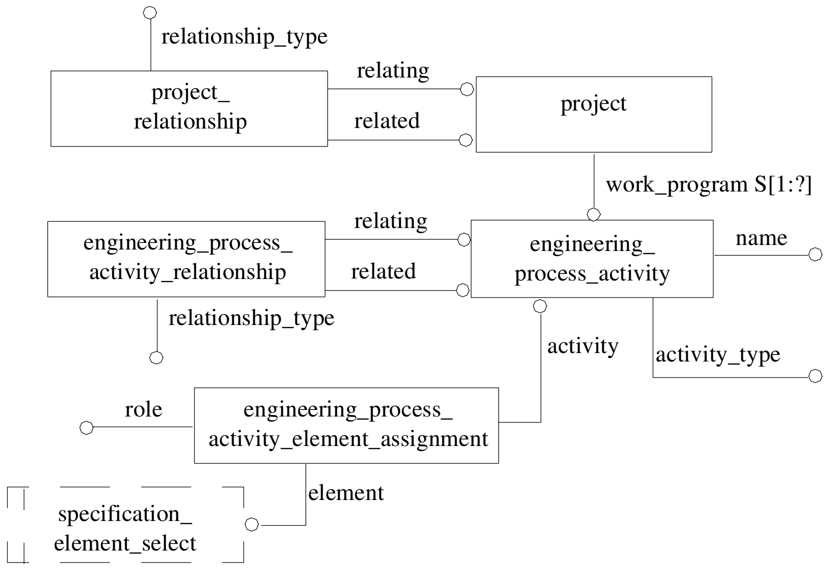

relationship_type
project_ relationship
relating
related
project
work_program S[1:?]
engineering_process_ activity_relationship
relating
related
engineering_ process_activity
name
relationship_type
activity
activity_type
role
engineering_process_ activity_element_assignment
element
specification_ element_select

Figure 11.5: Engineering process activity information model

- Capture of engineering justifications and assessments for design decisions made in design data.

Each area of the information model is outlined further below.

11.6.1 ENGINEERING PROCESS ACTIVITIES

The objective with the engineering process activity part of the information model is to allow for capture of the set of information available as input and reference for a particular activity in the engineering process and the resulting output from the activity. Thus providing the reference to the information that where considered for analysis or design at a particular point in time in the process.

The entities in the engineering process activity information model is presented in Figure 11.5 and defined below:

- *Project* is the representation of a undertaking of work including some aspect of system engineering activities
- *Project_relationship* is the mechanism for relating two *project* objects with the purpose of indicating some kind or relationship, e.g., projects handling the development of two system variants.
- *Engineering_process_activity* is the representation of an activity undertaken by engineers in the engineering process. The type of the activity is identified by the *activity_type* attribute. An activity object is intended to relate to, e.g., an analysis or design activity in the engineering process. Administrative information such as time frame and people involved in the activity may be captured.
- *Engineering_process_activity_relationship* is the mechanism for relating two *engineering_process_activity* objects. Two types of relationships may be captured, as defined by the *relationship_type* attribute, either a parent-child or a sequential relationship.
- *Engineering_process_activity_element_assignment* is the mechanism for assigning specification elements to an *engineering_process_activity* object. Depending on the value of the attribute role the relationship may indicate input, output or reference material for the activity.

The primary objective with the entities is to capture project activities as they evolve as opposed to as planned. Support for project management activities such as project planning, scheduling and resource allocation is not within the scope of the information model.

11.6.2 CHANGE MANAGEMENT

Management of change is key to the development of any complex system, regardless of domain. There is substantial risk in modifying an approved specification unless proper investigations are carried out to ensure that the modification will not have any adverse influence on the system under specification or any systems interfacing with the system. Change management is primarily a process, but specific entities need to be included in the information model to allow for capture of the motivations for performing, or not performing a change in an approved specification. Five concepts are supported by the change management part of the information model

1. The capture of a *critical issue*, a note or report identifying the existence of a potential problem associated with one or more specification elements.
2. The capture of a *change request*, a formal request to perform a change based on one or more *critical issues*.
3. The capture of the result of impact analyses performed to identify consequences of implementing a modification based on a *change request*.
4. The capture of an approval to implement a change, a *change order*, in accordance with a *change request*.
5. The capture of a *change report*, a summary of the changes implemented as a result of one or more approved *change requests*.

The change management part of the information model may be combined with the engineering process activity part to capture both the information associated with change management and the change management process, i.e., investigations of the effects of a potential change and the decision on a particular change may be represented by the engineering process activity part of the information model. This allow for the capture of why a decision was taken as well as the process in which the decision was taken.

11.6.3 JUSTIFICATIONS AND ASSESSMENTS

It may not always be possible or desirable to include all information related to a specification element within that specification element. Supplementary information may be provided to facilitate the understanding of why information has been captured as is and why a particular formulation or design solution has been judged positively or negatively. The information model contains two constructs for capture of this kind of information:

- Justification, a justification is a textual motivation on the quality of some aspect of one or more specification elements. A justification may be related to other justifications.
- Assessment, an assessment is similar to a justification with the discrimination that it is a time stamped evaluation on the status of a particular specification element, e.g., element risk or completeness.

Assessment and justification objects may be assigned to a wide range of objects in the information model. They provide a basic mechanism for providing contextual information to individual specification elements.

11.7 Commonality Traceability

Commonality traceability is the ability to identify and track that identical or related specification elements are in use in multiple system specifications or system views within the same specification. This traceability aspect is supported by the basic representation for specification elements in the information model. If individual specification elements are reused then it is possible to identify, e.g.:

1. The system versions or systems a particular specification element is assigned to. This allow for identification of the potential consequences of a modification to a specification element.
2. The evolution of a specification in terms of the versions of individual specification elements assigned. This allow for the identification of the difference between two versions of a system specification.
3. The level of commonality between system variants.

Commonality traceability will of course not be automatic, but requires the engineers in charge of creating and maintaining specifications to use existing specification element data whenever the situation arises where an identical copy of an element is to be used within a specification. The full implementation of this traceability dimension will require substantial effort for identification and categorisation of specification element data, but it appears that there is a huge potential for reuse of specification element data in industries developing and maintaining multiple variants of a base system.

11.8 Summary

This chapter has presented the information model support for relating different concepts in the information model to each other. The traceability support allow for capture of types of relationships between elements in the information model:

- Logical relationship within a system specification.
- Relationships for capturing the evolution of a specification element over time through the capture of new versions and variants of the initial element.
- Capture of relationships between Systems Engineering data elements allocated to different views of a system or different systems.
- Capture of relationships between Systems Engineering data element and the engineering process where the data elements were created, modified or referenced.
- Capturing the commonality in terms of specification elements assigned to system variants and versions.

PART IV

Evaluation

Chapter 12 Evaluation

This chapter presents the activities undertaken to evaluate the appropriateness of the information model. Quantitative methods are not applicable for the evaluation of an information model. There is no absolute truth or proofs that can be applied for evaluation of a model. The approaches undertaken to evaluate the quality and scope of the information model presented in this thesis has followed two paths.

- Evaluation through peer review within the supporting projects, INCOSE and ISO 10303. Results from this evaluation activity address the questions on the appropriateness of the scope and the selected architecture for the information model.
- Evaluation through the implementation of data exchange interfaces to Systems Engineering tools and exchanges of system specifications within and across organisations. In this evaluation activity the focus is on whether the information model is adequate for its purpose — Systems Engineering data representation and data exchange. The main questions are: Can the information model adequately represent information generated in computer tools used by systems engineers and whether tool interfaces can be developed with a reasonable expenditure of resources?

The description and outcome of the evaluation activities are presented in more detail in the following sections.

12.1 Peer Feedback

Over the past years the information model has been presented at a number of conferences and workshops as well as in ISO workshops. Feedback has been varied and dependent of the views of each individual reviewer. This is not surprising since there are no formal proofs for the correctness of an information model. Extensions may always be called for and the suitability of the guidelines used in model development can be debated. The main arguments raised against the model have been:

1. The information model does not support real Systems Engineering
2. The scope of the information model is too limited to be of value for Systems Engineering
3. The information model has a software engineering bias
4. The information model is too complex
5. The architecture of the information model is not conformant with the modular STEP architecture

Each argument is discussed in the sections below.

12.1.1 WHAT IS REAL SYSTEMS ENGINEERING?

The wide scope of Systems Engineering is presented in Chapter 3 along with motivations for restrictions made to the scope of the information model. Practitioners with other priorities in Systems Engineering may not agree with the restrictions made. However, we believe that the information model is supporting data exchange requirements for a substantial and important subset of the total scope for Systems Engineering data representations.

12.1.2 TOO LIMITED SCOPE

A similar criticism to that above is that the scope of the information model, while the content of the information model is essentially sound [33], is too limited for real industrial use. The main criticism has been the lack of comprehensive support for project management data and structures suitable for Systems Engineering as performed in the detailed design and realisation phases of the engineering process.

Extending the information model to support new domains is always an option. However an increase in scope will also increase the time required to reach agreement on the information model. Clearly development time must be balanced against the benefit provided by a standard. Standards do have a limited validity and are updated at regular intervals. Moreover, the STEP framework is structured such that extensive standards can build on more basic ones. Consequently, the information model can always be extended to include support for roles or aspects not currently included.

12.1.3 SOFTWARE ENGINEERING BIAS

A point that has been made is that the information model is biased to tools used by software engineers rather than Systems Engineering tools. This criticism has been raised against the functional architecture model, especially for the structures supporting finite state machines and the explicit structure selected for functional interfaces [113]. This comment seems to be due to a more restricted view on the set of tools used to support Systems Engineering or a preference for a specific set of reference tools. Motivation for the selected approach is presented in chapters 3 and 4.

12.1.4 TOO COMPLEX MODEL

There have been a number of comments on the large number of entities in the information model coupled with proposals for reducing the model size. The size of the model is not accidental but a consequence of the modelling guidelines presented in Chapter 4. A reduction of model size is possible by relaxing or dropping any of the guidelines presented. Such a decision will reduce model size, but also model capabilities. Three straightforward alternatives to scaling down the model can be identified:

1. Restrict the capabilities to use specification elements in the context of more than one system specification.
2. Relax the rigour of the information model through the removal of support for method specific modelling constructs.
3. Definition of a model supporting what is perceived to support state of the art Systems Engineering only. Thus removing any support for what is perceived to be inferior methods.

The first alternative shall be considered in case the perceived cost for implementing tool interfaces based on the information model is higher than the perceived benefits. This is a subjective decision that shall be considered in the light of the implementation experiences presented in Section 12.3.

The second alternative cannot be advised as long as there is an ambition to make the information model method and tool independent. If method independence remains an objective it is important that the information model can correctly represent method specific constructs, or information will be lost when tool data is exported to the information model format. I.e., mapping functions belonging to the generalisation class (as defined in Section 2.4) will be used when mapping from tool representations to information model ditto. As a consequence there the risk that modifications introduced by mapping functions are not detected when data is imported into another tool. See the discussion on mapping functions in Section 2.4.

The third alternative is also not advisable if the objective with a data exchange information model is to enable exchange between heterogeneous Systems Engineering tools. If restrictions are included in the information model in order to only support a specific method then there is a substantial risk the restrictions made are too constraining for most potential users of the information model.

12.1.5 WRONG ARCHITECTURE

The STEP architectures have been outlined in Chapter 2. Since 1999 there has been a move away from the monolithic application protocol architecture in favour of a modular architecture. A central theme in the criticism against the information model has been that it still adheres to the monolithic application protocol architecture.

The selection of architecture for the information model is mainly a political issue. There are no differences in what can be expressed in the two architectures. The justification for keeping with the monolithic architecture is that there were no interface development tool support available for the modular architecture when validation activities for the information model was initiated in the autumn of 2000. Still, the modular architecture appears to be the future within STEP and transformation of the information model into STEP application modules is likely. There appears to be no technical problems associated with transforming the information model from the monolithic to the modular STEP architecture.

12.2 Evaluation Through Tool Interface Implementation and Data Exchange

Five revisions of the information model has been developed and submitted for review to the AP-233 working group at ISO 10303. Four of the model revisions have been validated through the development of tool data exchange interfaces and through exchange of real design data. A summary of the evaluation results for the last revision is presented along with information model relevant feedback. Two levels of evaluation are considered in this thesis.

1. Feedback from the interface implementers regarding the problems realising the tool import and export interfaces including mapping from tool specific to the corresponding information model representations. This feedback respond to the question “Is the information model appropriate for Systems Engineering data exchange from the perspective of a specific tool?”
2. Feedback from users of the interfaces in their day-to-day work. This

feedback respond to the question "Is the information model coverage extensive enough for realising data exchange interface in an industrial setting?".

The presentation of tool interface development and validation scenario results herein is based on information collected and documented within the SEDRES-2 project by other project participants. They are presented in this thesis as they provide objective feedback on the quality of the information model. Additional comments and clarifications have been added by the author to evaluate the severity and consequences of individual comments and observations.

12.3 Tool Interface Implementation

The following tools and tool interfaces have been developed to validate the fifth information model revision. The tool selection was based on the tools in use by the partners in the SEDRES-2 project as presented in Table 12.1.

Table 12.1: Tool interfaces developed

id	Tool name	Implementer	Import	Export
1	Statemate Magnum	BAE Systems	√	√
2	Teamwork	Conformics	√	√
3	StP	EADS LV	√	√
4	Doors	EADS LV	√	
5	MatrixX/Systembuild	BAE Systems	√	
6	Demanda 2	TU Clausthal	√	√
7	Labsys	EADS LV		√
8	Express AP-233 data server	EuroSTEP	√	√

Tools 1 to 5 in Table 12.1 are commercially available and tools 6 to 8 are in-house tool used by respective organisation. Demanda 2 and the EuroSTEP Express AP-233 data servers are of special interest as they are implemented directly on top of the information model. As implemented they have a complete coverage of the information model, although not all aspects were used actively.

12.3.1 INTERFACE DEVELOPMENT — SETUP

The interfaces were developed by independent developers within the respective organisation. All the developers had previous experience with the EXPRESS language and EXPRESS development tools. In some cases they also had experience with interface implementation for previous revisions of the information model. The information provided to each developer was limited to:

- Formal information model documentation presented in the style mandated by STEP.
- Documents describing and motivating the information modelling philosophy.
- Development guidelines describing parts of the information model in detail together with sample instantiations.

The documentation made available to developers can be considered equivalent with, although less mature compared with what can be expected for a completed STEP standard. In addition the developers had access to the author for handling cases where ambiguities and errors were discovered in the information model.

It was agreed that modifications to the information model to correct errors and mistakes was to be accepted in the early phases of the development exercise provided the modification was agreed by all partners in the project. All in all four model information model revisions were released to tool interface developers in order to correct minor modelling mistakes.

Interface developers were also encouraged to communicate with their peers to perform early tests on their interfaces to detect errors and inconsistent usage of the information model. It is our estimate that the functions provided by this communication equate that of a known good reference implementation that is likely to be available for a standardised data exchange information model.

12.3.2 TOOLS VS. INFORMATION MODEL SCOPE

The coverage of individual tools compared with the scope of the information model is presented in Table 12.1. In the table the tools are identified via the identifier assigned in Table 12.1.

Table 12.1: Tool interface data coverage vs. information model scope

Information model conceptual group	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
System architecture	√	√	√	√	√	√	√	√
Engineering process						√		√
Configuration management	√	√	√	√		√	√	√
Requirement representation	√			√		√	√	√
Functional architecture	√	√	√	√	√	√	√	√
Physical architecture				√		√		√
Requirement allocation				√		√		√
Functional allocation				√		√		√
Presentation information	√	√	√		√	√		√
External document						√		√
Administrative information					√	√		√
Properties		√				√		√
Data types	√	√	√	√	√	√		√
Classification						√		√

A tick in respective category in the table indicate that there is support in the interface for some, but not necessarily all entities in the group.

12.4 Tool Interface Development Validation Results

This section summarizes the evaluation from tool interface developers. The evaluation has been compiled from interviews with developers and via the tool interface development document published by the SEDRES-2 project [74] and in more detail in papers by Eckert and Johansson [44] and Scott et al. [136]. Results of validation from the tool interface development are presented from five perspectives below.

12.4.1 INFORMATION MODEL COMPLETENESS

In all cases the interface developers where able to define mapping functions for the main data structures within the tools to the structures in the information model [74]. There were no cases where interface developers were forced to define mapping functions between tool and information model representations not belonging to the equivalence class, except for areas where conscious restrictions had been made in the information model scope, e.g., the lack of support for representing textual computer interpretable languages.

12.4.2 REUSE SUPPORT

The interfaces developed for the commercially available tools did not provide support for specification element reuse. Tool developers where forced to implement mapping functions for generating data not directly available in the tool format for export interfaces and code for synthesizing the tool specific structures for import interfaces. In both cases the mapping functions belong to the equivalence class so the mappings performed in the interfaces did not alter the semantics of the data exchanged.

It was commented that a lot of objects had to be created to represent tool data in the information model format. This did complicate interface development and also did have a negative impact on interface performance, especially for tool interfaces implemented on-top of database systems [75].

12.4.3 CONFIGURATION MANAGEMENT SUPPORT

Only three of the commercial tools used in the validation scenarios had some level of support for version management - Doors, Teamwork and Statemate. Version management data had to be synthesized for the remaining tools.

Another observation is that there is a multitude of version management models implemented in commercially available tools. For instance, version management is supported on diagram level in Statemate, but the tool can only maintain one specification configuration.

The existence of multiple version management models in Systems Engineering tools is not surprising. Studies investigating software [37] and mechanical engineering [80] tools indicate a large span of variants. In STEP the version management model defined in the integrated resources and can be considered fixed. Whether the STEP model structure is the most appropriate for Systems Engineering tools is an open question, but the model is rich enough to represent multiple configurations of version managed data elements.

12.4.4 ELEMENT IDENTIFIERS

In many cases the tools use the name of an element as its identifier, where the information model assume the existence of unique object identifiers for unique identification of each element. Consequently it is possible that a legal instantiation of the information model cannot be directly mapped onto the structures of individual tools. For instance, the Teamwork interface did not accept models containing elements with multiple distinct *functional_link* objects with identical name attribute values [44].

Issues relating to element identifiers is a result of the implementation architecture selected for each tool. It is likely that there will be conflicts no matter which identification schema is selected in the information model. However, we consider the approach based on object identifiers more general than any one based on attribute values.

It shall also be noted that issues relating to uniqueness constraints may be resolved in import interfaces via algorithms that test individual identifiers for uniqueness and generates new names in cases where conflicts are detected.

12.4.5 SCHEMA HETEROGENEITY

Schema heterogeneity is inevitable when data from multiple heterogeneous sources shall be integrated. For the tools for which interfaces were developed there are areas where the information model is more extensive than the corresponding representations supported by the individual tools. As a result there where multiple instances where tool interface developers had to implement extensive glue code to comply with the rules defined in the information model.

Function interfaces and io ports

Only the Systembuild/MatrixX tool included explicit support for *io_port* objects for function actual/formal parameter binding. In all other tools used in the validation activity parameter binding is performed via the flow name. In these cases the interfaces had to synthesize *io_port* objects when data is exported. Conversely the import interfaces had to synthesize tool specific representations from those in the information model.

Range of data types supported

Heterogeneity issues were also identified for import interfaces. The developers of the Teamwork import interface, for instance, reported that they had to develop substantial glue code to handle cases where the information model allowed for far more advanced data type representations than the Teamwork tool could handle. For individual tools there were multiple similar situations where the information model was richer than in internal tool schema. Yet, this situation is preferable over a situation where the information model is limiting the set of data that may be exchanged.

Schema heterogeneity — discussion

Heterogeneity between tool internal schemas and the information model do not imply information model deficiencies. Rather it is an expected consequence of the objective to make the information model method and tool independent.

The information model is designed to support constructs of multiple methods using a single unified representation. In doing so the mapping from tool representations to the information model ditto may become complex. The important point is whether there exists equivalent class mapping functions from tool specific representations to the information model. These ensure that no information is modified in the data exchange, c.f., the discussion on mapping functions in Section 2.4.

It was also observed that a large number of objects carrying relatively little information had to be instantiated when tool data was mapped onto the information model structures. The use of entities with few attributes is a conscious decision to allow tools supporting different methods to map information to the information model without any loss of data. The consequence of this design was additional work for tool interface developers and also low performance for interfaces built on-top of relational database management systems, i.e., the Express AP-233 data server.

Tool interface development and information model complexity

One of the issues raised against the information model is that it is too complex to interpret and implement. The implementation activities indicate that developers can realise a fully functional import or export interface to the information model in 1.5 - 5 man-months [74]. This time include tool familiarisation and time required to master the mechanism selected for accessing the tool internal database.

In some cases database access was made through public application programming interfaces, but at least in two cases developers were forced to parse and generate proprietary file formats to access and populate tool databases. Needless to say development of these interfaces required more time than for those utilising public interfaces.

Problems and bugs associated with the STEP development tools also influenced interface development time significantly as reported in [26]. This was particularly true for the version of the EXPRESS-X development tool used for the Statemate and MatrixX interfaces [74].

12.4.6 INTERFACE DEVELOPMENT SUMMARY

Whether it is possible to develop a data exchange interface for a tool is only partly up to the information model. A large proportion of problems reported have been due to:

- Incomplete tool import or export facilities or documentation of such facilities. In some cases tool suppliers have taken the decision not to publish file grammars or programmatic interfaces. This obviously complicates interface development substantially.
- Problems related to the development tools used to develop tool interfaces.

Still there are multiple cases where simple tool specific constructs did require definition of complex mapping functions in order to create the corresponding representation in the information model. Conversely, a large number of queries over specification data in the information model format were required to recreate a simple tool specific construct.

This is a consequence of the variety of tools and methods in use by systems engineers and the objective to keep the information model tool and process independent. Any change in the information model towards the structures used in a specific tool, would most likely complicate interfacing to other tools.

With these observations in mind it must be concluded that tool data exchange interfaces based on the information model can be implemented for a range of Systems Engineering tools.

12.5 Validation Through Data Exchange

The tool interfaces developed were used in mini-projects to validate that the information model could be employed for data exchange in realistic Systems Engineering projects. Two validation scenarios were carried out on industrial material within in the SEDRES-2 project:

- Validation scenario 1 was defined to be similar to a contractor - subcontractor relationship where functional specifications were

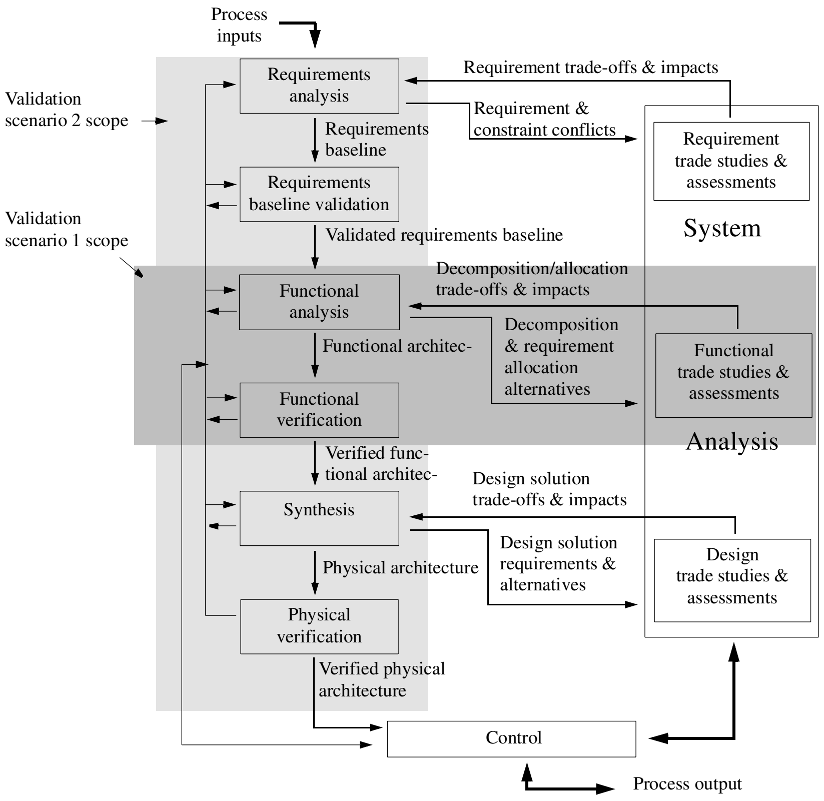

Process inputs
Validation scenario 2 scope
Validation scenario 1 scope
Requirements analysis
Requirements baseline
Requirements baseline validation
Validated requirements baseline
Functional analysis
Functional architec-
Functional verification
Verified func-tional architec-
Synthesis
Physical architecture
Physical verification
Verified physical architecture
Control
Process output
Requirement trade-offs & impacts
Requirement & constraint conflicts
Requirement trade studies & assessments
System
Decomposition/allocation trade-offs & impacts
Decomposition & requirement allocation alternatives
Functional trade studies & assessments
Analysis
Design solution trade-offs & impacts
Design solution requirements & alternatives
Design trade studies & assessments

Figure 12.1: Validation scenario coverage vs IEEE 1220 systems engineering process

exchanged. The scenario was defined to demonstrate the viability of data exchange between different tool sets supporting the same engineering activities.

- Validation scenario 2 was defined to evaluate the suitability of the information model for data exchange between different phases in the Systems Engineering process. The scenario demonstrates the viability of using the information model to integrate data from multiple standalone tools used within the Systems Engineering process.

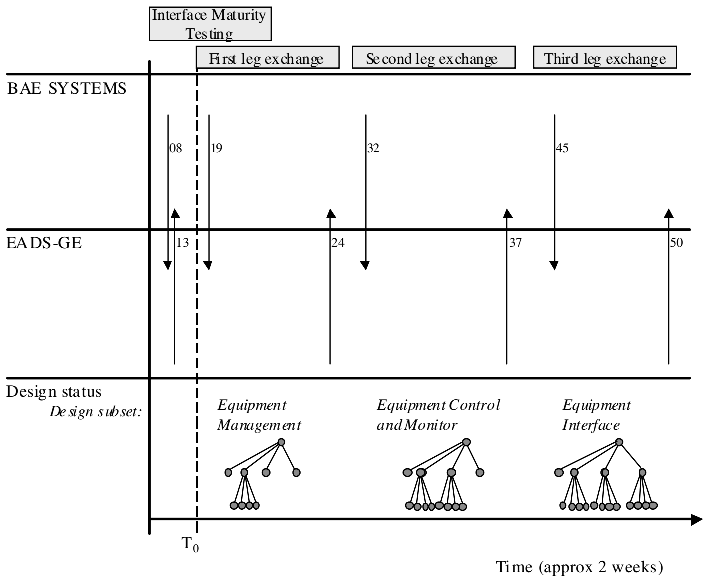


Figure 12.2: Validation scenario 1 data exchanges

An additional validation scenario was executed in conjunction with the SEDRES-2 project at BAE SYSTEMS Australia in cooperation with University of South Australia. In this scenario data was exchanged from Statemate to DOORS in a setting where DOORS where used store, view and analyse project data [136]. DOORS modules were created for requirements, functional and physical architecture elements. Traceability links were captured across the elements. Experience collected in this scenario also indicates the value of using a central repository for project data.

The extent of the two validation scenarios against the Systems Engineering process is illustrated in Figure 12.1. A brief description of each scenario and results are presented below.

12.5.1 VALIDATION SCENARIO 1

Validation scenario 1 was set up to demonstrate data exchange between functional analysis tools. The scenario mimics a multi-partner development effort, incidentally similar to that for the Eurofighter programme,

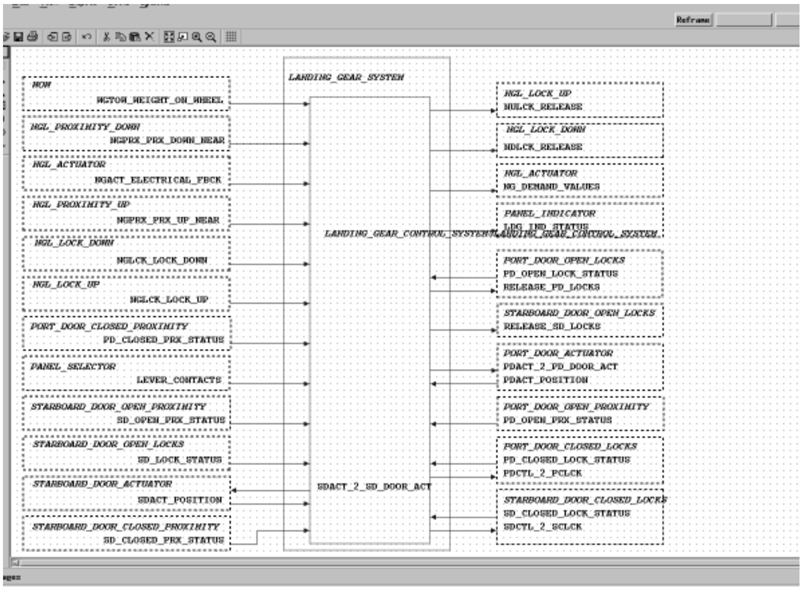


Figure 12.3: Original Statemate model

using the design of an aircraft landing gear system as a demonstrator. The validation scenario was performed by EADS-GE, using the tool Teamwork, and BAE SYSTEMS, using the tool Statemate. The full report on the validation scenario is available in [76]. Exchange of design data was performed in three legs as outline in Figure 12.2.

The design data exchanged contained only element from the functional architecture part of the information model. Moreover data exchanged included only functions and data flows, since the Statemate interface did not implement support for handling finite state machines and Statecharts. An example of the data exchanged from Statemate to Teamwork is presented in Figure 12.3. Two views on the resulting top level representation in the Teamwork tool is presented in Figure 12.4 and Figure 12.5. The view in Figure 12.5 is created based on layout information encoded in the information model format, while a generic placement algorithm has been used to create the view in Figure 12.4. Note that Statemate supports the representation of multiple functional levels within a single graphical view,

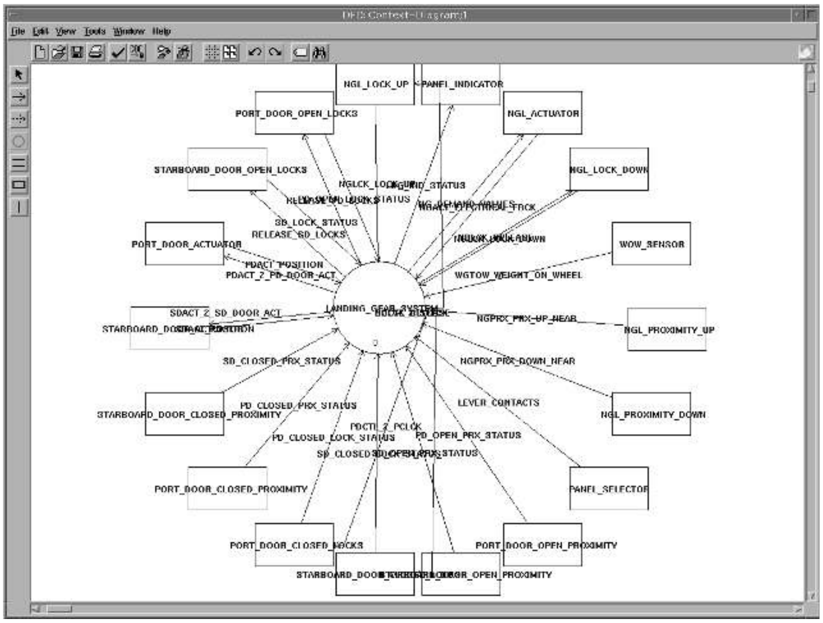


Figure 12.4: Teamwork model generated without layout information from Statemate

while Teamwork can represent a single functional level per graphical view. Consequently the Statemate model illustrated in Figure 12.3 is represented using two views in Teamwork.

The graphical layout of the model presented in Figure 12.4 is generated from the functional architecture elements of the information model only. The appearance of individual functional elements is a result of a generic layout algorithm implemented in the Teamwork interface. Hence there is no direct visual correspondence between the original Statemate model and the resulting one in Teamwork. As a result it is difficult for humans to ascertain that the models are equivalent, even though a closer inspection will reveal that the same model elements are present in both tools.

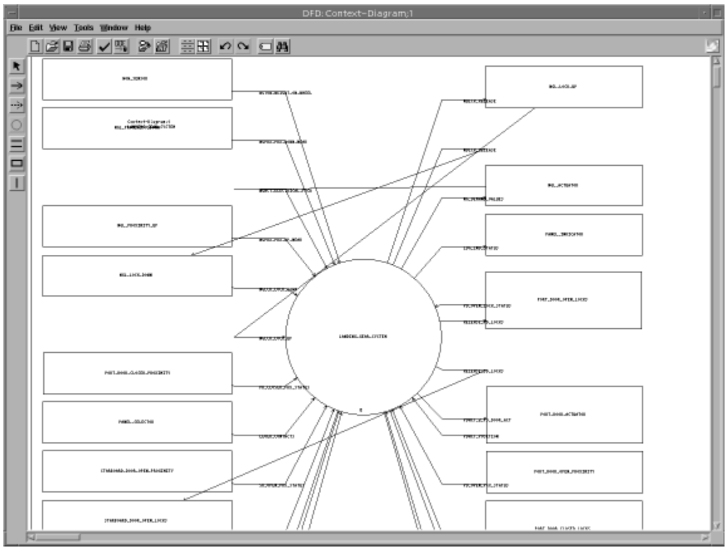


Figure 12.5: Teamwork top-level model with layout information transferred from Statemate

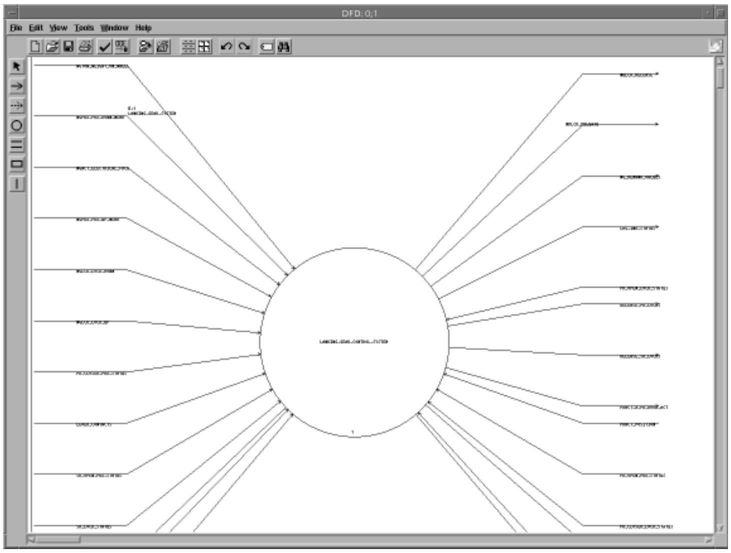


Figure 12.6: Teamwork first level model with layout information transferred from Statemate

In contrast the Teamwork model in Figure 12.5 and Figure 12.6 are much more similar to the original Statemate model and hence easier to interpret. Note that lightning shaped flows in Figure 12.5 are the result of incorrect representations/interpretation of layout information in the tool interfaces that would not be expected in production quality tool interfaces.

The symbols used in Teamwork for representing functions and external agents are fixed. Hence a function presented as a square in Statemate will always be presented as a circle in Teamwork. These transformations are inevitable and do not imply any modifications to the semantics of the models.

Validation scenario results

The validation scenario was very successful in the sense that the information model provide adequate structures for Teamwork and Statemate individually to read and write data to a data exchange format without any loss of data. Still there are substantial risks that information is lost in a data transfer between the tools as:

- Statemate supports a rich set of data types whereas Teamwork only support a basic set. Any use of structured data types in Statemate cannot be represented in Teamwork.
- Teamwork supports representation of functional behaviour using Mealy and Moore type state machines where Statemate supports Statecharts. Any use of Statechart constructs in a data exchange from Statemate to Teamwork will require the Teamwork interface to transform the Statechart to a Mealy or Moore state machine. While this is possible for many cases, the resulting state machine will bear no visual resemblance with the original Statechart. The original Statechart representation cannot be recreated if data is re-imported into Statemate.
- Teamwork supports the use of multiple control bars within a function where Statemate allows only one control activity. It is certainly possible to merge multiple state machines into a single Statechart, but the resemblance with the original specification will be lost. Moreover, a

complex transformation is required when data i re-imported to Teamwork.

The above observations indicate the limitation of data exchange between tools supporting similar, but not equivalent methods. Frequent data exchange between Teamwork and Statemate would require the implementation of restrictions on functionality usage in respective tool. It could be the case that these restrictions would be so constraining that the utility of data exchanges is lost. This result does not disqualify data exchange in all cases. Migration of design models from a basic tool (Teamwork) to an advanced one (Statemate) using the information model would provide substantial benefits.

12.5.2 VALIDATION SCENARIO 2 SETUP AND DESCRIPTION

Validation scenario 2 was setup to demonstrate tool data exchange between different phases in the Systems Engineering process from requirements identification, over functional analysis to identification of high-level system components [18]. The scenario was executed on material developed for the ATV[1] project. The following partners participated in the scenario:

- EADS Launch Vehicles acted as a prime contractor responsible for capturing requirements and performing functional analysis. The tools LabSys and Software through Pictures were used in the process.
- The Technical University of Clausthal acted as subcontractor responsible for identification of system components and allocation of requirements and functionality onto identified components. The tool Demanda 2 was used for this step in the process.
- Alenia acted as a quality assurance entity double-checking the appropriateness of all allocations made. Doors were used to validate allocations.

1. ATV = Automatic Transfer Vehicle, space vehicle under development to deliver freight to the ISS (International Space Station), to boost the orbit of and to move garbage from the ISS.

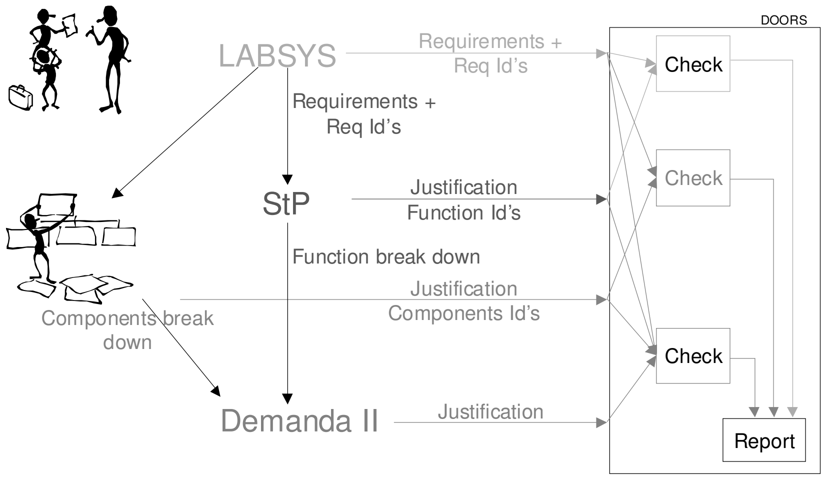


Figure 12.7: Validation scenario 2 setup

The setup of the scenario is illustrated in Figure 12.7. Data exchanges were performed from LabSys to StP, LabSys to Doors, StP to Demanda 2, StP to Doors and from Demanda 2 to Doors. The tools were selected such that the sink tool in all exchanges could handle all significant data generated by the source tool.

- LabSys allows for the capture of textual requirements and the identification of required system functionality in different system life-cycles, and interaction between the system under specification and external systems. Functions and interactions are captured in a textual view.
- The functionality captured in LabSys is interpreted in StP and used to form the basis for a system functional architecture per life-cycle. The imported functional architecture was further extended in StP.
- Identified requirements and functionality was imported into Demanda 2 and served as a baseline for the identification of system components and the allocation of functionality and requirements onto the components.
- Design data from each tool was imported into Doors for verification of the allocations made.

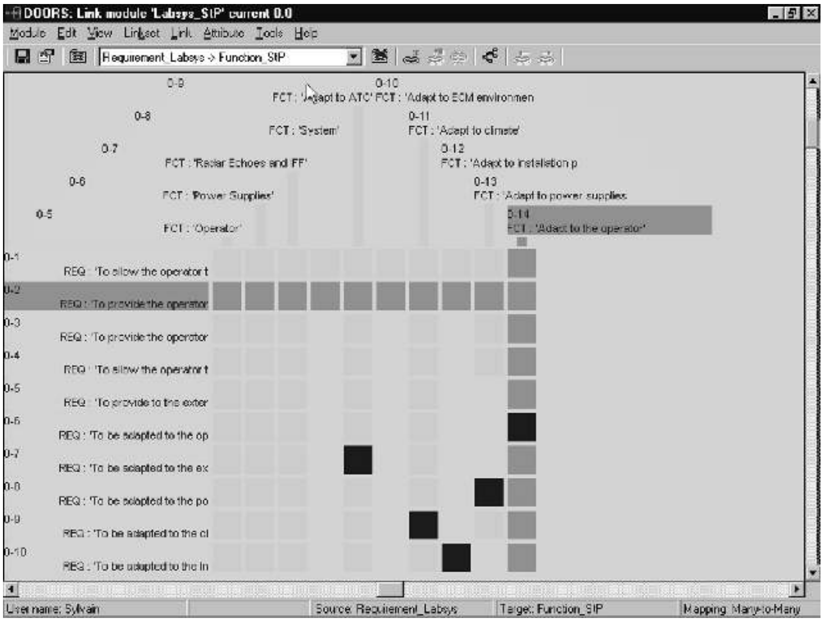

DOORS: Link module 'Labsys_StP' current 0.0
Module Edit View Linkset Link Attribute Tools Help
Requirement_Labsys -> Function_StP
0-5 FCT : 'Operator'
0-6 FCT : 'Power Supplies'
0-7 FCT : 'Radar Echoes and FF'
0-8 FCT : 'System'
0-9 FCT : 'Adapt to ATC'
0-10 FCT : 'Adapt to ECM environmen
0-11 FCT : 'Adapt to climate'
0-12 FCT : 'Adapt to installation p
0-13 FCT : 'Adapt to power supplies
0-14 FCT : 'Adapt to the operator'
0-1 REQ : 'To allow the operator t
0-2 REQ : 'To provide the operator
0-3 REQ : 'To provide the operator
0-4 REQ : 'To allow the operator t
0-5 REQ : 'To provide to the exter
0-6 REQ : 'To be adapted to the op
0-7 REQ : 'To be adapted to the ex
0-8 REQ : 'To be adapted to the po
0-9 REQ : 'To be adapted to the cl
0-10 REQ : 'To be adapted to the in
User name: Sylvain
Source: Requirement_Labsys
Target: Function_StP
Mapping: Many-to-Many

Figure 12.8: Doors view on requirement to function traceability

Validation scenario 2 prime focus was on the information models support for representing traceability links between requirements and functions, between functions and physical components, and between requirements and physical components. A view from Doors generated in validation scenario 2 illustrating traceability links generated from LabSys and StP is presented in Figure 12.8.

Validation scenario results

The usage of the information model in validation scenario 2 validates the information model support for traceability links between requirements, functional and physical architecture. A large portion of the success lie in the selection and use of tools for the scenario. The overlap in tool functionality allowed for data exchanges with no loss of information.

12.5.3 VALIDATION SCENARIO RESULTS

Both validation scenarios indicate that data exchange based on a tool neutral information model is a viable approach for industrial applications both between tools of similar types and between tools used in different phases in the engineering process. It is noted in [26] that "after interface maturity actions are discarded, realistic design data sets can be transferred between tools in minutes". This is certainly an important validation result. More specific findings captured include:

- The practical applicability of the information model for data representation and tool interface development was validated.
- Data exchange highlights the challenge of configuration control of data distributed over multiple tools
- There were cases where tool interfaces implementing the semantic integrity constraints in the data model did capture design inconsistencies not captured in tool internal consistency checks.

12.6 Validation Result Summary

The validation scenarios performed has validated that the information model is applicable for the representation and exchange of:

- Textual requirements, including defined and arbitrary requirement properties such as priority and requirement ownership.
- Functional models, including functional hierarchy, functional interaction models, and models containing explicit behaviour components in the form of finite state machines.
- Physical architecture model in terms of identification of logical system components.
- Requirement and functional allocation information capturing how requirements are allocated to functions and physical components and the allocation of functions to physical components.
- The appropriateness of the part of the model capturing specification layout was validated. Specifications exchanged with layout informa-

tion included are much easier for humans to interpret as compared with specifications exchanged without layout information.
- The utility of the system architecture model has been clearly demonstrated as it allow for representation of multiple system specification configurations. However, there have only been limited experiments involving multiple specification element versions and multiple configurations.

Despite the fact that substantial parts of the model have been validated there remain areas which have not been validated through interface development and data exchanges. The main areas are:

- Representations of functional behaviour using causal models such as Functional Flow Block Diagrams (FFBD) and Extended Functional Flow Block Diagrams (EFFBD). These functional models are widely in use and there is no doubt that a demonstration indicating the support for these models would increase peer confidence in the information model.
- Capture of the engineering development process in terms of activities performed, relating specification data to activities performed and change management.
- Capture of verification and validation requirements on a system.

The validation results are assessed positively despite the fact that the validation activities did not cover all parts of the information model. Moreover, the fact that the tool set developed for Validation scenario 2 has been put in operational use within EADS Launch Vehicles is a clear indication of the applicability of the information model for real tool data exchanges [20].

12.7 Validation Results vs. Information Modelling Requirements

In this section the results from the validation activities are related to the information modelling requirements as presented in Section 4.3. The evaluation is based on observations made throughout the SEDRES-2 project and captured in [74] [75] [26].

12.7.1 PROCESS AND METHOD INDEPENDENCE

The information model has been validated through real data exchanges between a number of tools and also been subject to dry mappings analyses from tool concepts to information model concepts. The results indicate that the information model is suitable for data exchange of data generated from a number tools supporting different methods appropriate for Systems Engineering.

12.7.2 COMPLETENESS ASSUMPTION

Overall this guideline proved to be appropriate. In all cases interface developers were able to map tool data to the information model, but in some cases some extensive transformation were required.

12.7.3 CONTEXT INDEPENDENCE

The approach to define entities such that they could be used independent of a specific system context has not been validated during the validation scenarios. The reason for this is the limited configuration management support in the tools used to validate the information model. None of the tools used had the facilities to support specification element reuse across multiple system specifications or multiple versions thereof. Moreover, at least one tool interface developers did questioned whether the additional effort required to implement context independence match the value provided by the concept [75]. Especially given the fact that there appears to be no tool support for the capabilities implemented in the information model.

12.7.4 INFORMATION MODEL DETAIL

The validation scenarios validated that the information model was suitable for the exchange of data between basic tools, between advanced tools and between a basic and an advanced tool. Moreover, the value of the support for multiple semantic variants of common modelling concepts has been validated.

12.7.5 PRESERVATION OF SPECIFICATION STRUCTURE

The tools used, and hence the underlying methods supported, in the validation scenarios were similar that the need to transform a specification element from an original structure to an equivalent one did never arise. However, in previous validation activities during the SEDRES project there were a case where process activation tables (PAT) was represented as an equivalent Moore finite state machines in the information model, and hence in the receiving tool. In this case substantial analysis efforts were required before it could be concluded that the process activation table and finite state machine representations were indeed equivalent.

12.7.6 PRESERVATION OF SPECIFICATION LAYOUT

The capability to represent the approximate layout of specification elements exchanged has been much appreciated. Readability of transferred specifications where layout information is included was deemed to be significantly higher as compared to the same information exchanged without any layout information included in the data exchanged. This is also clearly illustrated in Figure 12.4 and Figure 12.5.

12.8 Discussion

The results gathered from validation activities performed were generally positive.

- Data exchange capabilities have successfully been implemented and validated for a number of commercial and in-house tools. The completeness of the information model was validated through validation

scenarios involving multiple partners using multiple tools.
- The requirements guiding the development of the information proved to be sound in the sense that tool interfaces to complex commercial tools could be developed within relative short time periods.

In this sense the results from using the information model is a complete success. However, there are no results suggesting that the information model as implemented is the best possible vehicle for Systems Engineering data exchange.

A number of issues relevant for future development of tool neutral Systems Engineering data exchange information models have been highlighted. The main issues are over the complexity of the information model and the associated cost for tool interface development. This indicate that a number of conscious trade-offs have to be made when designing a data exchange information model.

1. Trade off between information model configuration management and specification element reuse capabilities vs. tool interface development cost. The more capabilities included in the information model, the more entities will be required to be handled in tool interface, which will result in higher tool interface development costs.
2. Trade off between information model detail and support for multiple Systems Engineering methods vs. tool interface development cost. The more variants to a concept supported, the more complex the information model, which in turn will results in higher tool interface development costs.

The issue of tool interface development cost must also be weighted against the utility provided by the tool neutral information model. Reducing model capabilities may make it less applicable in an industrial context and less appealing to adopt for industry as well as for tool vendors. Especially as long as there is a diversity of tools and methods in use for system specification in the Systems Engineering community.

Likewise, if support for multiple semantics variants to a basic concept is dropped then the information model will not be able to convey the fine facets of a system specification. This would increase the risk for non-detected modifications to specification data exchanged. One approach to simplifying the information model would be to drop the requirement for tool and method neutrality altogether and encode support for a specific Systems Engineering method only. This would be similar to the development of a "UML" for Systems Engineering as done in the SysML project [146]. While this could be an interesting approach from a usability point of view it is a breech with the approach selected for the work presented in this thesis and in conflict with the authors view on the purpose of a data exchange standard.

12.9 Summary

In this chapter we have reviewed activities undertaken by industry partners to validate the suitability of the information model for Systems Engineering tool data exchange.

Validation activities performed has clearly indicated that Systems Engineering tool data exchange implemented using a tool independent information model has the potential to save a lot of time and effort in Systems Engineering projects. High quality tool interfaces to the information model can be developed at was perceived to be reasonable cost. However, issues have been raised whether the information model could be simplified to facilitate tool interface development.

Chapter 13
Conclusions and Future Work

In this chapter a summary of the contents of this thesis is presented with emphasis on its contributions. Finally we discuss some areas for potential future work.

13.1 Thesis Summary

The objective of the work reported in this thesis has been to research methods for enabling data exchange between computer-based tools used for capturing Systems Engineering data. The work reported in the thesis can be divided into three overarching sections as follows:

13.1.1 INTRODUCTION

Parts I and II of the thesis present the domain of systems engineering and the restrictions made for the thesis. Product data modelling frameworks, especially STEP, and problems related to data exchange is introduced. Finally the approach and guiding non-functional requirements for implementing the information model is presented.

13.1.2 INFORMATION MODEL OVERVIEW

Part III of the thesis presents and motivates the major constructs in the information model. The presentation is made for each significant component of the information model coupled with example instantiations to illustrate how the information model constructs shall be used to represent tool data.

13.1.3 EVALUATION

Part IV of the thesis presents activities performed to evaluate the suitability of the information model for data representation and data exchange. Evaluation activities undertaken include:

- Information model review by Systems Engineering specialists from INCOSE and the AP-233 working group within ISO 10303.
- Implementations to validate the applicability of the information model for Systems Engineering tool data exchange interface development.
- Validation scenarios to validate the suitability of the information model for data exchange of realistic Systems Engineering data.

13.2 Conclusions

The reviews of the information model, the results tool interfaces developed and the tool data exchanges carried out as presented in this thesis clearly illustrate the benefits of using the information model for data exchange. Separate scenarios have demonstrated the ability of the information model to support:

- Data exchange across tools used in the same phase in the Systems Engineering process
- Data exchange between tools and a central repository to support traceability throughout the engineering process

The validation activities have been carried out in an industrial context, using industrial material so a fair amount of confidence can be placed in the results obtained. The number of tools used in the validation scenarios also indicates that the model can be used by a range of tools. The objective to make the information model method and tool independent is considered validated.

With these results in mind it is fair to say that the information model has been successful, considering the constraints documented and the model guidelines applied to the design of the model. However, many comments from the Systems Engineering community on the work presented herein relate to these imposed constraints. The objective to build a method and tool independent information model result in a model which is complex to interface with for some tools. Some stakeholders have found the lack of explicit support for popular methods most distracting. Likewise there has been a lot of debate over the scope of the information model. Again this is not surprising considering the large number of people with different backgrounds that has participated in formal reviews of the information model.

These observations highlight the dilemma of a data exchange information model. Ideally the information model shall be trivial in structure, provide a perfect interface to all relevant methods and be extensive in its method support. Unfortunately, in a field like Systems Engineering with embedded complexity and where multiple methods are in use, it is not possible to comply with all these points. Support for multiple methods will inevitable increase model complexity.

13.3 Future Work

There are multiple paths to extend on the work presented herein. Potential extensions are presented below.

13.3.1 EXTENDING THE INFORMATION MODEL

The STEP framework offer product models (and modules) for mechanical and electrical design as well as maintenance and logistics. There are obvious advantages of combining these models to create a single integrated model supporting representation of product data supporting larger product data sets of the system life-cycle. Much of this work is already underway within the AP-233 working group through the creation of STEP modules supporting Systems Engineering. These modules can quite easily be combined with modules created for other engineering domains. The primary activity is to define modules that allow for capturing of traceability links between the domains, i.e., traceability between Systems Engineering data and, e.g., mechanical engineering data.

13.3.2 DEFINITION OF FORMAL SEMANTICS

The semantics of each entity in the information model is defined textually. This is sufficient for the representation of requirement and the system physical architecture, but the definition of a formal semantics for the functional architecture part of the information model would improve the integrity of the model substantially. The definition of a formal semantics for the information model must encompass multiple combinations of information model entities in order to cover all alternatives that can be expressed in the information model. This work may build on work performed in the SAFEAIR project [53].

13.3.3 TOOL IMPLEMENTATION

The implementation of a user friendly Systems Engineering tool based on the information model would provide a computer based specification tool for evaluation of the benefits of the context independence principle information modelling requirement presented in Section 4.3. A tool could be implemented with relative ease based on a database generated directly from the information model.

13.3.4 STANDARDISATION

One obvious path is to continue work towards information model standardisation. To prepare the information model for standardisation it must be partitioned to suit the STEP modular architecture. Technically speaking this is primarily a matter of transforming the structures in the information model to fit the data structures in the STEP PDM modules, but it also includes activities to collect information from the stakeholders active within STEP.

13.4 Concluding Remarks

The information model presented herein was developed with the intention that it should eventually become a Systems Engineering data exchange standard. Judged by the interest the work has generated at ISO 10303 and INCOSE (the International Council for Systems Engineering) there is no doubts that there exists a real industrial need for data exchange capabilities between Systems Engineering tools. However, the problems associated with striking an agreement on the scope and objectives with the information model is a clear indication that additional work is required to define the extent of Systems Engineering representations suitable for standardisation and also the level of support the information model shall provide. It is the hope of the author that the ideas, objectives and results presented in this thesis will provide some guidance to future projects for facilitating Systems Engineering tool data exchange.

References

[1] Abramovici, M., Gerhard, D., and Langenberg, L. Application of PDM Technology for Product Life Cycle Management. In *Proceedings 4th International Seminar on Life Cycle Engineering*, pages 15 – 29, 1997.

[2] Alford, M. Software Requirements Engineering Methodology (SREM) at the Age of Two. In *Proceedings The IEEE Computer Society's Second International Computer Software and Applications Conference*, pages 332 – 339. IEEE Computer Society, 1978.

[3] Alford, M. Attacking Requirements Complexity Using a Separation of Concerns. In *1st IEEE Conference on Requirements Engineering*, pages 2 –5. IEEE Computer Society, 1994.

[4] Alford, M. W. A Requirements Engineering Methodology for Real-Time Processing Requirements. *IEEE Transactions on Software Engineering*, 3(1):60–69, January 1977.

[5] Anon. Integration Definition for Function Modelling (IDEF0). Draft Federal Information Processing Standards Publication 183, December 1993.

[6] Anon. *Industrial Automation Systems and Integration - Product Data Representation and Exchange. Part 1: Overview and Fundamental Principles.* ISO 10303, December 1994.

[7] Anon. *Industrial Automation Systems and Integration - Product*

Data Representation and Exchange - Part 203: Application Protocol: Configuration Controlled Design. ISO 10303, 1 st. edition, December 1994.

[8] Anon. *Industrial Automation Systems and Integration - Product Data Representation and Exchange - Part 11: Description Methods: The EXPRESS Language Reference Manual*. ISO 10303, 1 st. edition, December 1994.

[9] Anon. *ISO 10303-41 Product Data Representation and Exchange - Part 41: Integrated Generic Structures: Fundamentals of Product Description and Support*. ISO, December 1994.

[10] Anon. *ISO 10303-41 Product Data Representation and Exchange - Part 43: Integrated Generic Structures: Representation Structures*. ISO, December 1994.

[11] Anon. *ISO 10303-41 Product Data Representation and Exchange - Part 44: Integrated Generic Resources: Product Structure Configuration*. ISO, 1994.

[12] Anon. Guidelines for Application Interpreted Construct Development. Internet, June 2000. http://www.ukceb.org/step/pages/application_interpreted_construct.htm.

[13] Anon. *Meta Object Facility Specification version 1.3*. OMG, 2000.

[14] Anon. *ISO/IEC CD 15288 Systems Engineering - System Lifecycle Processes*. ISO, 2001.

[15] Bahill, T. and Daniels, J. Using Object-Oriented and UML Tools for Hardware Design: A Case Study. *Systems Engineering*, 6(1):28 – 48, 2003.

[16] Bahill, T. and Dean, F. Discovering System Requirements. In Sage, A. and Rouse, W., editors, *Handbook of Systems Engineering and Management*, chapter 4, pages 175–220. John Wiley and Sons, 1999.

[17] Barbeau, S. *Methodolgie d'ingenierie des Systemes Aerospatiale*. PhD thesis, Ecole Doctorale Sciences Pour l'ingenieur de Nantes, November 1997. In French.

[18] Barbeau, S. VS 2 Performance Report. SEDRES Technical Report WP/4-03/AM/03-1, EADS Launch Vehicles, October 2001.

[19] Barbeau, S., Carlsson, E., and Törne, A. Data Exchange Requirements. Project report D.4.4.1, SEDRES, November 1996.

[20] Barbeau, S. and Loeuillet, J.-L. Utilisation des Techniques STEP pour le Transfert d'Information Entre Outils de Conception Fonctionnelle et Meteurs de Simulation - Application au Véhicule Spatial Automatique de Transport de Fret (ATV). In *Proceedings MICAD*, 2001. In french.

[21] Batini, C., Lenzerini, M., and Navathe, S. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18(4):323 – 364, December 1986.

[22] Beresi, L. and Pezzi, M. Toward Formalizing Structured Analysis. *ACM Transactions of Software Engineering and Methodology*, 7(1):80–107, January 1998.

[23] Bilgic, T. and Rock, D. Product Data Management Systems: State of the Art and the Future. In *Proceedings of DECT'97, 1997 ASME Design Engineering Technical Conferences*. ASME, September 1997.

[24] Blanchard, B. and Fabrycky, W. *Systems Engineering and Analysis*. Prentice Hall International Series in Industrial ans Systems Engineering. Prentice Hall, 3 edition, 1998.

[25] Buede, D. Functional analysis. In Sage, A. and Rouse, W., editors, *Handbook of Systems Engineering and Management*, chapter 25, pages 997 – 1035. John Wiley and sons, 1999.

[26] Candy, L. and Britton, J. Sedres2 Evaluation Report. SEDRES2 Deliverable SEDRES-2 DE/1-03/LU/D12, Loughborough University, 2001.

[27] Carlsson, E. *The Cohsy Project - Complex Heterogeneous Systems*, chapter 12 Tool Integration Methods and Techniques - an Overview. Number LiTH-ISY-R-1920. Linköpings Universitet, 1996.

[28] Cattell, R. *Object Data Management*. 1994.

[29] CDIF. *CDIF CASE Data Interchange Format - Overview*. Number EIA/IS-106. Electronic Industries Association, 1994.

[30] CDIF. *CDIF Integrated Meta-model Data Flow Model Subject Area*. Number EIA/IS-115. Electronic Industries Association, 1995.

[31] Chen, P. The Entity-Relationship Model: Towards a Unified View of Data. *ACM Transactions on database systems*, 1(1):9–36, 1976.

[32] Chestnut, H. *Systems Engineering Methods*. Systems Engineering and Analysis. John Wiley & Sons, 1 edition, 1967.

[33] Christensen, M. Report on SEDRES presentation. AP-233 workspace memo, July 2001. Published in the AP-233 forum at http://bscw.gmd.de.

[34] Cocks, D. The Suitability of Using Object Modeling at the System Level. In Fairbairn, A., Jones, C., Kowalski, C., and Robson, P., editors, *Proceedings of the 9th International Symposium of the International Council on systems engineering*, volume 2, pages 1047 – 1054. INCOSE, INCOSE, 1999.

[35] INCOSE Modelling and Tools Technical Committe. Systems Engineering Tools Survey. http://www.incose.org/tools/, July 2003.

[36] Compatangelo, E. Towards an Open Framework for Conceptual Knowledge in ECBS Domain and Information Modelling. In *Proceedings 1997 IEEE Conference and Workshop on Engineering of Computer Based Systems*, pages 211– 218. IEEE Press, 1997.

[37] Conradi, R. and Westfechtel, B. Version Models for Software Configuration Management. Technical Report AIB 96-10, RWTH Aachen, October 1996.

[38] Curwen, P. System Development Using the CORE Method. BAE SYSTEM technical report, July 1993.

[39] Dai, H. An Object-Oriented Approach to Schema Integration and Data Mining in Multiple Databases. In Chen, Li, Mingins, and Meyer, editors, *Technology of Object-Oriented Languages*, pages 294 – 303. IEEE Computer Society, September 1997.

[40] Danner, W. and Kemmerer, S. *STEP the Grand Experience*,

chapter 3. Number Special publication 939. NIST, 1999.

[41] Dick, J. and Jackson, K. Requirements Management for Product Families: a Three Dimensional Approach. In *GEIA 34th Annual Engineering & Technical Management Conference*, pages 21 – 26, 2000.

[42] do Prado Leite, J. C. S. and Freeman, P. Requirements Validation Through Viewpoint Resolution. *IEEE transactions on software engineering*, 17(12):1253 – 1269, December 1991.

[43] Dunford, J. and Shaw, N. PLCS Statement of Technical Requirements. Requirement specification, PLCS, 1999.

[44] Eckert, R. and Johansson, G. Experiences from the use and Development of ISO 10303-ap233 Interfaces in the Systems Engineering Domain. In *Proceedings 9th international conference of Concurrent Enterprising*, June 2003.

[45] Engwall, M. *Jakten på det effektiva projectet*. Nerenius & Santerus Förlag, 2 edition, 1999.

[46] Fang, D., Hammer, J., and McLeod, D. The Identification and Resolution of Semantic Heterogeneity in Multidatabase Systems. In *Proceedings of the first International workshop on Interoperability in Multidatabase systems*, pages 136 – 143. IEEE Computer Society, 1991.

[47] Feeney, A. B. *The building blocks of STEP*, chapter 4, pages 47–59. Special Publication 939. NIST, 1 edition, July 1999.

[48] Goh, C. H. *Representing and Reasoning about Semantic Conflicts in Heterogeneous Information Systems*. PhD thesis, Sloan School of Management, Massachusetts Institute of Technology, 1997.

[49] Goh, C. H., Madnick, S., and Siegel, M. Ontologies, Contexts, and Mediation: Representing and Reasoning about Semantic Conflicts in Heterogeneous and Autonomous Systems. Working paper 3848, Sloan School of management, Massachusetts Institute of Technology, 1995.

[50] Goh, C. H., Madnick, S. E., and Siegel, M. D. Context Interchange: Overcoming the Challenges of Large-Scale Interoperable Database Systems in a Dynamic Environment. In *Proceedings of*

the Third International Conference on Information and Knowledge Management, pages 337 – 346. ACM, November 1994.

[51] Goldstein, B. In the Beginning There was Product Data Exchange. In Kemmerer, S. J., editor, *STEP - the Grand Experience*, number SP 939, chapter 2, pages 7 – 21. NIST, 1999.

[52] Gonzales, R. Completeness of Conceptual Models Developed Using the Integrated System Conceptual Model (ISCM) Development process. In *10th Annual International Symposium of the International Council on Systems Engineering*. INCOSE, INCOSE, 2001.

[53] Goshen-Meskin, D., Gafni, V., and Winokur, M. Safeair - an Integrated Development Environment and Methodology. In *Proceedings INCOSE 2001 Symposium*. INCOSE, July 2001.

[54] Gould, R. *Graph Theory*. Benjamin/Cummings, 1988.

[55] Grady, J. O. *System Requirements Analysis*. CRS Press, 1993.

[56] Gudgeirsson, G. RQML. Masters thesis, Department of Computer Science, University of York, 2000.

[57] Hall, A. *A Methodology for Systems Engineering*. Van Nostrand, 1 edition, 1962.

[58] Harel, D. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[59] Harel, D. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[60] Harel, D. and Naamad, A. The Statemate Semantics of Statecharts. *Transactions on Software Engineering and Mathodology*, 5(4):293–333, October 1996.

[61] Harel, D. and Politi, M. *Modeling Reactive Systems with Statecharts: the Statemate Approach*. Number D1100-43-A, 6/97. I-logix, June 1997.

[62] Hatley, D. and Pirbhai, I. *Strategies for Real-Time System Specification*. Dorset House, 1987.

[63] Hatley, D. J. Current System Development Practices Using the

Hatley/Pirbhai Methods. *The Journal of NCOSE*, 1(1):69–81, 1994.

[64] Holagent. Holagent home page. http://www.holagent.com, 2004.

[65] Hurson, A. R., Bright, M. W., and Pakzad, S. H., editors. *Multidatabase Systems: An Advanced Solution for Global Information Sharing*. IEEE Computer Society Press, 1993.

[66] I-logix. I-logix home page. http://www.ilogix.com.

[67] IEEE. *IEEE Trial-Use Standard for Application and Management of the System Engineering Process, IEEE STD 1220-1994*. IEEE Press, 1994.

[68] IEEE. *IEEE Std 1233-1996, IEEE Guide for Developing System Requirements Specifications*. IEEE, 1996.

[69] IGES. *The Initial Graphics Exchange Specification*. IGES/PDES, Nationa Institute of Standards and Technology, Gaithersburg, MD 20899, USA, 1991.

[70] Ishikawa, Y. and Vaughan, C. SC4 Industrial Framework. Technical Report ISO TC184/SC4/WG10 N300, ISO TC184/SC4/WG10, 2000.

[71] Jackson, K. An Information Model for Computer Based Systems. In Lawson, H. W., editor, *1994 Tutorial and Workshop on Systems Engineering of Computer-Based Systems*, pages 32–43. IEEE Computer Society Press, 1994.

[72] Johannesson, P. Schema Transformation as an Aid in View Integration. In *Proceedings of the fifth internation conference on Advanced Information systems engineering*, number 685 in Lecture notes in computer science, pages 71 – 92. Springer verlag, 1993.

[73] Johansson, O. Using a Meta-Database for Integration Planning and Documentation of product modelling systems. In *Proceedings of Produktmodeller 98*. LiU-IKP, 1998.

[74] Johnson, J. Sedres-2 Prototypes, Interfaces and Environments. Project Deliverable SEDRES-2 DE/3-3/BA/01-1, BAE SYSTEMS, November 2001.

[75] Johnson, J. Sedres-2 Validation Report. SEDRES2 Deliverable SEDRES2 DE/4-4/BA/01-1, BAE SYSTEMS, 2001.

[76] Johnson, J. and Giblin, M. VS1 Performance Report. SEDRES2 technical report WP/4-2/BA/01-1, BAE SYSTEMS, September 2001.

[77] Josifovski, V. *Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration*. PhD thesis, Department of Computer and Information Science Linköpings Universitet, 1999.

[78] Kahn, H. Information Modelling - a Basis for Building Standards. *EDIF Newsletter June 1995*, page 2, June 1995.

[79] Kangassalo, H. *Structuring Principles of Conceptual Schemas and Conceptual models*, chapter 4, pages 223–308. Studentliteratur, 1983.

[80] Katz, R. Towards a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys*, 22(4):375–408, 1990.

[81] Kemmerer, S. STEP - the Grand Experience. Special publication 939, NIST, 1999.

[82] Kim, W. and Seo, J. Classifying Schematic and Data Heterogeneity in Multidatabase Systems. *IEEE Computer*, December 1991.

[83] Kotonya, G. Practical Experience with Viewpoint-Oriented Requirements Specification. *Requirements Engineering*, 4:115–133, 1999.

[84] Kotonya, G. and Sommerville, I. *Requirements Engineering: Processes and Techniques*. Worldwise Series in Computer Science. John Wiley and sons, 1998.

[85] Lake, J. Unraveling the Systems Engineering Lexicon. In *Proceeding INCOSE conference*, 1996.

[86] Larson, J. A. and Sheth, A. P. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[87] Larson, J. A., Navathe, S. B., and Elmasri, R. A Theory of Attribute Equivalence in Databases with Application to Schema Integration. *IEEE Transactions on Software Engineering*, 15(4):449 – 463,

1989.

[88] Lee, P. T. and Tan, K. P. Modelling of Visualised Data-Flow Diagrams Using Petri-net Model. *Software Engineering Journal*, 7(1):4 – 12, January 1992.

[89] Lerat, J.-P. What is Needed to Engineer a System. Paper booklet, ViTech Europe, 41 rue d'Arsonval, 44 000 - Nantes, France, 1999.

[90] Lewis, A. System Architectures. In Sage, A. and Rouse, W., editors, *Handbook of systems engineering and management*, chapter 12, pages 427 – 453. John Wiley and sons, 1999.

[91] Lewis, A. and Wagenhals, L. C4ISR Architectures: 1. Developing a Process for C4ISR Architecture Design. *Systems Engineering the Journal of the International Council on Systems Engineering*, 3(4):225 – 247, 2000.

[92] Litwin, W. and Abdellatif, A. Multidatabase Interoperability. *IEEE Computer*, 19(12):10 – 18, December 1986.

[93] Loborg, P., Risch, T., Sköld, M., and Törne, A. Active Object Oriented Databases in Control Applications. *Microprocessing and Microprogramming*, (38):255 – 263, 1993.

[94] Loffredo, D., editor. *ISO/DIS 10303-24 Product data representation and Exchange: Implementation Methods: C Language Binding of Standard Data Access Interface*. ISO, 1999.

[95] Long, J. Relationships Between Common Graphical Representations used in Systems Engineering. In *Proceedings of the 5th Annual International Symposium of the International Council on Systems Engineering*, 1995.

[96] Long, J. Relationships Between Common Graphical Representations used in Systems Engineering. Whitepaper, Vitech Corporation, 2002.

[97] Lykins, H., Friedenthal, S., and Meilich, A. Adapting UML for an Object Oriented Systems Engineering Method (OOSEM. In *Proceedings of the 10th International Symposium of the International Council on Systems Engineering*, pages 519 – 526. INCOSE, INCOSE, 2000.

[98] Malmqvist, J. Implementing Requirements Management: A Task for Specialized Software Tools or PDM Systems. *Systems Engineering: The journal of the international council on Systems Engineering*, 4(1):49 – 57, 2001.

[99] Mannion, M., Keepence, B., Kaindl, H., and Wheadon, J. Reusing Single System Requirements from Application Family Requirements. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 453–462. IEEE Press, 1999.

[100] Mar, B. Systems Engineering Basics. *The Journal of NCOSE*, 1(1):7–15, 1994.

[101] Martin, J. *Systems Engineering Guidebook*. CRC Press, 1 edition, 1997.

[102] Martin, J., editor. *EIA-632 Processes for Engineering a System*. ANSI/EIA-632-1998. EIA, January 1999.

[103] Martin, J. Telecom and Defense Systems: Exploring Differences in the Practice of Systems Engineering. In Fairbairn, A., Jones, C., Kowalski, C., and Robson, P., editors, *Proceedings of the 9th annual international symposium o fthe International Council of Systems Engineering*, volume 2, pages 1191–1198. INCOSE, June 1999.

[104] McCabe, L., editor. *ISO 10303-23 Product Data Representation and Exchange: C++ Language Binding to the Standard Data Access Interface*. ISO, 1999.

[105] Mohrmann, J., editor. *Product Data Representation and Exchange; Application Protocol: Core Data for Automotive Mechanical Design Process*. ISO, 1997.

[106] Murata, T. Petri nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[107] Nettles, D. Industrial Automation Systems and Integration - Product Data Representation and Exchange - part 1017: Application Module: Product Identification. http://wg10step.aticorp.org/moduleweb/navigation/default.htm, August 2001.

[108] Nissanke, N. *Realtime systems*. Prentice Hall, 1 st edition, 1997.

[109] Nordstrom, G., Sztepanovits, J., Karsai, G., and Ledeczi, A. Me-

tamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments. In *Proceedings 1999 IEEE Conference and Workshop on Engineering of Computer-Based Systems*, pages 68–74. IEEE Press, 1999.

[110] Nuseibeh, B., Kramer, J., and Finkelstein, A. A Framework for Expressing the Relationship Between Multiple Views in Requirement Specification. *IEEE Transactions on software Engineering*, 20(10):760 – 773, October 1994.

[111] Ögren, I. Possible Tailoring of the UML for Systems Engineering Purposes. *Systems Engineering*, 3(4):212 – 224, 2000.

[112] Oliver, D. A Draft Integration of Information Models: Complement model and Oliver model. In Lawson, H. W., editor, *Tutorial and Workshop on Systems Engineering of Computer-Based Systems*, pages 44–69. IEEE Computer Society Press, 1994.

[113] Oliver, D. Re: Lean and fat. E-mail to the author debating the structure of the functional hierarchy model, May 2001.

[114] Oliver, D., Kelliher, T., and Keegan, J. *Engineering Complex Systems with Models and Objects*. McGraw-Hill, 1 st. edition, 1997.

[115] Orsborn, K. *On Extensible and Object-relational Database Technology for Finite Element Analysis Applications*. PhD thesis, Department of Computer and Information Science, Linköpings Universitet, 1996.

[116] Owen, J. *STEP - An Introduction*. Information Geometers, 1 edition, 1993.

[117] Pandikow, A. *A Generic Principle for Enabling Interoperability of Structured and Object-oriented Analysis and Design Tools*. Ph.d. dissertation, Linköpings Tekniska Högskola, 2002.

[118] Parnas, D. L. Using Mathematical Models in the Inspection of Critical Software. In Hinchey, M. and Bowen, J., editors, *Application of Formal Methods*, chapter 2, pages 17 – 31. Prentice Hall, 1995.

[119] Patterson, F. G. Systems Engineering Life cycles: Life cycles for Research, Development, Test and Evaluation; Acquisition; and Planning and Marketing. In Sage, P. and Rouse, W. B., editors, *Handbook of Systems Engineering and Management*, chapter 2, pa-

ges 59 – 111. John Wiley and sons, 1999.

[120] Petersohn, C. *Data and Control Flow diagrams, Statecharts and Z: Their Formalisation, Integration and Real-Time Extension*. Ph.D. Thesis 9801, Institut für Informatik und Praktische Mathematik det Christian-Albrechts-Universität zu Kiel, 1997.

[121] Peterson, J. *Petri Net Theory and the Modelling of Systems*. Prentice-Hall, 1981.

[122] Potts, C. Software Engineering Research Revisited. *IEEE Software*, 10(5):19 – 28, September 1993.

[123] Price, D., editor. *ISO/FDIS 10303-22 Product Data Representation and Exchange: Standard Data Access Interface*. ISO, 1 st edition, 1997.

[124] Price, D., editor. *ISO/PDTS 10303-28 Product Data Representation and Exchange: Implementation Methods: XML Representation of EXPRESS Schemas and Data*. ISO, 2000.

[125] Price, D. Step Modularization Overview. Technical Report TC 184/SC4/WG10 N235, ISO, October 2000.

[126] Ptack, K. R., editor. *IEEE Standard for Application and Management of the Systems Engineering Process*. IEEE Press, December 1998.

[127] Rechtin, E. *System Architecting: Creating and Building Complex Systems*. Prentice Hall, 1991.

[128] Richter, G. and Maffeo, B. Towards a Rigorous Interpretation of ESML - Extended Systems Modeling Language. *IEEE Transactions on Software Engineering*, 19(2):165 – 180, February 1993.

[129] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. *Object Oriented Modelling and design*. Prentice Hall, 1991.

[130] Rumbaugh, J., Jacobson, I., and Booch, G. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1998.

[131] Sage, A., editor. *Systems Engineering: Mathodology & Applications*. IEEE Press, 1 st edition, 1977.

[132] Sage, A. P. and Rouse, W. B. *Handbook of Systems Engineering*

and Management, chapter 1 An Introduction to Systems Engineering and Systems Management, pages 1 – 111. John Wiley and sons, 1999.

[133] Schenk, D. and Wilson, P. *Information Modeling: The EXPRESS Way*. 1994.

[134] Schier, J., Walker, J., Nallon, J., and Kolberg, D. Tool Integration Interest Group report: Scenarios Leading Towards a Concept of Operations for an Integrated Systems Engineering Environment. Technical report, INCOSE, 1996.

[135] Sciore, E., Sigel, M., and Rosenthal, A. Using Semantic Values to Facilitate Interoperability among Heterogeneous Information Systems. *ACM Transactions on Database Systems*, 19(2):254 – 290, June 1994.

[136] Scott, W., Cook, S., Harris, D., Smith, J., and Johnson, J. An Initial Application of AP-233. In *Proceedings 2002 international Symposium of INCOSE*. INCOSE, July 2002.

[137] Sheard, S. Twelve Systems Engineering Roles. In *Proceedings of the INCOSE*. INCOSE, 1996.

[138] Shipman, D. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, 1981.

[139] Shumate, K. C. and Keller, M. M. *Software Specification and Design - A Disciplined Approach for Real Time Systems*. John Wiley & Sons Inc., 1 edition, 1992.

[140] Simpson, H. The Mascot Method. *Software Engineering Journal*, pages 103 – 120, May 1986.

[141] Simpson, I. and Weiner, J. A., editors. *Oxford English dictionary*, volume 17. Oxford University Press, 2 edition, 1989.

[142] Stevens, R., Brook, P., Jackson, K., and Arnold, S. *Systems Engineering Coping with Complexity*. Prentice hall, 1 st edition, 1998.

[143] Storey, N. *Safety Critical Computer Systems*. Addison-Wesley, 1996.

[144] Svensson, D. *On Product Structure Management Throughout the*

Product Life Cycle. Licenciate thesis, Machine and Vehicle Design, School of Mechanical and Vehicular Engineering, Chalmers University of Technology, 2000.

[145] Svensson, D., Malmström, J., Pikosz, P., and Malmström, J. A Framework for Modelling and Analysis of Engineering Information Management Systems. In *Proceedings of the 1999 ASME Design Engineering Technical Conferences*. ASME, 1999.

[146] SysML. Systems Modelling Language. www.sysml.org, March 2004.

[147] Tao, Y. and Kung, C. Formal Definition and Verification of Data Flow Diagrams. *Journal of Systems and Software*, 16(1):29 – 36, 1991.

[148] Turner, K., editor. *Using Formal Description Techniques - An Introduction to Estelle, Lotos and SDL*. John Wiley and sons, 1993.

[149] van den Hamer, P. and Lepoeter, K. Managing Design Data: The Five Dimensions of CAD Frameworks, Configuration Management, and Product Data Management. *Proceedings of the IEEE*, 84(1):42 – 56, January 1996.

[150] van der Putten, P. and Voeten, J. *Specification of Reactive Hardware/Software Systems*. PhD thesis, Technische Univeriteit Eindhoven, 1997.

[151] Vitech. Vitech CORE. Internet, March 2004.

[152] Ward, P. and Mellor, S. *Structured Development for Real-Time Systems*, volume 1 - 3. Prentice-Hall, 1985.

[153] Ward, P. T. Embedded Behaviour Pattern Languages: a Contribution to a Taxonomy of Case Languages. In Shriver, B. D., editor, *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, volume 2, pages 273 – 284, January 1988.

[154] Warthen, B. Implementable ARMs. Technical Report ISO TC184/SC4/WG3/N530, ISO, 1996.

[155] Wymore, W. *Model-Based Systems Engineering*. CRC Press, 1993.

[156] Wymore, W. Model-Based Systems Engineering. *The Journal of*

NCOSE, 1(1):83 – 92, July 1994.

[157] Zaniolo, C. The Database Language GEM. In *Proceedings 1983 ACM - SIGMOD Conference on Management of data*, May 1983.