KTH Industrial Engineering
and Management

A Model Management and Integration Platform for Mechatronics Product Development

Jad El-khoury

Doctoral Thesis
Stockholm, Sweden 2006

TRITA – MMK 2006:03
ISSN 1400-1179
ISRN/KTH/MMK/R-06/03-SE
ISBN 91-7178-268-0

School of Industrial Engineering
and Management
Department of Machine Design
Royal Institute of Technology
SE-100 44 Stockholm, Sweden

Academic thesis, which with the approval of Kungliga Tekniska Högskolan, will be presented for public review in fulfilment of the requirements for a Doctorate of Engineering in Machine Design. The public review is held at Kungliga Tekniska Högskolan, Salongen, Osquars backe 31, Stockholm; at 10:00 on the 3rd of March 2006.

Abstract

Mechatronics development requires the close collaboration of various specialist teams and engineering disciplines. Developers from the different disciplines use domain-specific tools to specify and analyse the system of interest. This leads to different views of the system, each targeting a specific audience, using that audience's familiar language, and concentrating on that audience's concerns. Successful system development requires that the views of all developers produced by the different tools are well integrated into a whole, reducing any risks of inconsistencies and conflicts in the design information specified.

This thesis discusses techniques of managing and integrating the views from various disciplines, taking better advantage of multidisciplinary, model-based, development. A Model Data Management (MDM) platform that generically manages models from the various domain-specific tools used in development is presented. The platform is viewed as a unification of the management functionalities typically provided by the discipline-specific PDM and SCM systems. The unification is achieved by unifying the kind of objects it manages – models. View integration is considered as an integral functionality of this platform.

In demonstrating the platform's feasibility, a generic version management functionality of models is implemented. In addition, model integration is investigated for the allocation of system functions onto the implementing hardware architecture. The proposed approach promotes the independent development of the views, allowing developers from each discipline to work concurrently, yet ensuring the completeness, correctness and analysis of any inter-view design decisions made.

The prototype MDM platform builds on existing technologies from each of the mechanical and software disciplines. The proposed MDM system is built based on a configurable PDM system, given its maturity and ability to manage model contents appropriately. At the same time, the version control functionality borrows ideas from the fine-grained version control algorithms in the software discipline.

The platform is argued to be feasible given the move towards model-based development in software engineering, bringing the discipline's needs closer to those of the hardware discipline. This leads the way for an easier and more effective integrated management platform satisfying the needs of both disciplines using a common set of mechanisms.

Table of Contents

Preface

The work presented in this thesis is partially funded by the Swedish Strategic Research Foundation through the SAVE project, the national Swedish Real-Time Systems research initiative ARTES, and the Swedish Governmental Agency for Innovation Systems (VINNOVA) through the CODEX project, a collaborative effort between Scania and KTH.

On a personal note, I would first like to show my appreciation to my supervisor *Martin Törngren* for giving me the opportunity to do my PhD studies under his supervision. Thanks also to my colleagues at the Mechatronics Lab, current and old, for providing a pleasant working environment, and for putting up with me all these years. The *Ola*s deserve special recognition for a great cooperation and interesting discussions. My senior, *Redell*! Your role as a co-supervision was vital for my survival. Thank you for your Swedish lessons, and for not giving up on my stubbornness to learn. (Jag tycker att jag börja fatta nu.) My junior, *Larses*! Your enthusiasm for Hårdrock, Capitalism and Sprit will be missed. A salute to my old colleague and friend *Jonas*, our brother in the struggle, for all the laughs and good times we've had in the room and outside.

My thoughts go to all at the Department of Machine Design for the small chats in the photocopier room, 'Good Mornings', lunches, etc. Special thanks go to the administration team. In particular, the admin girls, *Heléne Hedin* and *Jacqueline Bernsten*, for their long dedication for a good drink - I mean for their efficient personnel and economic support. Unfortunately, my work did not involve much welding. Just the same, a thank goes to *Ulf Andorff* and *Kurt Lindqvist* for supervising and blessing my early arrivals to work, during their lunch breaks. You made the dreaded fifteen minute long walk to work less painful. An appreciation goes to *Peter Reuterås*, *Tobias Grundel* and *Payam Madjidi* for their superb IT-support. Most importantly, the Fredagsöl gang leader, *Martin Grimheden*, deserves a special mention for being there late enough to initiate a Friday Beer, any time of the week. Your dedication is appreciated by many enduring PhD students.

Hugs and kisses go to my Family in Sweden: *Anja*, *Anja*, *Farnosh*, *Laurent* and *Mike*. Thank you for your moral support and sustaining my needs for a good chat over coffee breaks, lunches or the odd drink.

Elin! A superb job at reminding me about post-PhD life. Thank you for being there, listening and sometimes ignoring my whining.

I am grateful for my parents, *Marwan* and *Nawal*, for giving me the opportunities to be where I am today. Your unquestioning support is invaluable. Thanks sister *Josiane*; brothers *Walid* and *Toufic*; and dear friends *Sam* and *Toufic* for your arbitrary phone calls in the middle of the night, just because you felt like a chat. I know you care.

Last but certainly not least, for the one person without whom this work would not have been possible: *Me*. Thank you for putting up all these six years. I promise I will not put you through this again. I have said it before, and I say it again: '*Jad*! You're a Genius!'

Stockholm, January 2006

Jad El-khoury

No other humans or animals were harmed
in the making of this thesis.

Appended Publications

Paper A

El-khoury J., Redell O. and Törngren M., A Tool Integration Platform for Multi-Disciplinary Development, 31st Euromicro Conference on Software Engineering and Advanced Applications, 2005.

The platform and algorithms presented are implemented by Jad El-khoury. The paper was written in close collaboration between the authors.

Paper B

El-khoury J. and Redell O., Towards a Multi-View Modelling Environment for Mechatronics Systems, Technical report, ISRN/KTH/MMK/R-05/24-SE, TRITA-MMK 2005:24, ISSN 1400-1179, Department of Machine Design, KTH, 2005.

The work presented in the paper and the writing was made by Jad El-khoury. Ola Redell assisted in discussing and provided input on the mechanisms and implementation details.

Paper C

El-khoury J., Model Data Management – Towards a common solution for PDM/SCM systems, Twelfth International Software Configuration Management Workshop (SCM-12), 2005.

Paper D

El-khoury J., The Version Control Algorithm Implementation in the Model Data Management (MDM) Platform, Technical report, ISRN/KTH/MMK/R-05/27-SE, TRITA-MMK 2005:27, ISSN 1400-1179, Department of Machine Design, KTH, 2005.

Other Publications by the Author

Publications Discussed in this Thesis:

1. Larses O. and El-khoury J., Function Modelling to Improve Software Documentation. Technical report, ISRN/KTH/MMK/R-05/25-SE, TRITA-MMK 2005:25, ISSN 1400-1179, Department of Machine Design, KTH, 2005.
2. Redell O., El-khoury J. and Törngren M., The AIDA-toolset for design and implementation analysis of distributed real-time control systems. Microprocessors and Microsystems, volume 28, 2004.
3. El-khoury J., Chen D. and Törngren M., A survey of modelling approaches for embedded computer control systems (Version 2.0), Technical report, ISRN/KTH/MMK/R-03/11-SE, TRITA-MMK 2003:36, ISSN 1400-1179, Department of Machine Design, KTH, 2003.
4. El-Khoury J. and Törngren M., Towards a Toolset for Architectural Design of Distributed Real-Time Control Systems, Proceedings of Real-Time Systems Symposium (RTSS), 2001.

Other Publications:

1. El-khoury J., Redell O. and Törngren M., Integrating views in a multi-view modelling environment, Proceedings of the 15th International Symposium of the Systems Engineering Conference, 2005.
2. Larses O. and El-khoury J., Views on General System Theory, Technical report, ISRN/KTH/MMK/R-05/10-SE,TRITA-MMK 2005:10 ISSN 1400-1179, Department of Machine Design, KTH, 2005.
3. Larses O. and El-khoury J., Multidisciplinary Modeling and Tool Support for EE Architecture Design, 30th World Automotive Congress, FISITA, 2004.
4. Chen D. J., El-Khoury J. and Törngren M., A Modeling Framework for Automotive Embedded Control Systems. SAE World Congress, SAE Technical Paper Series 2004-01-0721, 2004.
5. Henriksson D., Redell O., El-khoury J., Törngren M. and Årzén K., Tools for Real-time Control Systems CoDesign - A Survey, Department of Automatic Control, Lund Institute of Technology, Internal report - ISRN LUTFD2/TFRT—7611—SE, 2004.
6. Törngren M., El-khoury J., Sanfridson M. and Redell O., Modelling and Simulation of Embedded Computer Control Systems: Problem Formulation, Technical report, TRITA-MMK 2001:3, ISSN 1400-1179, ISRN KTH/MMK/R--01/3--SE, 2001

1. Introduction

With the introduction of computer technology as a feature in mechanical engineering products, a change is experienced in the expected functionality of these mechatronics products, as well as the means of their development. The use of micro-controllers, software, and network systems in modern technical products has permitted functionality that would otherwise be impossible or very expensive. The contribution of this technology is indispensable, and product success is increasingly dependant on it. More resources are allocated to computer technology, in order to gain an edge over competing products. For example, in the ever increasing complexity of automotive electronics, roughly 70% of functional innovations are made possible and performed by software [1].

The advantages of introducing computer technology in modern products come at the cost of increasing the product development complexity, where designers are facing many challenges to ensure that the products meet their requirements.

One source of complexity is due to the dramatic increase in the number of software-based functions in the system. For example, in the automotive industry, X-by-wire functions are projected to boost the share of electronics in chassis production from today’s 12% to approximately 40% within the next ten years [2]. While the functions themselves can vary in complexity, the sheer number of these functions forms a development challenge for the complete system. Weinberg [3] discusses the issue of system complexity as related to its size. In promoting his General Systems Thinking, he declares that ‘To a first approximation, we were able to use the number of objects as a measure of complexity – the complement of simplicity’. The challenge is to handle systems of ‘organised complexity’ – systems that are too complex for analysis and too organised for statistics.

Complexity is further compounded by the dependencies between the system functions. Previously standalone functions are becoming more interdependent, where functions need to share common resources, as well as cooperate with each other in order to fulfil their expected behaviour. Besides these functional dependencies, other types of relationships need to be considered during system development such as the mission-criticality or the strategic make/buy relationships between functions [4].

Complexity is not an inherent property of the system itself, but lies in the relation between the system and its observer. Depending on the observer's concerns, different types of objects and relations between them are perceived. For example, given the automation facilities in a modern car, its driver does not necessarily perceive the system complexity in the same manner as its developer that needs to provide such automation support.

In discussing the complexity problems of science, Checkland explains in [5] that the world is a giant complex, and to cope with it, we are forced to reduce it into separate areas which can be examined separately. This arrangement of knowledge is inevitable given our limited ability to take in the whole. 'Our knowledge of the world is thus necessarily divided into different "subjects" or "disciplines"'.

Similarly, when dealing with system development complexity, multidisciplinarity may become a necessity. Mechatronics systems development requires the close collaboration of various specialist teams and engineering disciplines. In automotive system design, for example, developers from the many disciplines of engineering, such as control, software, mechanical and electrical engineering, need to interact to meet the demands for dependable and cost-efficient integrated systems.

The developers from the different disciplines use their own specific tools, providing their own specific views of the system to be developed. Each system view targets a specific audience, using that audience's familiar language (viewpoint), and concentrating on that audience's concerns [6]. Figure 1 illustrates some of the viewpoints and views that may be necessary during the development of a typical vehicular system.

However, multidisciplinarity may in turn become a source of complexity. Developers from the different disciplines differ in the design concerns and interests in which they are involved. These concerns and interests are not necessarily exclusive, which leads to overlap and dependencies in their development information space. Even though they attempt to develop the same system, developers from the different disciplines may then form a different perception of the system's aims, problems and solutions. This becomes a source of conflict and complexity during development.

To take full advantage of multidisciplinary development, it is essential to have good integration of the efforts of all involved disciplines, as well as good communication between them. For successful system development, the views of all developers produced by the different tools should be well integrated into a whole, reducing any risks of inconsistencies and conflicts in the design information specified in these views.

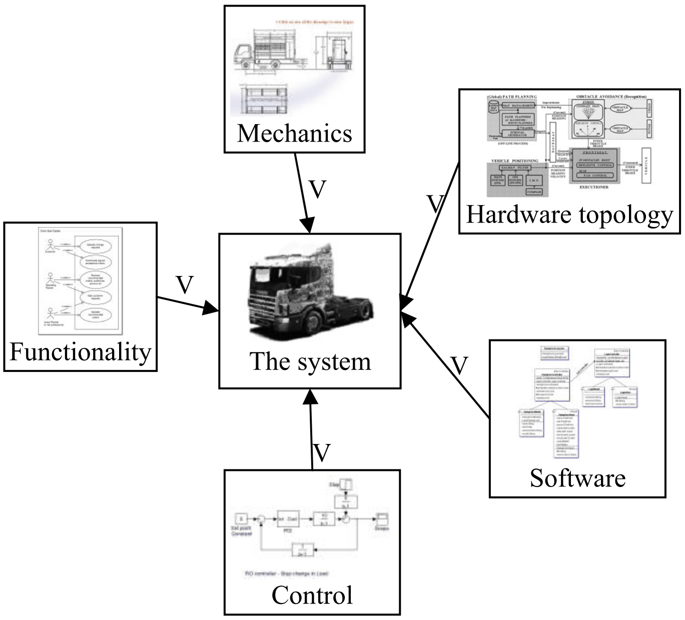


Figure 1. Some of the disciplines and views in system development.

This thesis discusses techniques of managing and integrating the views from the various disciplines, in order to minimise the complexity due to multidisciplinary development, while optimising its benefits.

Prior to presenting the contribution of this thesis, some earlier experiences within the research project in multidisciplinary tool development are discussed in the following section. These experiences justified and inspired the aim and approach advocated in this thesis, which will be detailed in sections 3 and 4. Section 5 introduces the particular thesis contributions, further detailed in the appended papers. A survey of modelling and integration approaches is then presented in section 6, followed by a summary of relevant industrial case studies in section 7. Finally, future work is discussed in section 8 before concluding in section 9.

2. Background - Earlier Attempts

This section presents earlier efforts made within this research project at developing modelling and analysis tools to support certain aspects of mechatronics system design. The aim and approach dealt with in this thesis are motivated by first hand experiences in tool and model integration, discovered by the author when developing and using these tools. A more complete description of the Aida-toolset and XILO tools can be found in [7] and [8] respectively.

2.1. The AIDA-toolset – A Real-time System Design Tool

The Aida-toolset integrates the specification and performance analysis of control systems with embedded real-time system design. Various aspects of the system can be described, from the control system specification to its implementation on a distributed network of processors.

The aim of the toolset is to help the user evaluate a number of different system designs before the actual realisation of the system. Design iterations may include changes in the software structuring, function allocations, hardware structuring, process priorities, process scheduling, communication protocols, etc. Evaluations are based on timing analyses as well as simulations of the resulting control system performance.

The AIDA-toolset is designed to support one particular work-flow, visualized in figure 2, leading to a specific precedence in the order of building the models. Initially, a pure control specification is designed and tested using Matlab/Simulink [9], within which control performance analysis can be performed by simulation. The resulting control algorithm and system dynamics provide the necessary information for the software specification. At this stage of development, important requirements such as controller jitter and delays are often overlooked, since they are dependant on implementation details and their values can only be deduced once the system is implemented. Next the control design is imported into the AIDA-toolset where the Simulink model is translated to a data-flow diagram. The resulting model is augmented with additional information such as execution times for functions and size of data-flows. This model becomes the base for the real-time system design. In the real-time system design, the user defines the target hardware, allocates the functions to processors, maps the functions into processes and specifies communication, triggering and scheduling related characteristics. When the real-time design is complete, response time analysis techniques are used to calculate the response times and release jitter of the processes and their contained functions. Once successfully analysed, the model is exported back to Simulink for further simulation. The new Simulink diagram is a copy of the original, augmented with the implementation-induced time delays. These implementation effects are hence taken into account in the resulting control performance analysis.

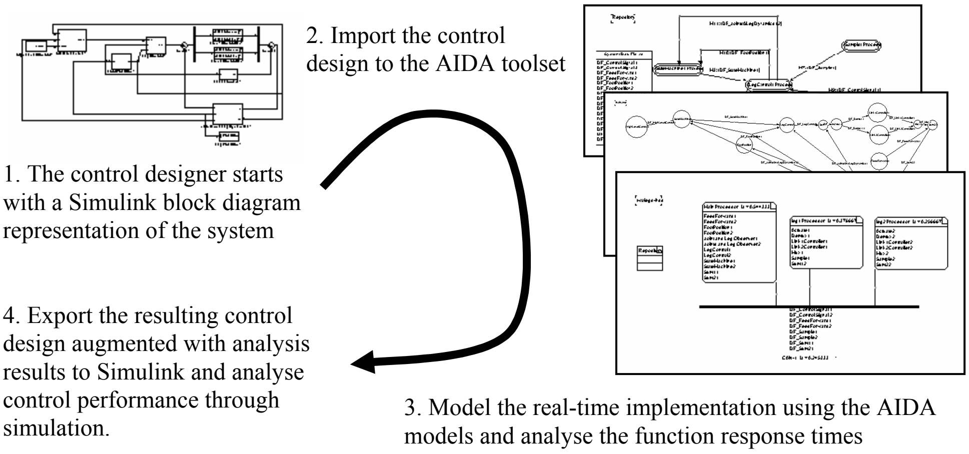

1. The control designer starts with a Simulink block diagram representation of the system
2. Import the control design to the AIDA toolset
3. Model the real-time implementation using the AIDA models and analyse the function response times
4. Export the resulting control design augmented with analysis results to Simulink and analyse control performance through simulation.

Figure 2. The work flow supported by the AIDA-toolset. Three different system views in the AIDA-toolset are represented to the right: a Process Structure Diagram, a Data Flow Diagram and a Hardware Structure Diagram.

The models used in the Aida-toolset are based on a larger modelling framework for mechatronics systems [10]. In this framework, sixteen different models are defined, of which seven are used in the toolset:

- The *data-flow diagram* (DFD) defines functions specifying the system functionality and data-flows specifying the data exchange between these functions.
- The *function timing and triggering diagram* (FTTD) defines the required time precedence relations between these functions.
- The *hardware structure diagram* (HSD) describes the structure of the target computer hardware.

- The *process timing and triggering diagram* (PTTD) defines, for each processor in the system, the timing and triggering properties of its set of processes and the mapping of functions into processes.
- The *process structure diagram* (PSD) defines the inter-process messages, based on the data-flow information from the DFD and the processes described in the PTTDs.
- The *communication link diagram* (CLD) defines, for each communication bus, the communication frames based on the messages defined in the PSD.
- The *process internal timing and triggering* diagram defines, for each process in the system, the time precedence relations between the functions allocated to the process.

The environment of the Aida-toolset is based on two separate tools: DoME [11] and Matlab/Simulink [9]. The use of the single tool, DoME, for the real-time domain modelling allows easy integration and exchange of data between models, given its provided facilities to define new domain-specific models. Matlab/Simulink was chosen for its good support of control design and simulation capabilities, which are also used to evaluate the implementation architecture developed. These capabilities could not be provided in the DoME environment. As shown in figure 3, the Aida-toolset consists of three major parts:

- Aidasign - The real-time system modelling environment.
- Aidalyze - The response time analysis tool, implemented in C++, performing timing analysis methods for distributed real-time systems [12].
- The interface with Matlab/Simulink - connects Aidasign to Matlab/Simulink, enabling import of Simulink data flow diagrams to Aidasign and later export to Simulink.

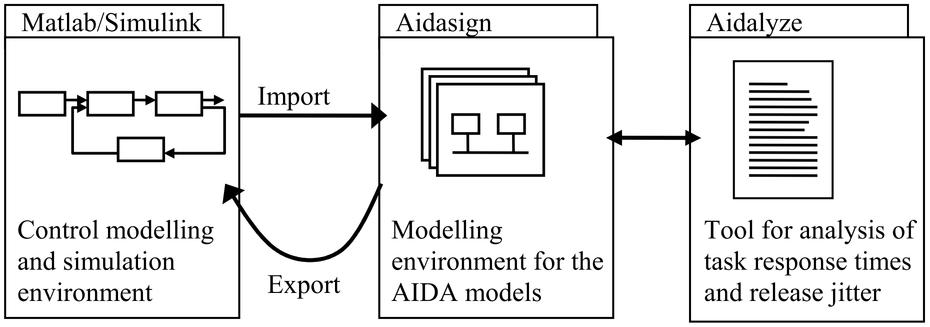

Matlab/Simulink
Aidasign
Aidalyze
Import
Export
Control modelling and simulation environment
Modelling environment for the AIDA models
Tool for analysis of task response times and release jitter

Figure 3. Architectural overview of the AIDA-toolset, highlighting its three major parts and their relations.

2.2. XILO – A Control/Scheduling Co-simulation Tool

The XILO tool supports the design of distributed real-time control systems, through the modelling and co-simulation of control functionality together with the controlled processes and the behaviour of the computer system. The co-simulation of scheduling and other implementation-related mechanisms with the control application allows the user to directly study the impact of such design decisions on the resulting system behaviour. The tool promotes interdisciplinary design by combining the views of control and computer engineering into one view.

The workflow supported by XILO is similar to that of the AIDA-toolset, visualized in figure 2, with the following differences:

- The complete set of XILO models are developed within the same environment. Hence, there is no need to perform import/export of the models between tools.
- In XILO, the analysis is only performed through the co-simulation of the application software behaviour, together with the system software and hardware behaviour.

In order to achieve the goal of a multidisciplinary modelling environment, modelling aspects were borrowed from a number of sources:

- The AIDA modelling framework [10] provided insights into the control implementation requirements needed, the component models and their parameters.
- The CODARTS method [13], as a software engineering design methodology and model, highlighted the aspects of software that need to be included.
- Data flow diagrams from the control engineering approach were used for the modelling of the application functionality.

XILO allows the modelling and simulation of the following views:

- *Application software* encompassing different functionalities in a wide variety of styles (e.g. discrete-time, even-triggered, data-flow, state machines etc.).
- *System software* including the behaviour of the operating system scheduling and inter-thread communication protocols.
- *Distributed computer systems* including communication networks and computer nodes.
- *Mechanical systems* including sensors, actuators and mechanical system dynamics.

The various views are modelled within a single hierarchy. At the top level, the hardware topology of the whole system is modelled. This hardware structure consists of three types of components: (1) The *environment* modelling the mechanical dynamics of the system including sensors and actuators; (2) *Communication Links* defining the communication protocols between computer nodes; and (3) *Computer Nodes* in which the application and system software is modelled.

Within each computer node, the software structure is defined through: (1) *Tasks* defining the application software; (2) A *task scheduler* modelling a wide range of schedulers such as event/time triggered, static/dynamic, and off-line/on-line schedulers; (3) *Operating system services* such as inter-task communication, task synchronisation and semaphores and (4) *Hardware drivers* such as communication controllers, timers, ADCs and DACs.

Finally, within each software task, the application functionality is defined as a sequence of sub-functions.

The XILO tool is based on a set of library components for the modelling of standard functionalities such as schedulers, communication mechanisms and basic operating system services. This approach allows the developers to evaluate a number of different system designs, by the simple exchange and reconfiguration of components.

The environment used to build and execute the models is Matlab/Simulink. This environment is biased towards the control engineer environment, allowing the control engineer to specify, validate and interact with the computer engineer in a familiar environment.

2.3. Integration Experiences

2.3.1. Tool Integration

In the Aida-toolset, the relationships between the various models are outlined in figure 4, where solid arrows correspond to subdiagram relationships while dashed arrows indicate import relationships between tools.

From a usability perspective, it is desired to transparently integrate the tools. Since Matlab/Simulink and DoME tools have no common mechanisms that enable direct communication between them, integration of the models is performed through import/export mechanisms. The import mechanism of the Aida-toolset allows the translation of a Simulink model into a DFD model, through a one-to-one mapping from Simulink blocks to DFD functions. Once a Simulink model has been imported into the AIDA-toolset, additional information such as function execution times and data-flow sizes can be specified. However, to enable future export to Simulink, the model may not be otherwise modified, since the export mechanism assumes the structure of original imported Simulink model. This restriction undesirably creates a precedence relation between the models from the different tools, preventing their parallel and independent development.

In comparison, the XILO tool handles all models within a single tool and hence avoids the problem of tool integration. The adopted tool is however not necessarily optimal for software and hardware development.

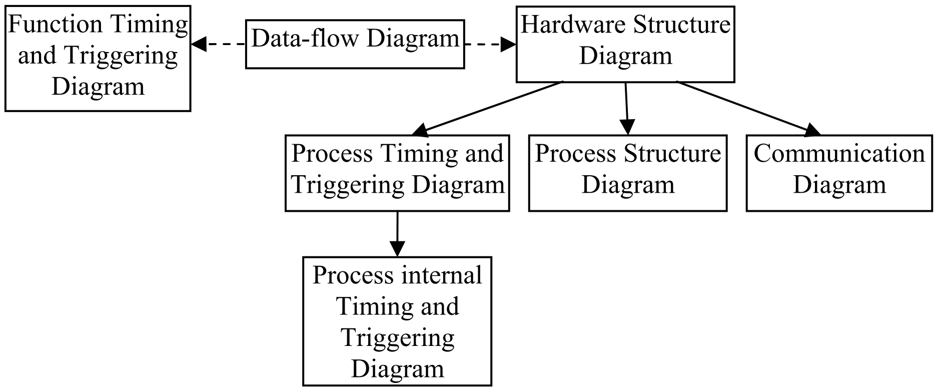

Function Timing and Triggering Diagram
Data-flow Diagram
Hardware Structure Diagram
Process Timing and Triggering Diagram
Process Structure Diagram
Communication Diagram
Process internal Timing and Triggering Diagram

Figure 4. The structure of the models in the AIDA-toolset, where solid arrows denote subdiagram relationships while dashed arrows denote import relationships.

2.3.2. View Integration

Within the Aida-toolset models, a challenge in having the many different views is to keep the models consistent, whereby changes of information in one model are propagated to other related models that share the information. The use of a central database to manage all data shared by the models in the toolset was identified as a need to avoid the problem of inconsistency. This was not possible due to DoME limitations. Instead, the approach taken was to, for each piece of data, designate one model that is the data owner, while the other dependent models operate on data copies. Data is then automatically updated, when manually triggered by the user, and in this way regaining consistency in the model set. The major drawback of this approach is that model changes are not reflected in the whole system immediately, leading to inconsistent models in the intervals between model updates.

In the XILO tool, the mapping from the control-based functional model to the real-time implementation model is not managed, and no attempt is made to maintain the models synchronised. In addition, the XILO tool avoids the consistency problem by assuming a single model structure to fit the many implementation views of the system. This approach however conflicted with the need for different viewpoints for different disciplines, allowing developers to concentrate on specific aspects.

2.4. Integrating the Aida-toolset and XILO Tools

During their development, it was realised that the Aida-toolset and XILO tools had many properties in common, leading to the intention of integrating them. This goal was deemed feasible given that the tools are inspired by the same modelling framework [10]. The main differences between the tools are presented in table 1. The tools essentially contain the same modelling content, while they mainly focus on different analysis techniques, namely timing analysis and co-simulation. It would hence be desired to provide the two complementary approaches for system analysis based on the same modelling framework, and without the need to manually duplicate the models.

Table 1. The main differences between the AIDA-toolset and the XILO tool.

	XILO	**Aida-toolset**
Analysis	Co-simulation	▪ Timing analysis ▪ simulation
Tools	One tool for all disciplines	Two domain-specific tools
View modelling	Views modelled within one hierarchy	Separate models for each view.
Analysis results	Control performance	▪ Timing behaviour in terms of worst/best case response times and jitter. ▪ Control performance

However, each analysis technique requires a specific environment to work within: the Simulink simulation environment for XILO and Dome for the Aida-toolset. The challenge is to manage the modelling content in a tool-independent manner, not favouring one tool over the other, nor creating dependencies between them. This desire directed the research interest towards model content management and tool integration.

3. Goal

This thesis aims to develop a model integration and management platform that supports the multidisciplinary, model-based development of mechatronics systems. The platform should allow for the management and sharing of the product information produced by tools and disciplines throughout the development life cycle. Consequently, various analyses can be performed based on the same information set. The platform should also facilitate the communication of information between the different stakeholders, allowing any inconsistencies and conflicts to be identified and dealt with.

Two assumptions or limitations are implicit in the above inter-disciplinary integration aim: (1) A product domain focus and (2) a model-based development approach. These are further developed in the following subsections.

3.1. The Product Domain Focus

In studying the complexity of product development, Eppinger and Salminen introduce three domains of analysis: Process, product and organisation [14]. Decomposition is used within each of these domains in order to manage the development complexity. The full development process is decomposed into phases; an organisation is decomposed into teams; and a product is decomposed into sub-systems. With the separation of development into product, process and organisation domains, the interactions between these domains can be better analysed, giving a better understanding of the complexity of product development. The interactions within and between the three domains are illustrated in figure 5.

This model of product development does not explicitly take into consideration the multidisciplinary nature of certain products. It is assumed that a single product decomposition exists within the product domain. This assumption simplifies the patterns of interaction between the product structure and the remaining domains.

However, the development of multidisciplinary products adds another dimension of development complexity, whereby within each domain, the interactions between the disciplines play an important role and need to be additionally analysed.

For example, no single product structure can be assumed in a mechatronics product. Developers from the different disciplines have their own specific viewpoints of the system to be developed. That is, different description languages and analytical methods are adopted to deal with the specific concerns of the different disciplines [6]. The need to consider the product from different viewpoints leads to different product structures – or views – of the system.

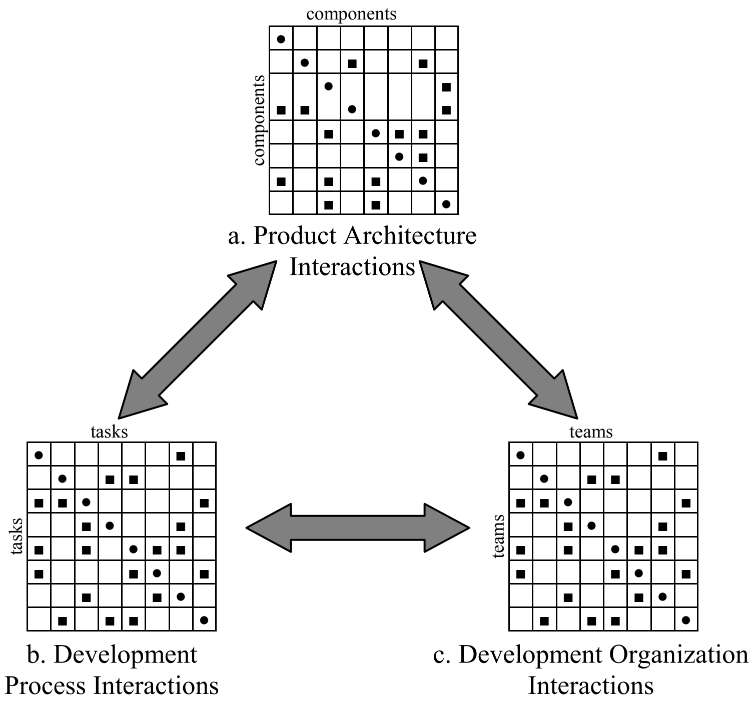

components
components
a. Product Architecture Interactions
tasks
tasks
b. Development Process Interactions
teams
teams
c. Development Organization Interactions

Figure 5. The patterns of interaction within each of the three domains of product development, as well as across them (Reproduced from [14]).

Within the product domain, the interactions between the various structures need to be analysed, in order to avoid inconsistencies between them. Similarly, the different disciplines may need to follow different development processes, leading to different process structures for each discipline [15]. In multidisciplinary development, this leads to multiple process structures. From the organisational perspective, the teams can no longer be viewed homogenously, as various members (or entire teams) may belong to specific disciplines, creating multiple organisation structures. As a result, the interactions between the domains can no longer be treated as suggested in [14], since the mapping is no longer between single structures within the domains.

Note that the source of different viewpoints (and hence the different structures) stems not only from the different needs of the disciplines. Within each discipline, different viewpoints may also be needed. The predominant system structure used in traditional mechanical development reflects the physical decomposition of the product into its designed components. On the other hand, software development employs many structures, which also need to be integrated. In UML [16], for example, many structures are adopted such as Class, Statechart, Use Case and Deployment models. In this general sense, a discipline can be viewed as a broader grouping of many views.

With this complex model in mind, the contribution of this thesis focuses on the interactions between the various disciplines within the product domain. We aim to integrate the various views produced by the different disciplines, ensuring the consistency of the information assumed from their various viewpoints, and providing a common basis for information flow between them.

It is acknowledged that the remaining domains cannot be simply ignored, and handling the complexity within one domain does influence the complexity in the remaining aspects. After all, the integration's final aim is to support the engineers in their development process. Nevertheless, it cannot be claimed that this thesis' contribution directly integrates the development processes assumed by the different disciplines, nor the integration of people within an organisation.

By formalising the interactions between the various product structures within the product domain, this thesis can form a step to understand the more complex interactions between the above three domains, assuming a multidisciplinary product and development.

3.2. Model-based Development

A precondition to be able to integrate and handle the interactions between the various product views is the availability of an explicit representation of these views. That is, models describing the product structures – and hence the product – are available.

Moreover, it does not suffice that the product models are simply provided. Instead, for successful development, tying the product, process and organisation domains together, the product models should be the basis of the development process within the organisation. Product models form the basis for the interactions and communication between the teams of the organisation; as well as the information flow between the development phases. Such a basis for development is here termed as model-based development.

Model-based development refers to a development approach whose activities emphasise the use of models, tools and analysis techniques for the documentation, communication and analysis of decisions taken at each stage of the development lifecycle. Models can take many forms such as physical prototypes, graphical and textual models. It is essential however that the models contain sufficient and consistent information about the system, allowing reproducible and reliable analysis of specific properties to be performed. In model-based development, analysis plays the critical role of ensuring that the models being built - hence the design decisions being taken – are consistent and satisfy the system requirements.

Within a given discipline, model-based development is commonly used, such as the use of CAD tools in mechanical engineering. In the maturing software engineering domain, model-based development is gaining acceptance. The popularity of modelling languages such as UML is an indication of this trend.

In multidisciplinary model-based development, several viewpoints of the system are formed by the different disciplines. This leads to several models, representing the different product structures produced. In the integration of these models, the discipline-specific description languages and analysis methods used to model these structures should be preserved. Proper model integration may become a strong basis of communication between engineers of different disciplines.

This thesis suggests an approach in which the integration of models from the various design domains is also model-based, ensuring the explicit documentation of the interactions between the product views. The state of practice of social integration [17], where informal communication between engineers tries to ensure consistency, is not desired.

Given the recent establishment of the model-based development in certain disciplines such as software engineering, the sensibility of this assumption can be questioned. According to Encyclopædia Britannica [18], 'engineering' is defined as the 'professional art of applying science to the optimum conversion of the resources of nature to the uses of humankind'. Given this definition, one can reverse the question and wonder how the application of the sciences can be validly performed during engineering activities without access to explicit and reproducible information. Product information and design decisions need to be explicitly and unambiguously documented for their communication between engineers, and to become a basis onto which scientific analysis can be performed. Engineering is a combination of craftsmanship and scientific exploration; and model-based development is a basic requirement for the latter to be possible. In other words, in order for software development to change from an art to becoming an engineering discipline, it ought to become model-based.

4. Approach

The aim of the integration platform is to integrate the different models used to represent the structures or views from the various development disciplines. In the development of large and complex products, an organisation normally adopts some kind of product management tools in order to manage the large amount of documents storing these models. For example, the development of software-intensive products relies on Software Configuration Management (SCM) systems, while mechanical system development uses Product Data Management (PDM) systems. The need to obtain consistent access to the documents storing the models leads to the necessity to coordinate the intended integration platform with these management tools.

In multi-disciplinary product development, a number of these management environments come into simultaneous use. This is necessary since developers from each discipline require the specific support provided by its corresponding management system. Integrating these environments becomes essential for the successful integration of the efforts of all disciplines involved, considering the central role they take in controlling the development process as well as facilitating the communication between developers.

In summary, a model integration platform integrating different development tools needs to be itself integrated with the management tools, which in turn need to be integrated with each other. The various integration needs are illustrated in figure 6.

Another approach to the problem is to step back and treat the view integration problem as part of the management problem already covered by PDM/SCM systems. Model integration is treated as another functionality that can be augmented to the conventionally expected functionalities of management tools. This approach is illustrated in figure 7.

In one sense, incorporating the management tools expands the integration problem. However, expanding the problem domain provides a better fit of the view integration problem. Much can be borrowed from the PDM/SCM integration efforts such as the work suggested in [15] and [19]. In addition, by absorbing the management tools into the platform, a smaller number of tools need to be integrated.

Problem simplification can also be claimed given the assumption of model-based development. As argued in section 5.3 (Paper-C), the integration of PDM/SCM is considered more feasible with this assumption, suggesting a unified platform that generically handles models from all disciplines. Based on this platform, the integration of the models from the different disciplines is made more feasible.

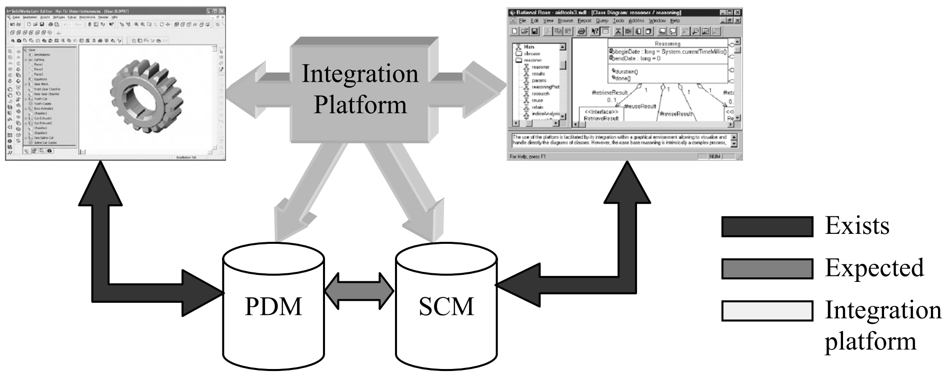


Figure 6. The integration needs of the various development and management tools for mechatronics systems.

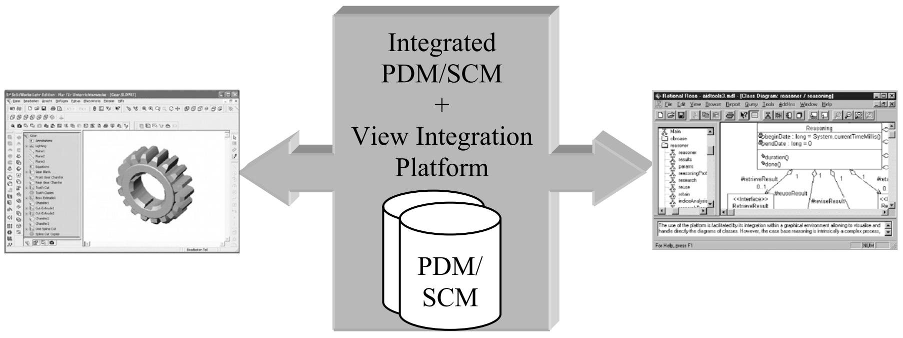


Figure 7. An integration approach treating view integration as part of the management systems.

The integration problem is reduced to that of integrating PDM and SCM systems, plus providing integration functionality based on the integrated solution. Within the context of figure 5, the approach not only contributes to the integration of the disciplines within the product domain by integrating their views, but by also contributing to the integration of the management facilities such as process control, workflow control, user management, etc. These facilities are used in the process and organisational domains, leading to a better alignment of the three domains.

4.1. Model and Tool Integration

Model integration is made a lot easier if one assumes a single tool that fully supports the development of all involved views. Model management and integration can thus be provided within the tool implementation itself. While this may be desired, experience shows that no such silver bullet can be provided. Our conviction is that no matter how large and encompassing modelling tools get, one will never reach the point when a single tool will meet all the needs of a multidisciplinary development process in any organisation. As a consequence, the need to integrate model information between the tools that act on this information will always exist.

No tool in the tool-set should take a predominant role, to which all other tools integrate. Such an approach creates a dependency on that tool, and peripheral tools can only be integrated indirectly. Instead, a central platform is suggested to which tools are connected. It is through this platform that communication between tools, and the integration of their models, occurs. Naturally, dependencies are created to the integration platform, which is however expected to be more stable, as suggested in section 4.3.

4.2. Platform Requirements

In summary, the integration platform should support the following needs:

- **Support for discipline-specific tools** – It should be possible to integrate different kinds of tools from the various disciplines, recognising that different organisations will assume a different toolset.
- **Data sharing and view integration** – A tool integration mechanism should manage the duplication of information between tools, synchronizing and maintaining its consistency. In addition, having chosen a specific set of tools, certain design information ends up in between tools. This information specifies a relationship between the different views (inter-view information). Good integration mechanisms should permit the specifications of such cross-view information, reflecting points of interaction at which the respective stakeholders need to communicate.
- **Model management** – includes functionalities such as the storage of models, handling of versions and variants of models, change request management,

process/workflow management as well as support for geographically distributed development. Support for discipline-specific functionality should also be provided such as build management for software development. An integration platform ought to provide these functionalities centrally for all tools that it integrates.

4.3. Integration Cases

Caution should be taken when adopting a given integration solution, given the central role such a platform assumes in an organisation, and the dependencies it creates between developers. In addition, an integration platform is expected to outlive the many tools it integrates. While metrics such as the Return on Investment (ROI) are developed to justify investments in central systems like PDM and SCM [20], no such metrics are necessary in adopting tools such as compilers or editors, which may be used locally within an organisation and are replaced relatively more easily over time.

For these reasons, a stable, long-lasting and universal integration solution, which can anticipate future changes in tools, is to be expected.

This stability is threatened by factors such as the fast growth in modelling languages and tools, specifically for the maturing software engineering discipline. On the other hand, partial standard efforts such as the MOF modelling standard [21], formatting standards such as XML [22], and basic communication mechanisms such as CORBA [23] and COM [24], provide a valuable foundation. The appearance of the STEP [25] standard within the mechanical engineering discipline is historical evidence that such efforts are possible.

In this thesis, it is recognised that achieving the stability expected of an integration platform is very much a standardisation effort. For this reason, focus is instead placed on two cases of integration techniques to cover each of the main needs specified above: view integration and model management.

Concerning view integration, the integration of the system functional view to the hardware architecture view, through the allocation of functions to hardware components, is investigated. With each view related to a different discipline, this example highlights the multidisciplinary problem. Further details are discussed in section 5.2 and Paper-B.

Concerning model management, a generic version management functionality of models is investigated. While version control is needed in both the mechanical and software disciplines, the functionality differs between SCM and PDM systems. This allows us to investigate how far such mechanisms can be aligned between the disciplines. Version control is also critical since it will put to the test the other crucial management functionalities of any common management system such as the possibility of having a common product structure and data representation. Further details are discussed in section 5.4 and Paper-D.

Finally, to satisfy the need to support discipline-specific tools, these cases need to be dealt with assuming different modelling tools.

5. Summary of Appended Papers

This section provides a summary of the appended papers of this thesis. The combination of these papers provides a good description of the tool integration platform.

The reader is advised to read these papers before proceeding with the remaining chapters of the thesis.

5.1. Paper A - A Tool Integration Platform for Multi-Disciplinary Development

This paper presents the architecture for the Model Data Management (MDM) platform that aims to satisfy the needs for tool and model integration presented in section 4.2. MDM generically manages and integrates models from the various tools used in the development of mechatronics products.

The platform aims to provide generic model management functionalities including supporting the storage of models, handling of versions and variants of models, access control, change request management, process/workflow management as well as support for geographically distributed development. This is viewed as a unification of the management functionalities typically provided by the discipline-specific PDM and SCM systems traditionally used in the hardware and software disciplines respectively. The model-based approach to data management unifies the software and hardware disciplines by unifying the kinds of objects it manages – models. The model-based management functionalities and the need to interrelate the internal model contents require that the platform manages the fine-grained details of each model from the integrated tools.

The architecture supports the decoupling of the modelling tools from the MDM platform, permitting an open architecture where various tools can be integrated as desired. This is made possible through the adaption layer that maps the tool-specific format and meta-model, used internally by the tool to manage its model data, to the generic format and meta-model of the platform.

The proposed architecture explores the idea of building on existing technologies from the more mature discipline of mechanical engineering, as well as borrowing advanced functionalities from the software domain. MDM is built based on a configurable PDM system. PDM is adopted due to its maturity and ability to define information models, with a high level query language to access and modify the model data in the repository. In addition, it is envisaged that the development of the remaining MDM functionalities is made easier given the already developed functionalities of PDM such as the support for distributed development, change management, workflow control, etc. At the same time, the version control functionality borrows ideas from the fine-grained version control algorithms in the software discipline.

Model management functionalities are illustrated through the implementation of the version control algorithm of Paper-D. In addition, model integration techniques are provided, where model content can be shared across different tools. This is illustrated in the partial implementation of the view integration mechanisms proposed in Paper-B.

5.2. Paper B - Towards a Multi-View Modelling Environment for Mechatronics Systems

The paper presents an approach to multi-view modelling and integration which systematically integrates the two generally accepted complexity reduction techniques of multi-view and hierarchical decomposition. The approach defines how inter-view relationships can be used to tightly interweave the views' hierarchies.

Through the use of a case study, model integration is investigated for the allocation of system functions onto the implementing hardware architecture. The resulting approach maintains the principle of hierarchical design within, as well as between the views, where allocation can be performed at arbitrary levels across the hardware and function hierarchies. The proposed approach promotes the independent development of the views, allowing developers from each discipline to work concurrently, yet providing support for a holistic view.

Mechanisms are defined to ensure the completeness and correctness of any inter-view design decisions made, as well as, to perform cross-view keyfigure analyses. The principle that a part of the complete system is a system of its own, with its own set of views is reinforced, with the possibilities to perform cross-view analysis on the complete system as well as its individual parts.

The feasibility of the inter-view mechanisms is investigated through the implementation of a prototype tool, in which views, as well as, inter-view design information and analysis, could be performed. In addition, a partial implementation of the approach is developed based on the MDM platform of Paper-A. Through a generic inter-view association mechanism, the model data from different tools can be interrelated. This acknowledges the need for the different views to be modelled using domain-specific tools. The integration platform takes a centralisation role in which the inter-tool information is managed and stored.

The paper also presents the meta-meta-model of the MDM platform. A simple meta-meta-model is adopted, allowing focus to be placed on the view integration mechanisms and the management functionalities of interest.

5.3. Paper C - Model Data Management – Towards a common solution for PDM/SCM systems

This paper investigates the effect of adopting model-based development in software engineering in bringing the discipline closer to the hardware engineering discipline and permitting a tighter integration of their management systems. The investigation considers the three crucial factors for a successful integration: tools and technologies, processes, and people [26].

It is argued that, as software development becomes increasingly model-based, its needs become closer to those of hardware development. In particular, the process management and information modelling functionalities expected of SCM systems come closer to those provided by PDM systems for hardware development. This leads the way for a more effective integrated management platform satisfying the needs of both disciplines using a common set of mechanisms. The model-based approach to data management unifies the disciplines by unifying the kind of objects it manages – models. Management functionalities deal with models and their internal contents as central entities, transparent of the file structure used to store them.

The MDM platform, presented in Paper-A, provides a basis for the desired common management functionalities, by generically handling different kinds of models produced from a set of different tools and disciplines. To illustrate the suggested common management solution, a model-based version management functionality is implemented, as presented in Paper-D.

5.4. Paper D - The Version Control Algorithm Implementation in the Model Data Management (MDM) Platform

In this paper, a simple model version control functionality (MVC) was implemented, in order to exemplify the PDM/SCM integration approach suggested in Paper-C, and test its feasibility using the MDM platform of Paper-A.

While version control is needed in both the mechanical and software disciplines, the functionality differs between SCM and PDM systems. This allows us to investigate how far such mechanisms can be aligned between the disciplines. Version control is most fundamental and best validates the MDM approach since it will put to the test the other crucial PDM/SCM integration factors such as the possibility of having a common product structure and data representation. Naturally, a full validation of the approach needs to investigate the feasibility of the remaining management functionalities using the model-based approach.

MVC provides mechanisms that allow a user to save and extract any part of the system model through check-in and check-out operations respectively. This permits stakeholders to perform design activities in terms of models, where they can organise, share and modify their models, transparent to the underlying file structure.

The algorithm generically supports the fine-grained versioning of any model that can be mapped to the meta-meta-model assumed in the platform, and presented in Paper-B. In the current implementation, Data Flow Diagram (DFD) [27] models from the Matlab/Simulink tool and Hardware Structure Diagram models [7] in the Dome tool are handled.

6. A Survey of Modelling and Integration Approaches

A survey of current approaches for the modelling of embedded computer control systems was performed as part of this research project [28]. A short summary of this study is presented in this section, together with a complementary survey of representative tool integration approaches. The study was initiated to appreciate the various flavours of modelling approaches available, and understand the differences between them. The common patterns found between the approaches formed a good basis for the definition of the meta-meta-model suggested in the MDM platform (Paper-B). The tool integration solutions suggested by these approaches, and their limitations, also became a good motivation for further research on model and tool integration.

The survey aimed to study 'what' each approach models, with less focus on the details of 'how' this is performed. For this purpose, a framework for characterizing, comprehending and comparing the different approaches was developed, focusing on the modelling content. As illustrated in Figure 8, the framework combines generic modelling concepts with multiple iterations from the evaluation of twelve modelling approaches covering different levels of design and disciplines. This evolved and stabilised the framework, consolidating more precisely the defined factors.

A modelling approach refers to any support technique or solution provided for the design of embedded computer control systems, such as computer tools, languages and standards. The choice of approaches covers different application domains, disciplines and levels of design, ensuring that a broad collection of modelling features are covered.

Twelve approaches have been evaluated based on published materials from the respective developers. ACME [29], Wright [30], UniCon [31] and Rapide [32] are software Architecture Description Languages (ADL). Lustre [33] and MAST [34] have a computer science origin with formal methods and scheduling theory background respectively. VCC [35] is an approach from the automotive industry. Orccad [36], Giotto [37] and MetaH [38] are domain-specific approaches that aim at control applications to be implemented on computer systems. Finally, both Ptolemy [39] and SDL [40] focus on the high-level specification of the system, and less on implementation details.

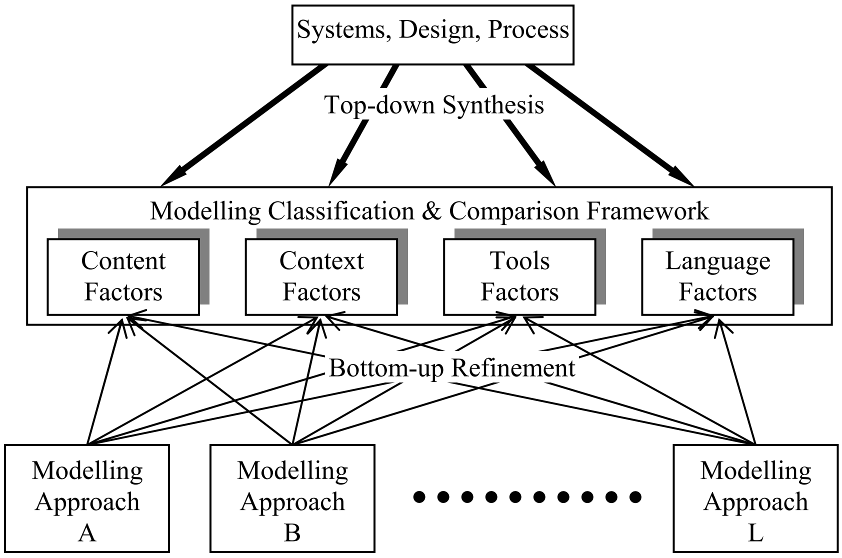


Figure 8. Technique for defining the framework – Top-down synthesis and bottom-up refinement

6.1. Comparison Framework

To compare different modelling approaches, both the model contents, as well as the design and analysis context within which the models are used, need to be taken into consideration. In the comparison framework, this is formulated using three groups of comparison factors: *modelling content*, *design context* and *analysis context*. These factors are summarized in figure 9.

The content factors aim to identify the various system aspects that can be modelled by a particular modelling approach. In this framework, a model is seen as consisting of a set of abstractions that represent real system entities. The abstractions may be classified into a set of common types. Furthermore, there exist different types of relationships between the different abstractions, such as communication between abstractions and decomposition of one abstraction into a set of other types of abstractions. Following this view on models, the set of *abstraction types*, the *properties* that define them, and the *inter-abstraction relation types* that may exist in any modelling approach are identified.

To facilitate the comparison, abstraction types, their properties and relation types most relevant for embedded control systems are predefined in the framework, as listed in figure 9. The content classification forms a common basis upon which it is possible to organise and compare the content support provided by each modelling approach.

Content
- Abstractions
  - Properties
    - Structural interface
    - Behaviour Semantics
      - Activation
      - Persistence
      - Timing
      - Error
    - Constraints
- Inter-abstraction Relations
  - Decomposition
    - Encapsulation
    - Behaviour Semantics
    - Constraints
  - Communication
    - Behaviour Semantics
    - Constraints
  - Synchronisation
    - Behaviour Semantics
    - Constraints
  - Commonality
  - Dependency
  - Refinement
  - Allocation
  - Criticality
  - Replication
  - Other

Design Context
- Levels
- Activities
- Domains & Disciplines
- Methodology
- Traceability
- Complexity Management
- Reusability

Analysis Context
- Functionality
- Performance
- Reliability
- Safety
- Other

Language
- Representation technique
- Adaptability
- Multi-views
  - Consistency guarantee

Tool
- Availability
- User interaction
- Tool integration
- System Generation

Figure 9. Comparison framework structure and factors

Within the design context, the *level of design* at which the content is used by the approach is of most interest. For comparison, four general design steps are defined, ranging from implementation-independent specifications, towards the final solution description: *functional design*, *architectural design*, *medium-level design*, and *detailed design*.

Within the analysis context, it is interesting to study the types of analysis that can be performed given the modelling content provided by the approach. For embedded computer control systems, relevant analysis types include: *functionality*, *performance*, *reliability* and *safety* analysis.

Two other groups of factors are also handled in the framework: *language* and *tool*. The former deals with the techniques and rules adopted by a modelling approach for representing its content. Even though two approaches have the same content, they may differ in the way this content is handled, used and represented in the models. Finally, the tool factors attempt to identify the computer-aided techniques and facilities available for manipulating, managing and verifying the models.

6.2. Comparison

The major part of this work was in the surveying and analysis of the modelling content of the approaches. A detailed discussion and comparison of the content can be found in the original study [28]. The procedure used to acquire the comparison framework highlights the common features between the studied approaches. Abstractions such as *communication* and *software* types; properties such as *timing*; and inter-abstraction relations such as *decomposition*, *communication*, *refinement* and *allocation* are most common between the studied approaches.

Furthermore, in structuring the modelling content, common techniques are found between the modelling approaches in order to absorb the complexity of the system being modelled. The major identifiable mechanisms for complexity management are: The widely adopted hierarchical decomposition, the use of domain-specific terminology and concepts, the repeated use of a few central concepts, good language and tool support, the division of content into multiple views, and commonality mechanisms such as typing and specialisation/generalisation.

Through the analysis of the modelling content, the design levels addressed by each modelling approach are determined, as illustrated in figure 10. In addition, table 2 presents a summary of the *available* and *possible* analysis techniques provided by each approach. Available analysis techniques are those explicitly identified and supported by an approach. Possible techniques are those that can be potentially performed, given the content supported by an approach.

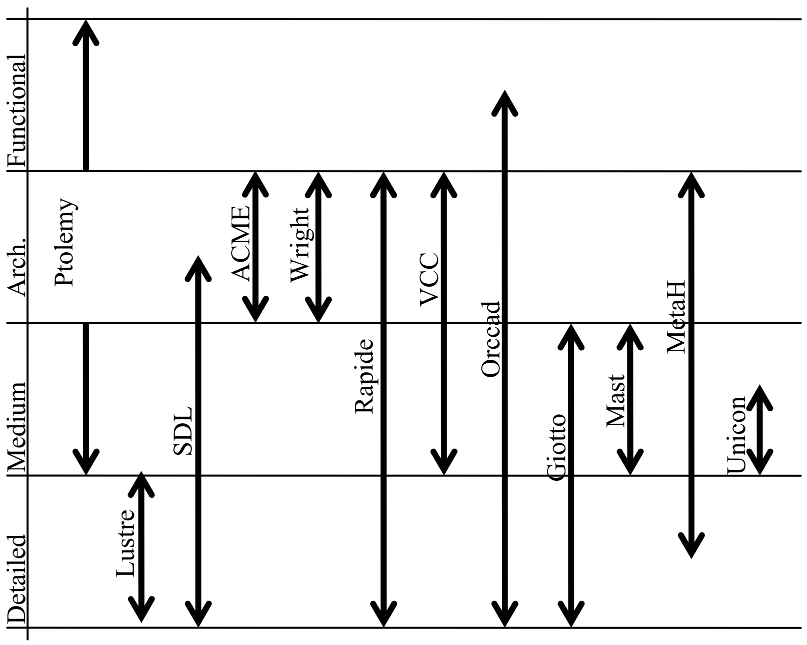


Figure 10. Design levels focused on by each modelling approach.

Table 2. Summary of available (√) and possible (+) analysis techniques

	Functionality		**Performance**			**Reliability**	**Safety**
	Simulation	**Model Checking**	**Simulation**	**Model Checking**	**Timing**		
Ptolemy	√		√				
Lustre	+	√	+	+			
SDL	+		+				
Acme							
Wright	+	√					
Rapide	√	√	+	√	+		
VCC	√		√		+		
Orccad	√	√	√	+	√		
Giotto	√	√	√		√		
MAST			√		√		
MetaH	+		+		√	√	
Unicon	+		+		√		

Concerning tool integration capabilities, the modelling approaches tend to integrate other tools in order to cover certain aspects that are weak or not covered in the original approach. Compared to integration platforms (section 6.3), such integration efforts tend to be ad-hoc, implemented to meet the current needs of the approach. For example, MetaH is integrated with ControlH for the functional description of its subprograms, and Giotto uses Simulink for graphical representations. Certain approaches become quite dependent on this integration to be usable. For example, Wright needs to have a CSP checker to perform any kind of analysis. On the other hand, MetaH can still be operable without the use of ControlH.

Much overlap exists between the content covered by the approaches. This is specifically the case for approaches that attempt to cover similar activities and analysis techniques, at the same level of design. The similarities between the ADL languages, where focus is mainly placed on software modelling at the architectural level, is a typical example. In these approaches, the main abstractions covered are components, connectors and configurations used to model system software. It can be argued that content overlap between approaches is an indication of integration potential between them. The challenge remains to coordinate the remaining content that does not entirely overlap.

Approaches covering the same activities at the same level of design can be used interchangeably. Integrating such approaches might be of interest when the different approaches provide complementary functionalities or analysis techniques. For example, the ACME ADL might be desired to use for its possibilities for generic specifications, while Wright provides analysis possibilities through simulation and model checking.

In addition, approaches covering different activities, or different design levels would be of interest to integrate to cover a wider range of design levels and activities. For example, it may be of interest to integrate an ADL such as Rapide with Ptolemy. While the latter provides higher level functional descriptions, the former can be suitable for the architectural level of design. The model of computation provided in Rapide (timed-posets) can also be complemented by the variety of models of computations provided by Ptolemy.

An abundance of modelling languages and approaches that target various aspects of system development exists. The union of these approaches may cover all that can be desired. The challenge remains however in providing such a union. A necessary component of any such integration effort is the integration of their modelling content. Ad-hoc integration, as experienced in the studied approaches, creates undesirable dependencies to the modelling tool. Instead, as discussed in section 4.1, a platform addressing the integration of tools should be used. The next subsection surveys a number of such platforms.

6.3. Tool Integration Approaches

This survey is based on the study of seven tool integration approaches: Cheops [41], Eclipse [42], Fujaba [43], GeneralStore [44], IDM [45], IMPROVE [46] and Toolnet [47].

Tool integration can be divided into two general categories: *data integration* and *control integration*. The former focuses on relating the model data produced by the different tools. On the other hand, control integration deals with tool activities such as integrating the services or functionalities provided by the tools, providing a common look and feel across the tools, controlling the workflow between the tools, managing tool interactions, etc. A typical example of control integration is the Eclipse platform for software development. Eclipse provides a plug-in based framework to create, integrate and utilize software tools. The plug-in mechanism is used to realise the services of the integrated tools, and through which tools can interact and request services from each other. However, any files and data items produced are managed internally by the integrated tools and are beyond the scope of the platform. Naturally, certain tools such as Fujaba take into consideration both aspects of integration. This section focuses mostly on data integration, given its relevance for the issues discussed in this research.

Two different needs for data integration can be identified: the integration of models covering different components of the complete system - *component model integration*; and the integration of models covering different views of the same system – *view integration*. These needs lead to different integration solutions.

The challenge in component model integration comes when the different components are modelled using different models of computation, such as the time-continuous or time-discrete models of computation. In this case, the heterogeneous models need to be appropriately coupled at their interfaces to form a complete model. From the surveyed approaches, GeneralStore and Cheops focus on component model integration of software systems and mathematical models of chemical plants respectively. Both perform component model integration through the transformation of the heterogeneous models to a common internal representation, based on a single meta-model. However, the common meta-model in GeneralStore is only used to store the models, while the integrated system model consists of the original models, together with wrapper elements generated based on the specified interface definitions. Cheops, on the other hand, integrates the transformed models into a complete system model, on which a common numerical analysis method can be used. With both approaches, the resulting complete model can be used for the co-simulation of the integrated components.

In dealing with *view integration*, the models generally need to be integrated at a finer level of detail, associating specific content within the models to each other.

In this survey, four integration approaches deal with view integration: Fujaba, IDM, IMPROVE and Toolnet. Different types of relations can be setup either manually or automatically between the models. As identified in Toolnet, two general categories of relations can be defined: *general dependencies* and *data duplication*. Once the relations are setup, the most common analysis support provided as part of the integration platforms is that of consistency checking of model data between the tools, as provided in Fujaba, IMPROVE and Toolnet. The approaches also provide mechanisms to repair any inconsistencies found during the analysis. In certain cases, the integrated models deal with the same or close aspects of the system being modelled. In other words, much duplicated or similar data is found in the heterogeneous models. In such cases, a transformation between the different model types can also be performed. Transformation facilities are provided by Fujaba, IDM and IMPROVE.

Very few platforms consider the issue of data management. In Eclipse, such support is gained through the integration of the CVS [48] versioning tool. Considering that Eclipse does not perform data integration, CVS is simply treated identically to any other development tool. Such integration is similar to that illustrated in figure 6. The management tool manages the documents at the coarse file level, without dealing directly with the fine-grained model data. From the studied platforms, GeneralStore is the only platform to provide management functionalities such as user authentication, transaction management and fine-grained object versioning. This approach is closer to that illustrated in figure 7, but not entirely satisfactory, since the need to integrate the platform with existing PDM/SCM systems remains.

The general trend in the implementation of the platforms focusing on data integration is to assume a centralised data storage system, to which tools are integrated through a wrapper or a plug-in. The wrapper provides the necessary abstraction from the tool-specific implementation and formats, and in this way providing a uniform interface to the platform. The storage system can be a database management system such as for GeneralStore, or a simple file as in IMPROVE.

With the exception of GeneralStore, the repository is not generally used to manage the complete set of model data from the tools. Instead, the platforms only handle reference objects to the model data and additional integration information such as relations between the references objects and relevant metadata. Model data is expected to be managed and stored by its producing integrated tool. The strongest motivation for not storing modelling data is to avoid the duplication of information in the modelling tools as well as platform. Such an approach however limits the possibility to provide the necessary management functionalities, as advocated in this thesis.

7. Industrial Case Studies

This section presents a summary of two industrial case studies carried out at Scania, as part of this research project. As briefly discussed in section 7.3, the case studies were used as a source of inspiration, as well as to evaluate some of the ideas presented in this thesis. The first case study aimed at a quantitative analysis of architecture designs based on a set of keyfigures that reflect important quality attributes. Given exposure to the challenges faced during this case study, a second case study was initiated to deal with an analysis of the function modelling capabilities at the organisation, together with a recommendation for future improvements. A more complete description of these case studies can be found in [49] and [50] respectively.

7.1. Keyfigure Analysis Case Study

During the early architectural design of a truck, architects face the challenge of choosing the Electrical/Electronics (EE) architecture, onto which the system functionality is to be implemented. It is desired to quantitatively analyse and compare different architecture designs, taking into consideration and optimising important design keyfigures such as the resulting system weight and costs. The evaluation needs to perform trade-offs between a set of keyfigures, taking into consideration a range of product variants.

For this end, a keyfigure tool supporting the architecture design of allocating functions to control units, as well as the quantitative calculation and weighting of selected keyfigures, was developed. The architecture of the developed keyfigure tool, together with its different data sources is shown in figure 11.

A central database was used to collect information about the functional specifications, communication signals and components set of the product variants. Data was collected from a range of dispersed sources in the organisation. A core source of information was the Function and Component Databases that needed to be manually manipulated to suite the needs of the study. Another important source of implementation data was the Communication Database used to deduce the communication needs between functions, the decomposition of functions into subfunctions, and their allocation to electronic units. In addition, specific product variants were imported from proprietary product identification files, in which variants were defined as a selection of a set of user functions.

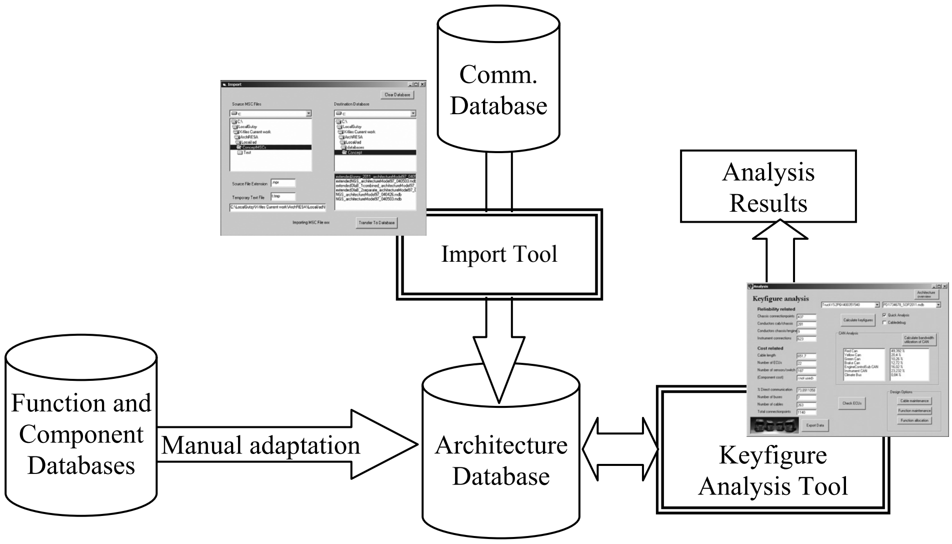


Figure 11. Tool architecture for the keyfigure calculation tool

A wide range of keyfigures (See table 3) was selected based on four important product aspects: Dependability, cost-efficiency, modularity and performance. An example keyfigure is the number of cable connection points. This keyfigure relates to the dependability aspect, since connections are an important source of faults and failures in embedded automotive control systems. The aim is to reduce the number of connection points in difficult environments, through the appropriate positioning of control units. The length of cables and number of components are other easily analyzable keyfigures relating to cost-efficiency.

In the study, the specification of the functionality and the hardware architecture were separated, creating two views of the system. The separation facilitated the possibility to perform multiple allocation strategies without needing to re-model the system functionality. The functionality was modelled as function blocks linked by communication links. The implementation was modelled as electronic units linked by cables. The electronic units include sensors, actuators and electronic control units (ECU). The different views are then interrelated once the functional allocation onto the hardware is defined, where function blocks are associated to electronic units and communication links are associated to one or more cables.

Table 3. The keyfigures considered in the quantitative architectural design analysis.

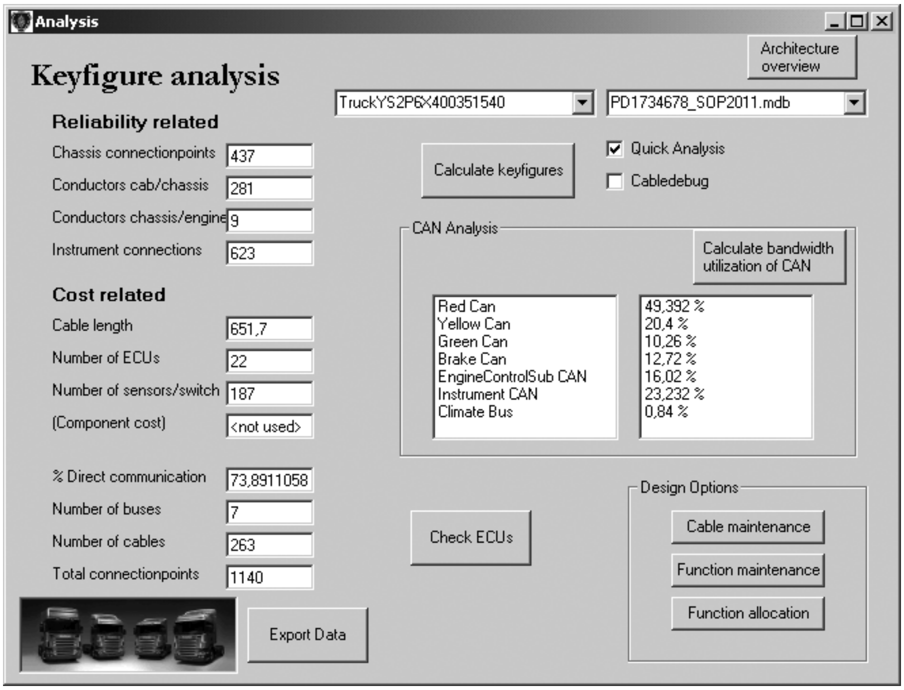
▪ Number of connection points ▪ Cable length ▪ Connections in bad environment ▪ Number of cables in difficult passages ▪ Number of ECUs ▪ Number of sensors ▪ Weight ▪ Number of units developed in-house	▪ Number of suppliers/sensors ▪ Modularity ▪ Number of messages through gateway ▪ Number of Mission critical units ▪ Processor utilization ▪ Gateway utilization ▪ Number of suppliers/ECU	▪ Number of mission critical connections ▪ Number of part numbers ▪ Number of distributed functions ▪ Number of widely distributed functions ▪ Number of pins/ECU ▪ Component cost ▪ Bandwidth utilization

Once the functional allocation is performed, an analysis tool allowed the keyfigure calculations for a specific product variant and system architecture. A screenshot of the main analysis window, highlighting some of the measured keyfigures, is provided in figure 12. Using this tool, it was possible to quickly compare alternative architectures and find the weaknesses and strengths of the alternatives as indicated by quantitative keyfigures.

7.2. Function Modelling To Improve Software Documentation

Among the many distributed sources of information within the Scania organisation, the current functional documentation of the EE (Electrical/Electronics) system is mainly based on three core documents:

- User Function Specification (UFS) - specifies a *User Function*, which is a specific functionality to be implemented in a vehicle, implemented over more than one system.
- System Description (SD) - specifies a *System*, describing the physical entities onto which User Functions are implemented such as sensors, actuators and ECU-hardware units.
- Message sequence charts (MSC) - Specifies a *Scenario* describing a specific sequence of events for a given User Function. Multiple scenarios are specified for each User Function and these are grouped into *Use Cases*.



Keyfigure analysis

TruckYS2P6X400351540

PD1734678_SOP2011.mdb

[✓ Quick Analysis]

Cabledebug

Reliability related

Chassis connectionpoints	437
Conductors cab/chassis	281
Conductors chassis/engine	9
Instrument connections	623

Cost related

Cable length	651,7
Number of ECUs	22
Number of sensors/switch	187
(Component cost)	<not used>
% Direct communication	73,8911058
Number of buses	7
Number of cables	263
Total connectionpoints	1140

CAN Analysis

Red Can	49,392 %
Yellow Can	20,4 %
Green Can	10,26 %
Brake Can	12,72 %
EngineControlSub CAN	16,02 %
Instrument CAN	23,232 %
Climate Bus	0,84 %

Design Options

Figure 12. Screenshot of architecture scorecard tool

In a preliminary internal study, a range of problems were identified with the current functional documentation, namely:

- **Document inconsistencies** - Text editors are used for the documentation, where references to other documents are hard-coded, with no mechanisms to update these links upon changes.
- **Incomplete information** – A scenario-based behaviour description of the functions is used, leading to an incomplete specification. In addition, functionality to be completely implemented within one hardware unit is not necessarily documented.
- **No user function overview** – No documentation currently provides a general overview of functions, focusing on the end-user aspects.
- **Unclear dependencies** - For a particular user function, the distribution of function parts onto systems is implicit.
- **Function and Implementation mixed-up** - The current User Function Specification document contains information about both function and implementation, limiting the possibility of function reuse given

implementation changes, as well as blurring the boundaries between the roles of the system owner and the function owner.

A brief investigation to deal with these problems was performed. The study resulted in an information model and a documentation approach to function specifications. The proposed techniques were evaluated through the specification of three functions of varying complexity.

7.2.1. Information Modelling

The proposed information model to handle the document contents is illustrated in Figure 13. The information model is broken down into different views that group entities together targeting particular aspects of the system. Roles were also identified to control access to the information model entities.

The three main views of the system are the *Functional view*, *Software view* and *Hardware view*. A common pattern exists between each of these views, specifically: (1) The hierarchical decomposition used within each view, for managing the size and complexity of the system description. This highlights that there exists no single dominating product structure, and each view describes the system from a specific perspective. (2) The definition of entity interface through which the entity interacts with its external environment.

- Function View - The main object in this view is the *Function*, with two sub-types: *PartFunction* and *Variable*. A PartFunction object designates certain functionality that given a certain input, produces a certain output. A Variable object designates a transportation link that manages certain data internally and provides access to this data to connected PartFunctions. A Function can be decomposed into a set of (sub-)Functions, forming a hierarchical product structure. The interface definition of a Function is defined by a set of *ports*, where a port acts as a placeholder for a subset of its object's externally accessible properties.
- Software View - Similar to the Function view, the main object in this view is the *SoftwarePart*, with two sub-types: *SoftwareComponent* and *Data*. A SoftwareComponent object designates a sourcecode module that given a certain input, produces a certain output. A Data object designates a data storage facility that manages certain data internally and provides access to this data to connected SoftwareComponents. A SoftwarePart can also be decomposed into a set of (sub-)SoftwareParts, forming a hierarchical product structure. The interface definition of a SoftwarePart is defined by a set of *SoftwarePorts*, where a SoftwarePort designates a certain internal data item that is externally accessible to other SoftwareParts.

- Hardware View - Similar to the Function view, the main object in this view is the *HardwarePart*, with two sub-types: *HardwareComponent* and *Cable*. A HardwareComponent object designates a physical block having geometrical dimensions and a position. A Cable object designates a single cable with a certain geometrical path. A HardwarePart can also be decomposed into a set of (sub-)HardwareParts, forming a hierarchical product structure. The interface definition of a HardwarePart is defined by a set of *pins*, where a Pin designates a spatial location at which the HardwarePart can be connected to other HardwareParts.

In addition, the *User Function* view is a special view targeting the product user, and hence focuses on structuring the product functionality from the user perspective. A complete system is described using a network of hierarchically decomposed Functions. However, from the user perspective, certain sets of Functions form a clear and valuable contribution that the user can relate to. Such a set is managed in the information model using the *UserFunction* object. Ignoring Function variants for the moment, a UserFunction is a grouping of Function objects, forming a fully defined specific functionality (just like the hierarchical composition of functions into PartFunctions). It is important to note that a Function object does not exclusively belong to a single UserFunction. Certain functionality, such a ‘speed sensing’, provides services that can be shared by many UserFunctions. Such functions are a good indication of the interaction and dependencies between user functionalities.

Finally, given the importance of product configurations, each of the above views is further described using a specific variant view: *FunctionVariant*, *SoftwareVariant* and *HardwareVariant* views, describing variants of functionalities, software realizations of functionality and the hardware platform in which the software realizations are allocated respectively. Again, a pattern can be found in representing these three variant needs, and in their relation to other objects in the information model.

- The FunctionVariant is used to represent variations for a particular user functionality. A UserFunction is a grouping of FunctionVariants that provide similar or competing functionality from which the user can choose. A FunctionVariant object is in turn a grouping of Function objects, forming a fully defined specific functionality. It is important to note that a Function object does not exclusively belong to a single FunctionVariant, since certain functionality can be a common part among the various variants of a given UserFunction.

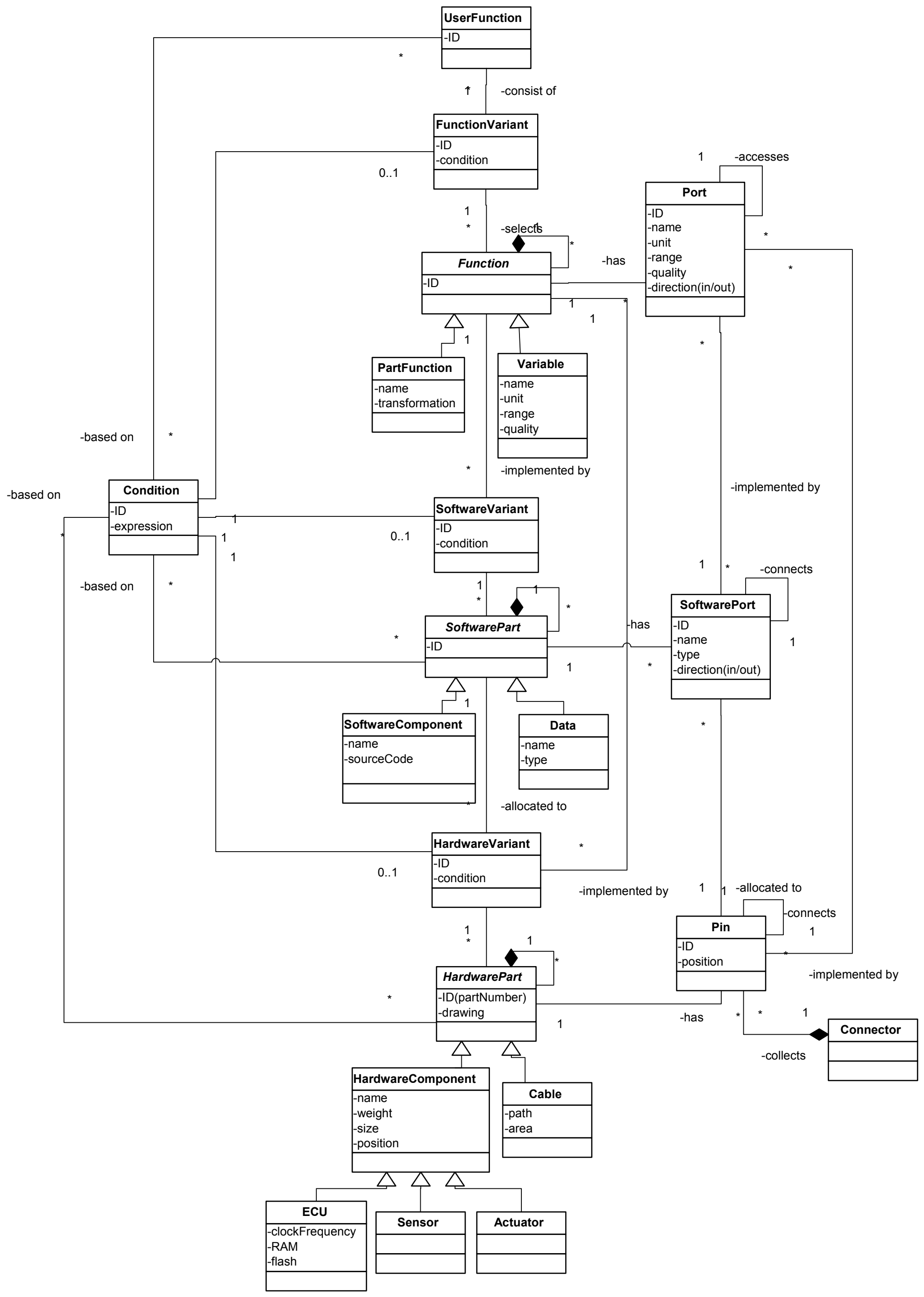


Figure 13. The proposed information model

- The SoftwareVariant is used to represent the different variants in how a particular Function is implemented in software. A SoftwareVariant is a grouping of SoftwarePart objects that together realise a given Function.
- The HardwareVariant is used to represent the different variants in how a particular SoftwarePart is allocated to hardware. A HardwareVariant is a grouping of HardwarePart objects that together implement a given SoftwarePart.

In the above views, objects do not exclusively belong to one view. For example, the SoftwarePart object belongs to both a Software view describing the software implementation, as well as a HardwareVariant view describing the allocation of software to hardware. Such objects help identify the dependencies that exists between views, calling for special attention for their management, in order to reduce duplication and inconsistencies in the product description.

7.2.2. Roles

As illustrated in table 4, certain roles responsible for the development of the views were identified. In most cases, the responsibility of defining the objects within a given view lies with the same role, and the table is hence presented relating views to roles. However, given that objects may not be exclusively defined within one view, it was necessary to relate the role responsibilities at a finer-grained level, relating roles to specific information objects. For brevity, the fine-grained responsibility sharing is not discussed here. In addition, besides the *Owner* roles, there exist several other roles that only need to access the product information, such as the system user, tester, safety analyst and maintenance/repair.

7.2.3. Proposed Documentation

The information model must be captured in some kind of descriptions, textual or graphical, collected in documents. Given the shortcomings of the original documentation, a new documentation solution is proposed replacing the original UFS and MSC documents. Two new documents are suggested instead: A User Function Description (UFD) document and a Function Architecture Description (FAD) document, specifying the implementation-independent functionality and their software/hardware implementation respectively. In the proposal, the SD document is also redefined to focus on the hardware aspects of the system it describes. The content of the new documents is simply a restructuring of the previous documentation, and major changes have been avoided where possible in order to permit a smoother shift to the new documentation structure. Since an analysis of potential tools and models were beyond the scope of the study, and recognising the effort needed in introducing new tools, documents are still defined using text editors. The use of UML 2.0 activity diagrams for describing functions is however proposed, given the present experience in its usage by some members of the organisation.

Table 4. The roles responsible for the development of the information model views.

View	Owner role	Role Description
Function	Function owner	Responsible for the specification, development and validation of a user function.
Software	System owner	Responsible for the development of a selected set of software/hardware components for the implementation of a selection of partFunctions/softwareParts.
Hardware	System owner	
Function variant	Configuration coordinator (functions)	Manages and ensures compatibility between the combinations of hardware and software for a given configuration. A configuration is a selection of systems with defined hardware and software versions. The configuration coordinator manages the conditions pointing out different variants.
Software variant	Configuration coordinator (software)	
Hardware variant	Configuration coordinator (hardware)	
User function	Function coordinator	Manages the interaction of user functions by coordinating the definition and development of partFunctions and their interactions.
F-SW allocation	Function coordinator	
F-HW allocation	Function coordinator	
SW-HW allocation	Communication coordinator	Manages the allocation of communication between software components both within and between processing units. The communication coordinator is responsible for reliable communication and non-congested channels.

7.3. Conclusion

The keyfigure analysis case study borrowed many ideas from the tool and mechanisms discussed in Paper-B. The multi-view principles presented in Paper-B were adopted in the restructuring and division of the available dataset into different views, thereby facilitating the desired analysis as well as the possibility to perform multiple allocation strategies without needing to remodel the system functionality. In addition, the database structure used in this case study is based on the meta-meta-model suggested in the paper.

Preliminary studies and keyfigure analysis of the case study were first performed using the prototype tool presented in Paper-B. However, a new keyfigure tool implementation was ultimately used to facilitate the process of importing information from the various sources at the organisation. In the final tool, the use of hierarchy within each view, and hence the cross-hierarchy allocation mechanisms, was not adopted. Nevertheless, the prototype tool later took advantage of the case study material for experimentation and testing purposes.

During the import of information from the various data sources, many inconsistencies in the documents were discovered due to duplication of information in the different documents and the lack of mechanisms to propagate changes between them. The needs for an integrated data management system as advocated in this thesis were confirmed from experiences in the case study.

The discovery of inconsistencies also triggered the documentation case study of section 7.2. The scope of the study did not encompass the implementation of tools for the automated management of the suggested documentation. For this reason, it was not possible, nor expected, to directly apply any of the solutions presented in this thesis. However, many ideas were borrowed such as the division of the information model into multiple views, as well as the particular meta-model within each view. Given the lack of automated support, integration was achieved through the restructuring of the documents to minimise the duplication of information and to highlight any relationships between their contents.

8. Future Work

As mentioned in section 4.3, this thesis focused on two cases of integration to cover each of the identified needs of view integration and model management. The potential for future developments is hence great.

The view integration mechanisms presented in Paper-B need to be expanded to cover other types of relationships. While specific to the allocation of system functions to hardware, it is believed that these mechanisms can be applied to other types of relationships such as that of mapping software components to hardware. However, no claim can be made that these mechanisms are general enough to handle all types of relationships. In particular, future work should address the management of duplicated information between tools, synchronizing and maintaining its consistency. A systematic approach when implementing these relationships should allow a reuse of many of the concepts already explored. In addition, the ability to perform inter-view associations over a larger number of views is a challenge to handle in future developments. Finally, a complete MDM-based implementation of the inter-view allocation approach remains to be developed.

A full validation of the PDM/SCM unification approach needs to investigate the feasibility of the remaining management functionalities. The functionalities of the union of typical SCM and PDM tools would include: Version management, product structure management, build management, change management, release management, workflow and process management, document management, concurrent development, configuration management and workspace management [15]. A unified approach should support the common needs of hardware and software development, as well as the discipline-specific needs such as build management for software development.

Relating to implementation issues, the current platform implementation investigates the potential of implementing the MDM platform using the technology offered by a commercial PDM system. This reference implementation can be used to highlight the shortcomings of conventional PDM, as well as the specific needs of MDM. The experience gained can then be used in the development of dedicated MDM systems.

The implementation of the current functionalities has not considered the performance issue yet, focusing instead on the feasibility of the approach in the large. It remains however to see if the expected performance can be provided by a conventional PDM, given that such a system is not normally designed to deal with a large number of fine-grained data items. Such an evaluation will provide valuable feedback on to the expected performance of new MDM solutions.

Finally, some process related and usability issues have been touched upon in this thesis, and are relevant for future work.

The inter-view mechanisms defined in Paper-B support a process-independent allocation practice. By placing certain restrictions, the allocation practices can be constrained. For example, disallowing the possibilities for association extensions through the sub-systems provides a top-down approach, where sub-system design can only refine design decisions specified at the higher level. The open approach however allows for the possibility to feedback information up the hierarchy. Exploring these process issues can be of interest for future extensions.

Doubt remains whether the inter-view mechanisms actually facilitate the developer's work. It is believed that the approach, while based on simple concepts, does require a new mind-set. From the limited gained experiences, the ability to focus on specific parts of the system design, as well as inheriting and extending other decisions made elsewhere in the system, is rewarding. This however does depend on good feedback and support by the integration tool. In the worst case, the approach advocated here can be seen as an experiment, or an initial step, towards other possibilities of view integration.

More advanced fine-grained version control algorithms need to be implemented in the platform. Future algorithms need to support concurrent development, by allowing parallel access to modelling elements, as well as providing branch/merge mechanisms. In addition, in supporting multiple product structures, support for the parallel development of these structures need to be provided, while ensuring the consistency of information across these structures. For usability reasons, the graphical visualisation of the differences between two model versions needs to be developed.

It would also be interesting to develop a number of version control algorithms based on the same MDM platform. The system can then be configured so that different strategies can be applied for different kinds of models. Different development needs can thus be satisfied using variants of the same basic mechanisms in a unified management system. For example, software development might require the complex version control mechanisms and concurrent development normally provided by SCM systems, while hardware development is satisfied with sequential revision control. The different solutions ought to be based on the same basic mechanisms, user interface and terminology.

9. Conclusion

Weinberg [3] states that ‘A system is a way of looking at the world… The system is a point of view – natural for a poet, yet terrifying for a scientist!’ System structuring is not an inherent property of the system. Instead, it is a way of looking at a system to better understand it.

In the shift from mechanical to multi-disciplinary mechatronics products, the need for multiple viewpoints becomes more evident. The need for multiple disciplines during development means that there will exist multiple viewpoints – multiple product structures. This is specifically amplified with software development within which the presence of many structures is more apparent.

For the successful integration of the efforts from each of these disciplines, the views need to be appropriately integrated, preventing any inconsistencies and divergences from creeping into the system design. Each view structure is equally important and the challenge is to integrate them appropriately.

An acceptable environment to perform view integration, should also deal with the various models used to represent these views. This leads to the need for model management functionalities and hence the challenge of integrating the management systems used by the specific disciplines, namely PDM and SCM systems. It is here argued that model integration ought to be one of the many functionalities supported by such an integrated, model-based, management system.

Recognising that such an environment ought to be a result of standardisation effort, this thesis focused on two cases of integration techniques to investigate each of the view integration and model management issues.

An approach to multi-view modelling and integration which tightly integrates the view hierarchies is presented. Specifically, model integration is investigated for the allocation of system functions onto the implementing hardware architecture. The proposed approach promotes the independent development of the views, allowing developers from each discipline to work concurrently, yet ensuring the completeness, correctness and analysis of any inter-view design decisions made.

A Model Data Management (MDM) platform that generically manages models from the various tools used in development is also presented. View integration is considered as an integral functionality of this platform. The platform is viewed as a unification of the management functionalities typically provided by the discipline-specific PDM and SCM systems. The unification is achieved by unifying the kind of objects it manages – models. The advantage of MDM over conventional PDM/SCM systems is the inclusion of the internal content of its supported models, allowing for a tighter integration of the design information between different models. In demonstrating the platform feasibility, a generic version management functionality of models is implemented.

The platform is argued to be feasible given the move towards model-based development in software engineering, bringing the discipline's needs closer to those of the hardware discipline. This leads the way for an easier and more effective integrated management platform satisfying the needs of both disciplines using a common set of mechanisms. The needs of the disciplines will always differ due to the nature of the products themselves. For example, the development process of software and hardware products differ [15]. However, in a unified management approach, the development needs of both disciplines can be satisfied, using variants of the same basic mechanisms, by providing different strategies for different kinds of models. It is essential however to base the strategies on the same basic mechanisms and user interface, allowing the reuse of basic components and preventing confusion in terminologies. While most critical for multi-disciplinary development, the platform is equally appropriate for the development of purely mechanical or software products.

The major aim of the current platform implementation was to experiment and illustrate the concepts discussed in this thesis. The architecture builds on existing technologies from each of the mechanical and software disciplines. The proposed MDM system is built based on a configurable PDM system, given its maturity, ability to manage model contents and the presence of already developed management functionalities such as the support for distributed development, change management, workflow control, etc. At the same time, the version control functionality borrows ideas from the fine-grained version control algorithms in the software discipline. The adoption of a PDM system is not indispensable and one can envisage building an independent MDM that supports both disciplines. It is our ideal vision that with the acceptance of model-based development, one no longer needs to discuss the integration of PDM and SCM systems. Instead, a truly unified approach to model data management can be used by both disciplines.

10. References

[1] Stuecka R., Bridging the Gap is not Enough – Life-cycle Management for Automotive Electronics and Software, Global Automotive Manufacturing and Technology, 2003.

[2] McKinsey & Company, Knowledge-based changes in the automotive value chain, HAWK-2015, 2003.

[3] Weinberg G. M., An Introduction to General Systems Thinking, Dorset House Publishing; Silver anniversary edition, ISBN 0932633498, 2001.

[4] Larses O., Factors influencing dependable modular architectures for automotive applications. Technical Report TRITA-MMK 2005:09 ISSN 1400-1179. Royal Institute of Technology, KTH, Stockholm, 2005.

[5] Checkland P., Systems thinking, Systems practice: Includes a 30-Year Retrospective. John Wiley & Sons, 1999.

[6] IEEE, ANSI/IEEE Standard 1471-2000, Recommended practice for architectural description of software-intensive systems, September 2000.

[7] Redell O., El-khoury J. and Törngren M., The AIDA toolset for design and implementation analysis of distributed real-time control systems, Microprocessors and Microsystems, Volume 28, Issue 4, 2004.

[8] El-Khoury J. and Törngren M., Towards a Toolset for Architectural Design of Distributed Real-Time Control Systems, 21st Real-Time Systems Symposium, 2001.

[9] Mathworks, Simulink, http://www.mathworks.com/products/simulink/, accessed January 2006.

[10] Redell O., Modelling of Distributed Real-Time Control Systems, An Approach for Design and Early Analysis, Licentiate Thesis, Department of Machine Design, KTH, TRITA-MMK 1998:9, ISSN 1400-1179, ISRN KTH/MMK--98/9--SE, 1998.

[11] Dome, Dome guide, Version 5.2.2, http://www.htc.honeywell.com/dome/index.htm, 1999, accessed January 2006.

[12] Redell O., Response Time Analysis for Implementation of Distributed Control Systems, Doctoral thesis, Dep. of Machine Design, KTH, TRITA-MMK 2003:17, ISSN 1400-1179, ISRN KTH/MMK--03/17--SE, 2003.

[13] Gomaa H., Software design methods for concurrent and realtime systems, Addison-Wesley publishing company, ISBN 0-201-52577-1, 1993.
[14] Eppinger S. and Salminen V., Patterns of Product Development Interactions, International Conference on Engineering Design, 2001.
[15] Crnkovic I., Asklund U. and Persson Dahlqvist A., Implementing and integrating product data management and software configuration management, Artech House Publishers, 2003.
[16] UML, OMG Unified Modelling Language Specification, V1.5, 2003.
[17] Larses O. and Adamsson N., Drivers for Model Based Development, Proceedings of the 8th International Design Conference on Design, 2004.
[18] Encyclopædia Britannica Premium Service, 'engineering', http://www.britannica.com/ebc/article-9363722, accessed January 2006.
[19] Westfechtel B. and Conradi R., Software Configuration Management and Engineering Data Management: Differences and Similarities, Proceedings 8th International Workshop on System Configuration Management, Springer-Verlag, pages 95-106, 1998.
[20] Bendix L. and Borracci L., Towards a Suite of Software Configuration Metrics, Twelfth International Software Configuration Management Workshop (SCM-12), 2005.
[21] MOF, "Meta Object Facility (MOF) specification", V1.4, April 2002.
[22] World Wide Web Consortium, Extensible Markup Language (XML) http://www.w3.org/XML, accessed January 2006.
[23] Object Management Group, Common Object Request Broker Architecture (CORBA), http://www.omg.org/technology/documents/formal/corba_2.htm, accessed January 2006.
[24] Microsoft, Component Object Model Technologies (COM), http://www.microsoft.com/com/default.mspx, accessed January 2006.
[25] Kemmerer S. J. (editor), STEP, the grand experience, National Institute of Standards and Technology, special publication 939, 1999.
[26] Dahlqvist, A.P., Crnkovic, I. and Asklund, U., Quality Improvements by Integrating Development Processes, 11th Asia-Pacific Software Engineering Conference, 2004.
[27] Cooling J., Software engineering for real-time systems. Pearson Education Limited, ISBN 0201596202, 2003.
[28] El-khoury J., Chen D. and Törngren M., A survey of modelling approaches for embedded computer control systems (Version 2.0), Technical report, ISRN/KTH/MMK/R-03/36-SE, TRITA-MMK 2003:36, ISSN 1400-1179, Department of Machine Design, KTH, 2003.
[29] Garlan D., Monroe R. and Wile D., ACME: An Architecture Description Interchange Language, Proceedings of the Centre for Advanced Studies on Collaborative Research (CASCON) Conference, 1997.

[30] Allen R. J., A Formal Approach to Software Architecture, Ph.D. Thesis, Carnegie Mellon University, Technical Report Number: CMU-CS-97-144, 1997.
[31] Shaw M., DeLine R., Klein D. V., Ross T. L., Young D. M. and Zelesnik G., Abstractions for Software Architecture and Tools to Support Them, IEEE Transactions on Software Engineering, pages 314-335, 1995.
[32] Luckham D. C. and Vera J., An Event-Based Architecture Definition Language, IEEE Transactions on Software Engineering, 1995.
[33] Halbwachs N., Synchronous programming of reactive systems: A tutorial and commented bibliography, Proceedings of the International Conference on Computer-Aided Verification (CAV), 1998.
[34] Harbour M. G., Gutiérrez J. J., Palencia J. C. and Moyano J. M. D., MAST: Modeling and Analysis Suit for Real-Time Applications, Proceedings of the Euromicro Conference on Real-Time Systems, 2001.
[35] Demmeler T., O'Rourke B. and Giusto P., Enabling Rapid Design Exploration through Virtual Integration and Simulation of Fault Tolerant Automotive Application, Society of automotive engineers, Document Number: 2002-01-0563, 2002.
[36] Simon D., Pissard-Gibollet R., Kapellos K. and Espiau B., Synchronous composition of discretized control actions: design, verification and implementation with Orccad, 6th International Conference on Real-Time Control Systems and Application, 1999.
[37] Henzinger T. A., Horowitz B. and Kirsch C.M., Giotto: A time-triggered language for embedded programming, In Proceedings of the First International Workshop on Embedded Software, 2001.
[38] Krueger J. W., Vestal S. and Lewis B., Fitting the pieces together: system/software analysis and code integration using MetaH, Proceedings of the 17th Digital Avionics Systems Conference, 1998.
[39] Liu X., Liu J., Eker J. and Lee E. A., Heterogeneous Modeling and Design of Control Systems, Software-Enabled Control: Information Technology for Dynamical Systems, 2003.
[40] Bræk R., SDL Basics, Computer Networks and ISDN Systems, volume 28, issue 12, special Issue: SDL and MSC, 1996.
[41] G. Schopfer, A. Yang and W.Marquardt, Tool-Integration in Chemical Process Modelling, 9th European software Engineering Conference and 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2003.
[42] Eclipse, The Eclipse Project, http://www.eclipse.org/, accessed January 2006.

[43] Burmester S., Giese H. (et al.), Tool integration at the meta-model level: the Fujaba approach, International Journal on Software Tools for Technology Transfer, volume 6, no. 3, 2004.

[44] Reichmann C., Kuhl M., Graf P. and Muller-Glaser K. D., GeneralStore - A CASE-Tool Integration Platform Enabling Model Level Coupling of Heterogeneous Designs for Embedded Electronic Systems, 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, 2004.

[45] Karsai G., Lang A. and Neema S., Design patterns for open tool integration, Software and Systems Modelling, Volume 4, Issue 2, 2004.

[46] Becker S. M., Haase T. and Westfechtel B., Model-based a-posteriori integration of engineering tools for incremental development process, Software and Systems Modelling, Volume 4, Issue 2, 2004.

[47] Freude R. and Königs A., Tool integration with consistency relations and their visualization, 9th European software Engineering Conference and 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2003.

[48] Ximbiot, CVS, http://ximbiot.com/cvs/, accessed January 2006.

[49] Larses O., Applying quantitative methods for architecture design of embedded automotive systems, Proceedings of INCOSE International Symposium, 2005.

[50] Larses O. and El-khoury J., Function Modelling to Improve Software Documentation. Technical report, ISRN/KTH/MMK/R-05/25-SE, TRITA-MMK 2005:25, ISSN 1400-1179, Department of Machine Design, KTH, 2005.

Paper-A

A Tool Integration Platform for Multi-Disciplinary Development

El-khoury Jad, Redell Ola and Törngren Martin

Published as
31st Euromicro Conference on Software Engineering and Advanced Applications, 2005.

Abstract

In multi-disciplinary development, where various domain specific tools are used by developers to specify and analyse a system, efficient system development requires that the models produced by these tools are well integrated into a whole, reducing any risks of inconsistencies and conflicts in the design information specified. In this paper we present an architecture for a model and tool integration platform that borrows its major components from well known and accepted standards from both computer and mechanical engineering. The architecture supports model integration, where models defined in different tools for different aspects of the same system are related such that they may share and exchange data.

The integration platform also enables model management functionalities on a fine-grained level, suggesting a combination of the functionalities found in traditional data management systems such as product data management (PDM) and software configuration management (SCM).

A.1. Introduction

Mechatronics systems development requires the close collaboration of various specialist teams and engineering disciplines in order to reach the expected complex functionality. In automotive system design for example, traditional engineering disciplines such as control, software, mechanical and computer engineering, need to interact to meet the demands for dependable and cost-efficient integrated systems. Even though working with the same system towards the same overall goal, developers from different domains use specific tools, providing their own specific views of the system to be developed. Each system view targets a specific audience, using that audience's familiar language (viewpoint), and concentrating on that audience's concerns [6]. Figure 14 illustrates some of the viewpoints and views that may be necessary during the development of a typical vehicular system.

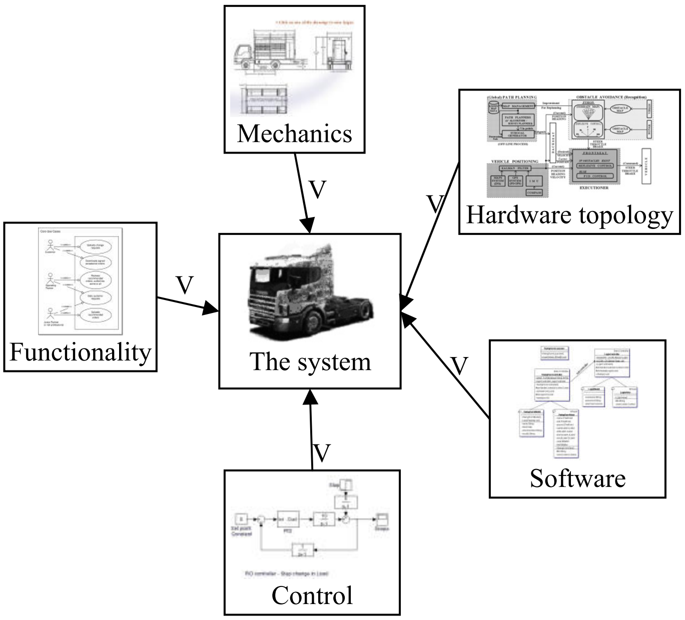


Figure 14. An example of some disciplines and views in system design.

For successful system development, the views of all developers produced by the different tools should be well integrated into a whole, reducing any risks of inconsistencies and conflicts in the design information described in these views.

View integration can be performed either through social communication among developers - social design, or through formalised and automated design processes -

model based design (MBD) [2]. MBD refers to a development approach whose activities emphasise the use of models, tools and analysis techniques for the documentation, communication and analysis of decisions taken at each stage of the development lifecycle.

This paper proposes an architecture for a model-based tool integration platform that allows for the generic management of different kinds of models from a set of different tools, as well as the automated sharing of data between these models produced during multi-disciplinary development.

In the next section, we categorise and discuss the needs for tool and model integration. Section A.3 presents the architecture for the Model Data Management (MDM) platform that aims to satisfy these needs. The major functionalities provided are further discussed in section A.4, followed by a presentation of the tool implementation. Finally, a discussion of related work is presented before concluding the paper in section A.7.

A.2. Needs for Tool and Model Integration

With the increasing acceptance of model-based engineering, a large number of tools are available that support specific aspects of the development process. While it is desired to obtain a single tool that can fully support system development processes, experience shows that no such silver bullet can be provided.

Our conviction is that no matter how many and large modelling languages get, we will never reach the point when a single language, and consequently a single tool, will meet all the needs of a development process in any given company. As a consequence, the need to integrate model information and the tools that act on this information will always exist. A platform that supports this type of integration should meet a number of needs in model based engineering. The more important of these are pointed out in the following.

Support for domain specific tools and languages. The presence of a variety of tools within a company arises from multiple reasons. First, developers from the different disciplines are accustomed to and educated in specific languages used in domain-specific tool environments that let them focus on specific aspects of the system. Second, it is not uncommon that, even within a given discipline, many tools that provide almost the same functionality are used within the same organization. While this redundancy may seem unnecessary and could be avoided in many situations, it is nevertheless a common practice when no single tool can offer all needed functionality.

Apart from design tools in which models are specified, another group of tools use the model data either for transformation or analysis purposes. Many such tools are integrated into design tools or are plug-ins acting on information from many different data sources.

Data sharing and view integration. There are two main reasons to use an integration platform to handle model data from different tools. First, it is necessary when certain system information is used and duplicated in more than one tool. A tool integration mechanism should manage the duplication of information in the tools, synchronizing and maintaining its consistency. Second, having chosen a specific set of tools, certain design information ends up in between tools. This information specifies a relationship between the different views (inter-view information). For example, the allocation of software components onto the hardware components of a system is not the sole concern of either the software or the hardware developer, and this design decision lies between the two views. Good integration mechanisms should permit the specifications of such cross-view information and reflect the interaction points at which the respective stakeholders need to communicate.

Model management. A further important aspect of model based development is the need to manage all the models produced during development. In the development of large and complex products, an organization normally adopts some kind of product management tool to support its development process, and deal with the large amount of design information, created and modified during the development and product life cycle. Model management includes supporting functionalities for the storage of models, handling of versions and variants of models, access control, change request management, process/workflow management as well as support for geographically distributed development. Only a few tools provide model management facilities, and for this reason an integration platform ought to provide this functionality centrally for all tools that it integrates.

A.3. MDM Architecture

Accepting that no single tool-suite would be sufficient in the development of mechatronic systems, and that tools need to be coordinated in order to tightly couple the model data produced and managed within them, we seek to define a generic architecture for tool management and integration. This architecture should support the needs identified in section A.2

No tool in the tool-set should take a predominant role, to which all other tools integrate. Such an approach creates a dependency on that tool, and peripheral tools can only be integrated indirectly. Instead, the solution proposed here makes use of a central platform to which each tool is connected. It is through this platform that communication between tools occurs. The envisaged tool integration architecture is shown in figure 15.

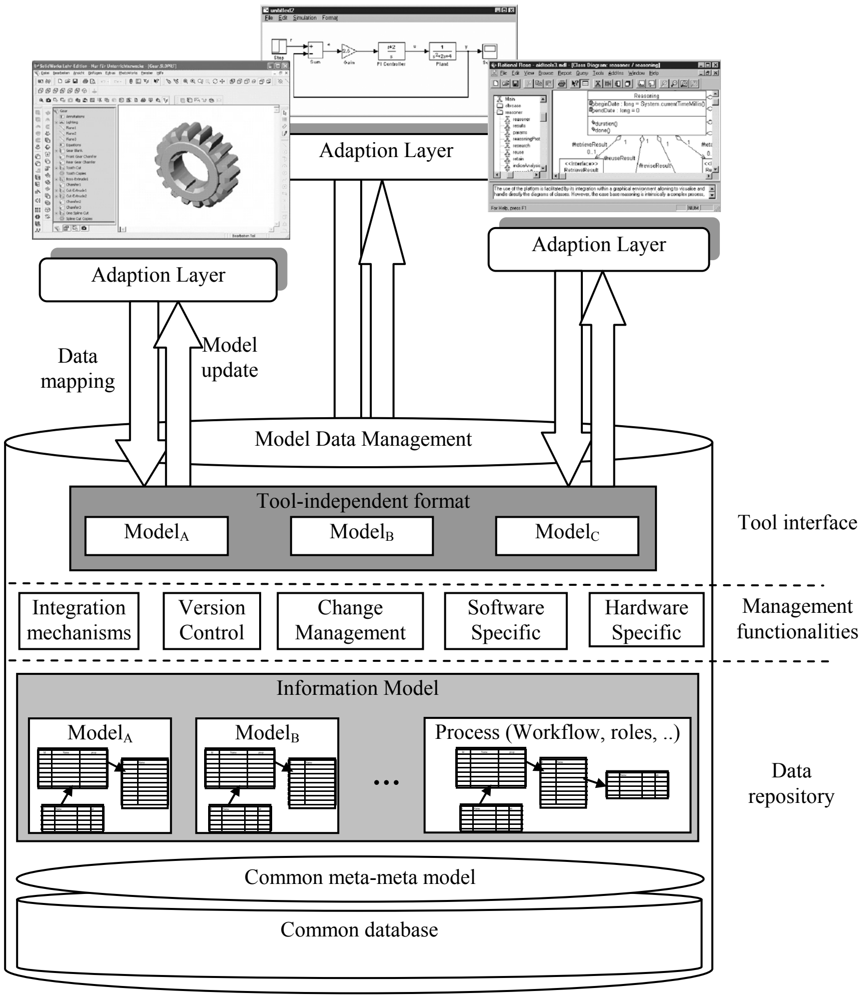


Figure 15. Tool integration architecture

The platform consists of two main parts: A set of tool-specific adaption layers and a data repository with mechanisms to handle this data. The data repository stores the data for each of the tools. To perform this role in a generic way, the data from the different tools is expected to be presented in a predefined form, and this functionality is provided by the adaption layer. Triggered either by a tool or the repository, the corresponding adaption layer permits the data flow between a tool and the repository. The following subsections will further discuss these components.

A.3.1. Data Repository

The data repository stores the data from each of the tools integrated into the platform. In addition, it provides mechanisms to manipulate this data such as the sharing of relevant data from one tool to other tools and tracing changes over time.

For the concern of model integration, it suffices to handle tool data that may be of interest to other tools, and a complete coverage of all information specified in a tool is not necessary. However, other functionalities, such as version control, place a more demanding requirement that it should be possible to completely re-generate a tool-specific model from the data in the repository, with no resort to additional external files or databases. For this use, the data repository needs to store all tool data.

In order to satisfy both needs, the platform needs to manage the complete data set ever needed to fully reproduce and control a model. However, this data is divided into two subspaces with differing access and formatting properties. The *public space* manages a subset of the complete data set that can be accessed by other tools in the toolset. The remaining data is stored in the *private space*. The public space is further divided into *read-only* and *read-write* subspaces. This division of subspaces allows flexible control over data access privileges in the platform. Using a common classification as an example, tool data can be separated into graphical and model data [12]. In most cases, the former will belong to the private space, while the later belongs to the public space.

Furthermore, while private data can be stored in a proprietary format that is only understood by the corresponding tool, the public data needs to be represented using a generic format and structure in the repository, understandable by all adaption layers in all connected tools.

Access to the tool data and the mapping of this data to the repository is performed by an adaption layer as discussed in section A.3.3. In the next section, we discuss the information model to be adopted in order to manage and structure the data in the repository.

Note that the data repository in this solution is not far from that adopted when using a Product Data Management (PDM) system in the development of mechanical systems. We here seek to perform a similar approach for the model-based design of multi-disciplinary systems, which may need to integrate data from a broader class of models from a multitude of disciplines of software and mechanical engineering.

A.3.2. Meta-modelling

A tool generally adopts a specific internal meta-model that it uses as a basis for the data schema to internally manage and store the model data produced. In many tools such as in Simulink [14], a meta-model is implicitly assumed, while others, such as UML [15] tools, are strongly based on a given meta-modelling framework.

The MDM platform managing an integrated model-set needs to map the meta-model of each tool onto that of the repository. In order to simplify the specification of a schema for each integrated model, a meta-meta-model is adopted as a basis for the repository. In developing an adaption layer, the meta-meta-model is instantiated to reflect a given meta-model, which is then further instantiated when storing the internal model data of its tool in the repository.

Note that adopting a common meta-meta-model between the models is not sufficient if there is a need to integrate the various model contents into a whole. It may well be the case that each model type occupies a separate space in the repository with a different data structure. To obtain an integrated information model, a unified information model of the set of models is necessary, specifying more detailed semantics of the models and their interrelations. While such information models are standardised for hardware products [6], no such standard model is currently available that also encompasses models from the software discipline. In section A.4.1, we explore the possibility to setup relations between the different models, integrating the models to form a complete information model.

We adopt a simple meta-meta-model which generalises among established meta-meta-modelling languages such as MoF [16], Dome [4] and GME [5], and based on a broad survey of modelling languages for embedded computer systems [3]. A model can be generally viewed as consisting of a hierarchical structuring of modelling objects that may possess properties; ports defining interfaces of these objects; and relationships (such as associations, inheritance and refinement) between ports. Modelling languages differ in the kinds of objects that can be specified, their relationships and the kind of properties they possess. When integrating a particular model, the adaption layer maps the model data onto the repository according to the meta-model specification defining the kind of objects, ports and relations that may exist.

In this approach, the granularity at which the MDM system operates on the models is controlled by the definition of the meta-model, implemented in the adaption layer. Mechanisms will understand the model semantics down to the level at which the elements, ports and properties are defined. Finer semantics within these entities are not the concern of MDM. For example, if a property of an element is defined as a blob of text, an MDM functionality cannot be expected to interpret the detailed semantics of this property.

A.3.3. Adaption Layer

Access to the tool data and the mapping of this data to the repository is performed by the adaption layer. An adaption layer is developed for each tool to be integrated into the MDM system. This layer isolates the tool-specific issues allowing MDM to operate generically on many tools implementing different technologies. The adaption layer fulfils three purposes. As discussed in section A.3.2, it maps the specific meta-model of its designated tool onto the repository's. The adaption layer also specifies how the model data is to be partitioned between the public and private data space in the repository.

Second, the adaption layer translates the specific format used by its designated tool to a generic format adopted in the platform. Different technologies are available for a tool to internally store its model data. A tool can use either a computer file system to store model data in a file, or a database management system. Various standards exist that specify how data should be handled using these technologies, yet one cannot assume that tools will not implement their own solutions. In a set of tools in which the tools adopt a combination of technologies (standard or not), it becomes necessary to translate these technologies onto a common format. This makes the tool-specific data technology transparent, and provides a generic interface to the rest of the platform functionalities. In the platform advocated in this paper, we adopt the data neutral XML format.

Third, the adaption layer accommodates the different techniques used to gain access to the tool's internal data. Different tools use different technologies to provide automated access to its internal data. In the simplest case, the adaption layer can access and interpret the textual file produced by the tool. A tool can also provide 'export' functionality, an Application Programming Interface (API), or a query language.

For a potential tool to be integrated into the MDM system, specific automation support is expected. In order to allow fine-grained accessibility to parts of models and the manipulation of models, a modelling tool whose models are to be managed need to:

- Provide access to the model data either through an API or using parsable text [6].
- Provide fine-grained mechanisms for the construction and modification of models through an API.

The translation executed by the adaption layer is triggered upon request by the user, whenever the model being developed is deemed satisfactory. The adaptor performs a transformation of the current state of the tool's internal data to an XML file format. This file is then received and interpreted by the repository in order to store the information in its store.

Naturally, the mapping is performed several times during development as changes to the model occur. For traceability reasons, the repository implements a version control mechanism as discussed in section A.4. In this mechanism, the repository needs to identify the changes performed since the last translation, and modify its information space accordingly.

A.4. Model-based Data Management Functionalities

In developing software-intensive products, an organisation generally adopts a Software Configuration Management (SCM) system to manage the large amount of files produced in the process. Similarly, mechanical system development relies on Product Data Management (PDM) systems.

In multi-disciplinary development, a number of such systems may need to simultaneously exist and considering the central role these tools take in controlling the development process, any tool integration effort needs to consider the integration of this class of tools. In this section, we will discuss how the MDM integration platform aims to provide the functionalities of these tools.

First, we will study the reasons why such a common management system has not been possible in the past. Two fundamental problems can be identified that has so far complicated the integration of PDM and SCM tools. First, the development process support expected of these tools from their respective disciplines differs greatly. While mechanical design expects full life-cycle product support together with control over the process itself, software design has traditionally only expected management of large amount of source files produced during the implementation stages. Another major difference has been in the kind of information that the support tools are expected to handle. In mechanical development, the need to provide a seamless workflow from design to manufacturing phases has forced PDM systems to not only handle the documents produced, but much of their contents as well. An information model of the product data is an integral part of a PDM system, providing the facility to related information to each other, within as well as across the development phases. Software development, on the other hand, has so far satisfied itself with a file-based approach, were the only relations handled between the files are those of the file system itself. The internal structure of these files and the semantic relationships between them are outside the scope of SCM tools [11].

However, with the increasing maturity of the software discipline, SCM is expected to manage the early development phases by managing documents such as design and analysis models [13]. This need leads to the need to also manage the development process itself since distinctions need to be made between these different kinds of documents. In addition, with the move towards a more model-based software design approach, where models, and not just files, are analysed and transformed when moving through the development phases, it is desirable that SCM systems also handle the internal structure of the files under its control, as well as the relationships between these structures. This change is newly recognised in the literature and is generally referred to as fine-grained SCM ([13] and [19]). In summary, as software development becomes more model-based, its needs move closer to those of mechanical development, making it easier to unify and support the needs of both disciplines using a more common management tool.

In model-based development, models and not files become the focus of engineering activities. For this reason, data management functionalities such as version management, product structure management, workflow and process management should focus on the models and their contents, transparent to the file structure used to store them. The ability of the MDM system platform to handle the internal contents of models forms a good basis to provide such support.

This approach also allows for the alignment of variations in behaviour of common functionalities within PDM and SCM. For example, the fine-grained version control facilities provided by SCM are more desired in a model-based development environment than the conventional facilities provided by PDM tools, which simply perform a copying of files. In this case, adopting the former approach would be beneficial for both disciplines.

A.4.1. Multi-view Integration

The management functionalities discussed so far act generically on all model types independent of the other models integrated into the platform. The multi-view integration functionality presented here differs since knowledge of the other models to be integrated is needed.

In [3], different kinds of relations that exist between modelling elements within a given modelling approach were identified such as dependency, allocation and refinement relations. Depending on the set of modelling tools adopted within an organisation, certain relations may not be directly supported by any specific tool within the tool-set and lie in between the models (or views) of the system. Such inter-view relations are however critical since they specify important design information that is otherwise implicitly assumed during development. In addition, inter-view relations act as integration points between the various models of the system, providing a complete and consistent information model. For a truly model-based development, the integration platform should provide generic mechanisms to specify such inter-view information, in the case where no explicit tool is used otherwise.

In this section, we will explore one such example of model integration through inter-view relations. This example originated from a case study at Scania in which the functionality to be implemented in a truck is to be mapped onto the Electrical/Electronics (EE) physical architecture, during the early architectural design stages, optimizing keyfigure values such as system weight and cost [17]. In this case, while the system's function and hardware structures are modelled in specific separate tools, allocation was performed by yet another tool, with no explicit synchronization of data with its data sources, leading to inconsistencies. Good model integration mechanisms were needed to permit the specification of cross-view function allocation information using the original tools as data sources, removing any source of data duplications.

Figure 16 illustrates example models describing a subset of the functional and hardware structure of a truck. As argued in [18], it is desired to allow allocations to be made on an arbitrary level in the hierarchies, so that, similar to when working within a given view, when performing design decisions across views, designers can focus on specific parts of the system and at a certain level of abstraction. For example, a designer may wish to specify that the brake system is to be implemented on a certain group of processors, without needing to specify in detail which specific brake sub-functions is to be allocated to which processor. Such a decision can be further refined or extended by others or at a later stage.

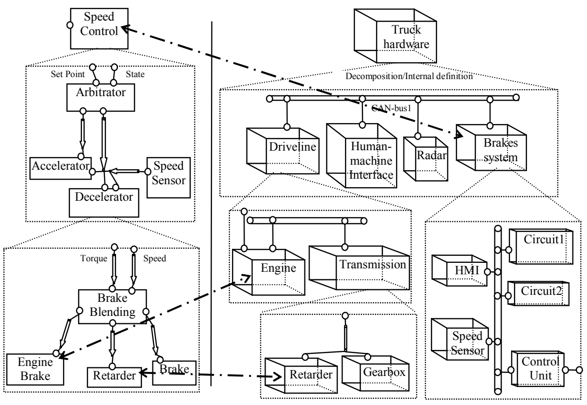


Figure 16. An example function structure model (left) and hardware structure model (right), of parts of a hypothetical truck EE architecture, together with the mapping (broken arrows) of functions onto hardware units.

This allocation strategy has the advantage of ensuring the independent development of each of the hierarchies, since hierarchical decomposition is a technique used to reduce the complexity perceived by a given stakeholder, and the use of this tool should not be compromised by the needs of other stakeholders. Further details on the cross-hierarchy allocation strategies and rules are presented in [18].

We use the example here to present the techniques of specifying inter-view information in the integration platform. Figure 17 shows the architecture in figure 15 in more detail, highlighting the allocation design data as inter-view data between two models. Note that in this case, the inter-view design data actually relates the structural aspects (functions and components) of two separate tools, and takes no consideration of their detailed properties.

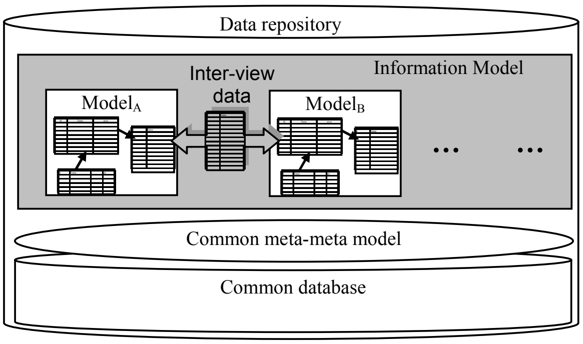


Figure 17. Detailed tool integration architecture, illustrating cross-tool data integration.

A.5. Tool Implementation

The MDM platform is currently based on a configurable PDM system, namely the Matrix PDM system [20]. The major advantage of using a PDM system is the possibility to define information models, with a high level query language to access and modify the model data in the repository. In addition, it is envisaged that the development of the MDM functionalities discussed in section A.4 is made easier given the already developed functionalities of PDM such as the support for distributed development, change management, workflow control, etc.

A simple model version control functionality (MVC) has been implemented. The algorithm supports the versioning of any model that can be mapped to the meta-meta-model assumed in the platform. In the current implementation, data flow diagram (DFD) models from the Matlab/Simulink [14] tool and Hardware

Structure Diagram models [21] in the Dome [4] tool, are handled. MVC provides mechanisms that allow a user to save and extract any part of the system model. In a ‘checkin’ operation, changes to the model since the last checkin operation are saved in the repository. When performing a ‘checkout’ operation, the specified element is reconstructed for a given version, together with its subparts, forming an XML document of the information in the repository. This document is then further transformed by the adaption layer to create a tool-specific format that can be used by the tool. The details of these operations are performed transparently to the user, allowing him/her to interface with the modelling tool’s interface and format.

A preliminary multi-view integration mechanism is also currently implemented for the case-study presented in section A.4.1. In the current implementation, the user is given direct access to the interface and data between Simulink and Dome models. This access is provided through generic mechanisms in the repository and the adaption layers of the two tools assuming the tools provide the necessary APIs. Another possible solution is to perform the allocation specification using a generic tool that forms an abstract presentation of the data in the repository of each of the tools, and through which the user specifies the relationships between the data items as desired. Such a generic tool will form part of the integration platform, and is reused in setting-up any kind of inter-view relationships. This tool can for example be built using one of the generic rapid-prototyping tool environments such as Dome [4] and GME [5].

A.6. Related Work

With the increasing automation support needs in product development and the variety of tools available, the need for tool integration is increasing.

Integration platforms such as Toolnet [7] and Fujaba [8] share similar aims to those presented in this paper. These solutions rightly advocate a limited repository that only stores the additional integration information not otherwise stored in the tools being integrated. Such an approach however limits the support functionalities that can be provided, specifically the data management functionalities generally expected of a PDM/SCM system.

The transformation-based approach to model integration as advocated by MDA [9] is also an important factor in tool integration. Focus in solutions such as [10] is placed on the integration of models in which a large amount of data is duplicated and where one model can be automatically deduced from another. However, this solution is of less help in integrating different types of models, where it is necessary to specify inter-view relations (section A.4.1) coupling data items across the modelling tools.

As shown in this paper, the integration of PDM and SCM systems should be a critical part in any tool integration effort for the development of mechatronics systems. In [11], three different techniques of integrating PDM and SCM systems are proposed. However, the suggested approaches accept the status quo of the file-based software development leading to limited integration success. Rejecting the status quo and focusing on the commonality between the disciplines (model-based development), as advocated in this paper should instead lead to a smoother integration.

A.7. Conclusion

In multi-disciplinary development, where various tools are intensely used by developers to specify and analyse the system, successful system development requires that the models produced by these tools are well integrated into a whole, reducing any risks of inconsistencies and conflicts in the design information specified. In addition, it becomes increasingly critical to provide generic functionalities to manage the models produced from the various developers.

In this paper, we proposed a model-based tool management and integration platform (MDM) that allows for the generic management of different kinds of models from a set of different tools, as well as the automated sharing of data between these models produced during multi-disciplinary development.

In essence, this approach attempts to borrow the best from each of the discipline-specific PDM and SCM technologies. We propose adopting a PDM system commonly used in the development of mechanical systems, and extending its usage for the model-based development of mechatronics products. The ability to manage workflow control and the specification of a product information model generally available in PDM systems is combined with the more advanced fine-grained version control mechanisms proposed for modern SCM systems. This should also help in the process of unifying the terminology used within an organisation.

A model-based approach to system development suggests that a more fine-grained handling of models is supported, in which the set of model data specified in each of the tools is managed by the integration platform based on a common information model and not simply as a set of files. This fine-grained approach then allows various coupling between data from the different tools to be performed.

A good model-based, integrating design environment is also a good basis for the communication of information between developers, where any conflicts and misunderstandings between developers are reflected, detected and dealt with in the models. An integrated platform allows design decisions taken by one developer to be communicated to the rest of the team in an understandable way.

To illustrate the MDM approach, an initial prototype tool has been developed on top of a PDM system. Functionalities currently supported include model version control (MVC) that allows the fine-grained version management of two types of models from two different tools. MVC permits stakeholders to perform design activities in terms of models, where they can organise, share and modify their models, transparent to the underlying file structure. In addition, the integration of two models from two different tools is studied facilitating the allocation of system functionalities onto the system hardware architecture.

A.8. References

[1] IEEE, ANSI/IEEE Standard 1471-2000, “Recommended practice for architectural description of software-intensive systems”, September 2000.

[2] O. Larses and N. Adamsson, “Drivers for Model Based Development” Proceedings of the 8th International Design Conference on Design 2004, Dubrovnik, May 2004.

[3] J. El-khoury, D. Chen and M. Törngren, “A survey of modelling approaches for embedded computer control systems (Version 2.0)” Technical report, ISRN/KTH/MMK/R-03/11-SE, TRITA-MMK 2003:36, ISSN 1400-1179, Department of Machine Design, KTH, 2003.

[4] Dome, “Dome Guide” Version 5.2.2, http://www.htc.honeywell.com/dome/index.htm, 1999.

[5] GME, “A Generic Modeling Environment, GME 4 User’s Manual” Version 4.0, Institute for Software Integrated Systems, Vanderbilt University, 2004.

[6] S. J. Kemmerer (editor), “STEP, the grand experience”, National Institute of Standards and Technology, special publication 939, 1999.

[7] R. Freude and A. Königs, “Tool integration with consistency relations and their visualisation”, ESEC/FSE workshop on tool-integration in system development, 2003.

[8] S. Burmester, H. Giese, et al., “Tool integration at the meta-model level: the Fujaba approach”, International journal on software tools for technology transfer, Springer, vol. 6, no. 3, pp. 203-218, 2004.

[9] OMG, “Model Driven Architecture Specification”, MDA Guide Version 1.0.1, Document Number: omg/2003-06-01, June 2003.

[10] G. Karsai, A. Lang and S. Neema, “Design patterns for open tool integration”, Software and Systems Modeling, Springer-Verlag, Volume 4, Number 2, 2004.

[11] I. Crnkovic, U. Asklund and A. Persson Dahlqvist, Implementing and integrating product data management and software configuration management, Artech House Publishers, ISBN 1-58053-498-8, 2003.
[12] D. Ohst, M. Welle and U. Kelter, “Differences between Versions of UML Diagrams”, Proceedings of the joint European software engineering conference (ESEC) and SIGSOFT symposium on the foundations of software engineering (FSE-11), 2003.
[13] D. Ohst, U. Kelter, “A fine-grained version and configuration model in analysis and design”, Proceedings of the international conference on software maintenance, October 2002.
[14] Simulink, Mathworks, http://www.mathworks.com/products/simulink/, accessed March 2005.
[15] OMG, “Unified Modeling Language (UML) Specification”, V1.5, March 2003.
[16] MOF, “Meta Object Facility (MOF) specification”, V1.4, April 2002.
[17] O. Larses and J. El-khoury, "Multidisciplinary modeling and tool support for EE architecture design." Proceedings of the 30th World Automotive Congress (FISITA 2004), Barcelona, Spain, May 2004.
[18] J. El-khoury, O. Redell and M. Törngren, ”Integrating views in a multi-view modelling environment”, To appear in the proceedings of the 15th international symposium of the systems engineering conference (INCOSE), 2005.
[19] S. Chien, V. J. Tsotras and C. Zaniolo, “Version Management of XML Documents”, Third International Workshop on the Web and Databases, 2000.
[20] MatrixOne, Matrix10, http://www.matrixone.com/, accessed April 2005.
[21] O. Redell, J. El-khoury and M. Törngren, “The AIDA toolset for design and implementation analysis of distributed real-time control systems”, Microprocessors and Microsystems. Vol-ume 28, Issue 4, 2004.

Paper-B

Towards a Multi-View Modelling Environment for Mechatronics Systems

El-khoury Jad and Redell Ola

Published as
Technical report, ISRN/KTH/MMK/R-05/24-SE, TRITA-MMK 2005:24, ISSN 1400-1179, Department of Machine Design, KTH, 2005.

A summary of this report is also published as:

El-khoury J., Redell O. and Törngren M., Integrating views in a multi-view modelling environment, Proceedings of the 15th International Symposium of the Systems Engineering Conference, 2005.

Abstract

The development of modern technical systems requires the close collaboration of various specialist teams and engineering disciplines. Even though working with the same system towards the same goal, developers from the different domains use their own specific tools, providing their own specific views of the system to be developed. For the successful integration of the efforts from each of these disciplines, the different views need to be appropriately integrated, preventing any inconsistencies and divergences from creeping into the system design.

In this report, we present an approach to multi-view modelling which systematically integrates the two generally accepted complexity reduction techniques of hierarchical decomposition and multi-viewing. While these techniques are common practice in many modern design tools, the approach presented defines how the inter-view relationships can be used to tightly interweave the views' hierarchies.

Through the use of a case study, model integration is investigated for the allocation of system functions onto the implementing hardware architecture. The resulting approach maintains the principle of hierarchical design within, as well as between the views, where allocation can be performed at arbitrary levels across the hardware and function hierarchies. The proposed approach promotes the independent development of the views, allowing developers from each discipline to work concurrently, yet providing support for a holistic view. This provides a good basis for an information sharing environment enabling model-based, multi-disciplinary development.

While specific to the allocation of system functions to hardware, these mechanisms can be reused for the mapping of system functionality to the software architecture, or software to hardware allocation. The generalisation of this work to cover other kinds of relations remains a challenge for future work.

B.1. Introduction

The development of modern technical systems requires the close collaboration of various specialist teams and engineering disciplines. In automotive system design for example, developers from the traditional engineering disciplines such as control, software, mechanical and electrical engineering, need to interact to meet the demands for dependable and cost-efficient integrated systems. Even though working with the same system towards the same goal, developers from the different domains use their own specific tools, providing their own specific views of the system to be developed. Each system view targets a specific audience, using that audience's familiar language (viewpoint), and concentrating on that audience's concerns [1]. Figure 18 illustrates some of the viewpoints and views that may be necessary during the development of a typical vehicular system. This separation of concerns has been well recognised in literature and is the common practice of modern engineering modelling languages and tools ([2], [3], [4] and [5]).

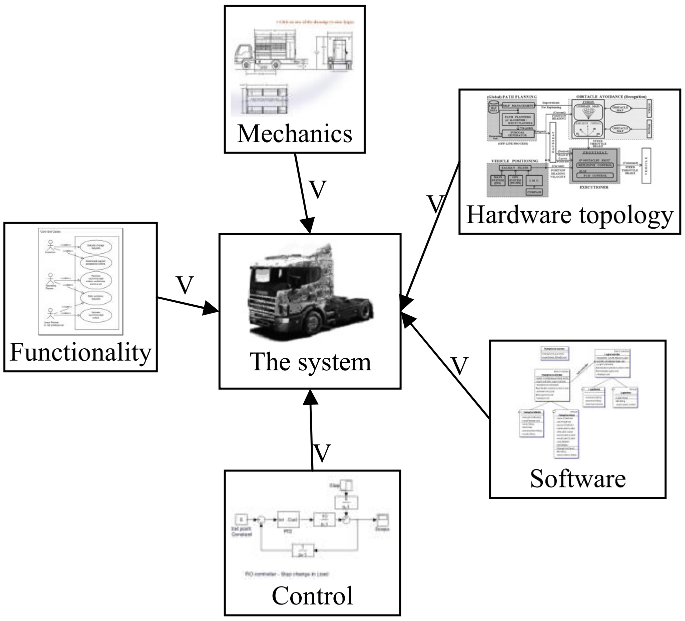


Figure 18. Some of the disciplines and views in system development.

Breaking up the design information of the system into multiple views, based on domain concerns, has the major advantages that it increases understandability and reduces the perceived complexity of the system at hand. However, the concerns and interests of each domain are not necessarily exclusive, which leads to overlap and dependencies in their development information space. In addition, even though they attempt to develop the same system, developers from the different disciplines may form a different perception of the system's aims, problems and solutions. Combined with the fact that these disciplines are distributed across several teams that focus on specific subsystems of a large system, it becomes essential that the efforts of all developers are well communicated and the different views are well integrated into a whole. This reduces any risks of inconsistencies and conflicts between the views.

There are two main reasons for the need of view integration. (1) Integration is necessary in the case where it is not desired to specify certain system information exclusively within a single view, since the information is the concern of more than a single aspect or discipline. Good integration mechanisms should allow this information to be duplicated in the relevant views while maintaining its consistency across the views. An example approach focusing on the consistency checking between views in software engineering, where the same or closely related entities can appear in different views and must be maintained consistent, can be found in [6]. (2) Depending on the adopted set of views, some information may not belong to one view or the other, but specifies a relationship between different views. For example, the allocation of software components onto the hardware components of a system is the sole concern of neither the software nor the hardware developer, and this design decision lies between the two views. Good integration mechanisms permit the specifications of such inter-view information and reflect the interaction points at which the respective stakeholders need to communicate. Inter-view information can naturally be considered as a view of its own. It is however interesting to highlight the fact that such an "inter-view view" cannot exist on its own, since most of its information lies in the other views it relates. This report focuses on the latter kind of view integration.

B.1.1. Inter-view Modelling - A Complexity Management Technique

Breaking up the system description into multiple views is simply an application of the decomposition or "divide-and-conquer" technique commonly used to manage system complexity. This technique is well adopted in many aspects of science and technology and is generalised in the General Systems Theory ([7] and [8]). A more common application of this principle is hierarchical decomposition, in which a complex system is recursively divided into smaller subsystems until a satisfactory level of detail or complexity is reached. Combining both techniques, system modelling can be envisaged as presented in figure 19, in which the complete system model information is first divided into its various views and then decomposition is used to form a hierarchy of the information specific to each view.

It is argued that a good view integration approach should maintain the use of hierarchies when specifying inter-view information in order to facilitate the developer's work. Relationships setup between views should be appropriately reflected in models and not simply as a list of references. Establishing relationships across the hierarchies of the views provides a tight interweaving of the views. Using this interweaving, mechanisms can be developed to allow a developer within a given domain to view the other aspects of the system from his/her own point of view. The other views should be reflected to the developer at a sufficient level of abstraction and detail that makes him/her appreciate the information provided. Such mechanisms also act as a good basis for information sharing between developers.

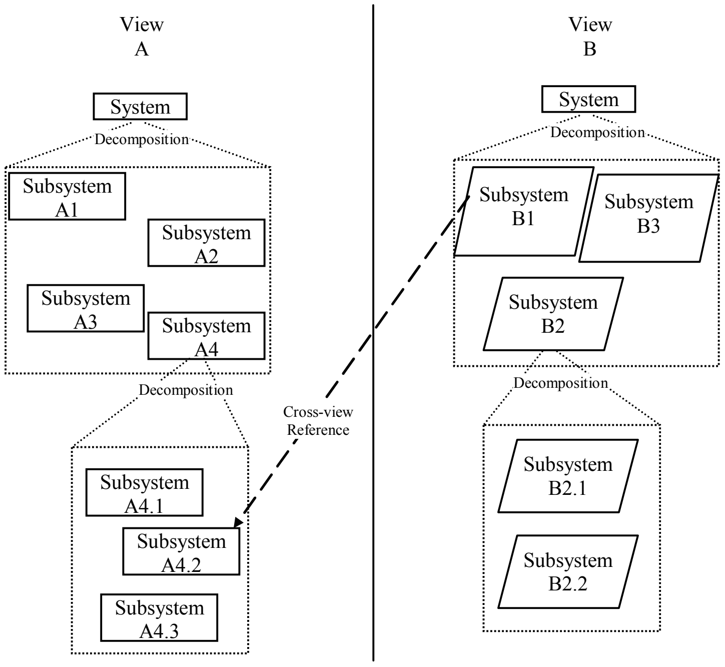


Figure 19. The integration of multi-view and hierarchical decomposition techniques. The broken arrow illustrates a relation between the separated views.

View integration can be performed either through social communication among developers - social development, or through formalised and automated design processes - model based development (MBD) [9].

MBD refers to a development approach whose activities emphasise the use of models, tools and analysis techniques for the documentation, communication and analysis of decisions taken at each stage of the development lifecycle. Models can take many forms such as physical prototypes, graphical and textual models. It is essential that the models contain sufficient and consistent information about the system, allowing reproducible and reliable analysis of specific system properties to be performed. In MBD, analysis plays the critical role of ensuring that the models being built - hence the design decisions being taken – are consistent and satisfy the system requirements.

Within a given domain or view, MBD is commonly used, such as the use of CAD tools in mechanical engineering. This report suggests an approach in which the integration of models from the various design domains is also model-based. By emphasising the use of tools, models and analysis techniques, this ensures the explicit documentation of all inter-view design decisions, making it possible to validate and verify them.

An integrated, model-based, multi-view design environment is also a good basis for the communication of information between developers, where any conflicts and misunderstandings between developers are reflected, dealt with and detected through the models. An integrated environment allows design decisions taken by one developer to be communicated to the rest of the team in an understandable way.

We here propose such a multi-view integration approach. In particular, through the use of a case study, model integration is investigated for the allocation of system functions onto the implementing hardware architecture. The resulting approach maintains the principle of hierarchical design within, as well as between the views, where allocation can be performed across the hardware and function hierarchies. Rules and mechanisms are developed to ensure the completeness and correctness of such inter-view design decisions. Additional mechanisms allow a developer within a given domain to view the other aspects of the system from his/her own perspective, making view integration a good basis for information sharing. The developed allocation rules permit the refinement of allocation specifications performed higher up in the hierarchies, as well as their extensions at the lower levels.

The next section briefly introduces a small case study that will be used throughout the paper to exemplify the approach. The meta-meta-model that should be used in defining a single view of the system model is then defined in section B.3, and exemplified using models relevant for the case study in section B.4. The section ends with a discussion on conventional integration mechanisms, highlighting their shortcomings and defining a set of desired requirements. In section B.5, the multi-view integration approach, satisfying these requirements, is suggested and explained through the case study. Section B.6 presents typical cross-view analyses that can be performed with this approach, followed by a short description of the implementations performed in section B.7. A discussion of related work is presented in section B.8, before concluding the paper in section B.9. Two typographic conventions are used in this report: (1) *Italics* are used for the definition of a term or keyword. (2) Once defined, L e t t e r s p a c i n g is used for most keywords in the remaining parts of the report. This is necessary given the multi-word composition of some keywords, simplifying their identification in the text.

B.2. Case Study

The following case study is an extract from a larger effort performed in cooperation between Scania AB and the Royal Institute of Technology, aimed at quantitative analysis of architectural design decisions [10].

The original case study deals with the increased design complexity of modern truck systems accompanying the introduction of software-based functionality in an otherwise mechanical product. Among other reasons, complexity arises due to the increased number of functions introduced. More importantly, complexity arises from the interdependencies between these functions, where functions need to share common resources such as memory space on Electronic Control Units (ECU), as well as cooperate with other functions in order to fulfil their expected behaviour.

During the early architectural design of a truck, architects face the challenge of choosing the Electrical/Electronics (EE) architecture, onto which the system functionality is to be implemented, taking into consideration and optimising design parameters or keyfigures such as the resulting cable weights, costs and the number of weak connection points. Additional aspects of the system design to be taken into consideration include reliability, available technology, safety, sub-contractors, etc. The EE architecture of a truck consists of a network of communicating ECUs of varying complexity. A critical factor that affects keyfigures is the allocation of system functions onto these ECUs. Different function allocations provide different performance requirements of the ECUs, communication bandwidths, and different sets of cable connections between ECUs for communication.

Evaluating keyfigures and making trade-offs between them is often performed through qualitative investigation efforts. The aim of the original case study was to perform quantitative keyfigure analysis, based on accurate models, to guide these tradeoffs. In addition, the EE architecture and the system functionality are currently modelled within one view, reducing the possibilities to easily explore different allocation strategies without changing the model itself.

In the original case study, a tool was developed which allows the specification of a hardware and functional architecture, followed by the possibility to specify various allocation specifications from which keyfigures can be calculated. These keyfigures become a trade-off basis for choosing the most appropriate allocation strategy.

In this report, we consider a subset of the complete truck functionality handled in the larger case study, to illustrate how the two views of the system ought to be separated and integrated, simplifying the process of function allocation. We illustrate how our technique of multi-view modelling identifies two types of concerns to be separated: Intra-view relations specified in the given view's model, and inter-view relations that deal with integrating views.

In particular, we focus on the Adaptive cruise control (ACC) function. ACC is a typical distributed functionality that requires the cooperation of many components of the system. ACC may be seen as an extension to the conventional cruise control, where ACC not only keeps the speed but also ensures a given distance to the vehicles ahead. The ACC is mainly seen as a comfort oriented function, although it could be seen as the first step towards more autonomous driving. In the future, this step could be followed by various functions aimed at comfort, safety and fuel economy. Sections B.4.2 and B.4.3 illustrate models of the ACC functionality and of the implementing hardware components respectively.

The ACC functionality described in this report is hypothetical and does not necessarily match that adopted at Scania. In particular, the function specification has been reorganised in order to introduce a hierarchical specification.

B.3. Single-view Modelling

In representing a given system, the types of properties selected are based on those properties that the observer or user is interested in and is capable of observing. Given that a system may have many different users, the set of properties to be represented needs to be the union of the properties of interest for each of the users.

A single representation covering all the needed properties can be provided. This solution implies that observers are exposed to properties to which they have no interest. Another solution is to provide a different *view* for each of the concerned observers, onto which the system properties are distributed. Each view of the system is represented using a single *model*. This solution allows observers to focus on the properties of their concerns. A system is hence said to be represented using a set of models together with their relationships. This definition of the "model" and "view" concepts almost agrees with that presented in the IEEE-1471 standard [1]. While in our definition, views and models form a one-to-one relationship, the standard defines one-to-many relation, where a view is represented using one or more models. This set of "models" is grouped into one in our terminology. A many-to-one relation, where a model is used to represent more than a single view of the system is not desired, since this would require the need to define which parts of the model belongs to which view.

B.3.1. The Meta-meta-model

Multi-view modelling generally requires that a certain meta-meta-model is defined from which the specific models are eventually instantiated [11]. This allows for many concepts to be reused across all model definitions, and hence facilitating the integration of these models.

We adopt a simple meta-meta-model which generalises among established meta-meta-modelling languages such as MoF [11], Dome [12] and GME [4], and based on a broad survey of modelling languages for embedded computer systems [19]. Since the suggested concepts are very basic and general, it is expected that most modelling languages can be instantiated using this meta-meta-model. It is important to note that the main aim is not to suggest yet another meta-meta-model that claims to cover any modelling language. A simple, generalised meta-meta-model was adopted, allowing focus to be placed on the view integration mechanisms.

As further detailed in this section, a model can be generally viewed as consisting of a hierarchical structuring of *elements* that may possess *properties*; *ports* defining interfaces to these elements; and *relations* (such as associations, inheritance and refinement) between ports. Modelling languages differ in the kinds of elements that can be specified, their relationships and the kind of properties they possess. The meta-meta-model is first instantiated to reflect a given meta-model by defining the kind of elements, ports and relations that will exist in that particular model. The meta-model is then further instantiated by the user when defining a specific model for a specific system. Figure 20 shows a graphical presentation of the concepts discussed in this section.

The main concept that is recurring in most modelling languages and will be adopted here is composability. In dealing with large complex systems, a system can be seen as consisting of a set of parts which together, through their interrelations, describe certain aspects of the system such as its functionality, structure, etc. These parts are considered systems of their own, which similarly consist of interrelated parts. This recursive decomposition of the system into its constituting parts helps in managing and absorbing the complexity of the system, where the observer can focus on a part of the system that is of interest at a given point in time while ignoring the others. Note that decomposition is not necessarily an intrinsic property of the system, but a technique of perceiving and structuring a system adopted by the observer to better grasp its details.

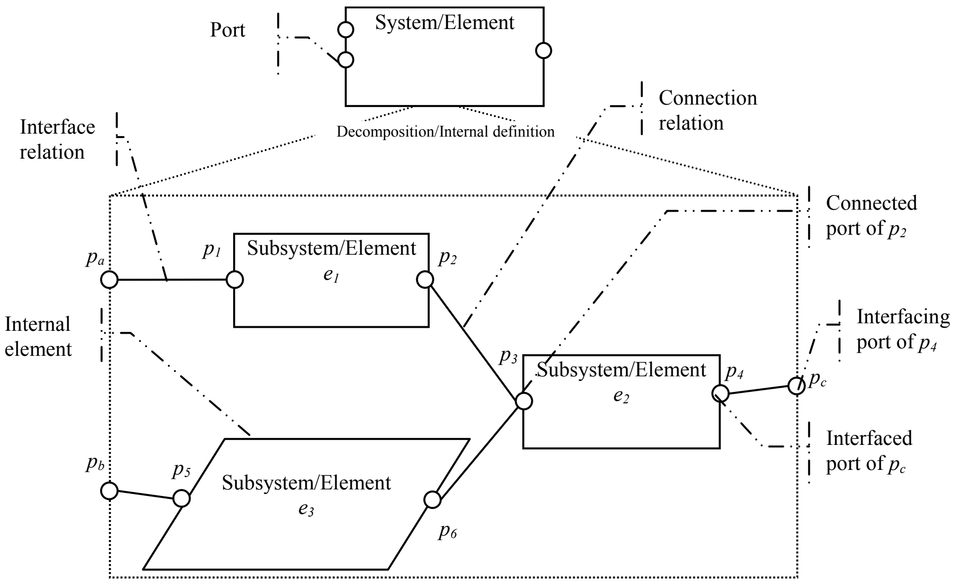


Figure 20. A graphical representation of the general modelling concepts.

B.3.1.1. Elementary and Composite System Definition

A system's properties are described by an *element*. An element is a placeholder of *attributes* describing the represented system's (relevant) properties.

For a simple description of an element, the properties can be specified as a set of attributes. Such a description is known as an *elementary element*. In defining a specific meta-model, the model designer specifies different types of elementaries to describe different types of systems, with each elementary type having a different set of properties.

When the complexity of the system increases, the use of elementaries becomes insufficient to satisfactorily specify all properties of interest. It may become impossible to define properties whose values can be simply specified; there may exist complex interdependencies between the properties; or the number of properties set may be too large to handle. For elaborate descriptions, the properties of the system can be decomposed into smaller, less complex, interacting elements, where each element contains a subset of the original system properties. Such a description is known as a *composite element*. In relative terms, a composite element is known as the *parent element* to each of its composing elements – the *internal elements*.

The internal elements of a composite can themselves be either elementary or composite elements. In this hierarchical decomposition, an element of a system becomes a system of its own, with its own set of elements and so on. The recursive decomposition terminates arbitrarily at a certain level once the level of complexity reached for a part is satisfactory, and the parts can be simply described. The decision of when an element can be described by a simple set of properties is made by the designer and reflects his/her mental capabilities and purposes.

Depending on the context used in viewing a certain element, two different descriptions of the element properties can be identified. If viewing the element as the parent element containing other elements, then the *internal definition* (white-box definition) deals with its complete set of properties, which consists of the set of internal elements. This definition defines the element as a stand-alone system and hence needs to be complete irrespective of its surrounding environment. If viewing the element as a composing element of a larger parent element, then the *external/interface definition* (black-box definition) reveals only those properties that need to be shared with the system environment. From the environment perspective, this definition is sufficient to know how the element can be used and related to other elements, while ignoring its internal workings.

B.3.1.2. Element Interface

The interface definition of an element is an extract of the internal definition, and is defined by a set of ports. A *port* forms part of the interface of its element and acts as a placeholder for a subset of its element's externally accessible properties. It is through ports that an element interacts with its external environment.

An interaction between elements is described through a *relation* between their ports, indicating a certain relationship between the properties specified in the ports. Two general types of relations are identified: *Interface relation* and *connection relation*.

In order to externally reveal the internal properties of an element, an element's port establishes an *interface relation* to the port of the internal element with the properties of interest. In figure 20, the interface relation between the ports p_a and p_1 indicates that the interface properties of the internal element e_1 are externally accessible. In relative terms, the port of the internal element is called an *interfaced port* of the port of the parent element. The latter is called an *interfacing port* of the port of the internal element. In this way, a port acts as a gate to the internal properties of its element to which the environment connected to that port gains access. Each direct interfacing port can have one, and only one, direct interfaced port and vice versa.

Section B.3.1.1 presented a simplified technique of distributing a composite element' properties into elements. However, it is generally not possible to obtain such independent elements. Certain properties that end up in specific parts need to be related to other properties in other parts, and relationships need to be specified between the elements to describe these dependencies. A complete system description hence consists of its composing elements, as well as the relations between them. A *connection relation* is established between two ports of peer elements, implying a certain dependency between the properties specified in the ports. (See figure 20 for an example connection relation between the ports p_2 and p_3.) The ports with such a relation are called *direct connected ports*.

We define the *equivalent ports* of a port to be the combined sets of its interfacing ports and interfaced ports (as well as itself). Given the definition of an interface relation, equivalent ports are hence the representations of the same set of properties of the system. Without any loss of information, an element/system can be replaced by its set of internal elements, where the interfaced ports of its internal elements connect directly to the ports which the interfacing ports connect to. This procedure can be executed down the hierarchy until the view consists of a flat structure of elementary elements. In other words, the model hierarchy is arbitrary, based on the needs of the developers.

We define the *connected ports* of a port to be the set of its direct connected ports and each of their equivalent ports, together with the direct connected ports of the equivalent ports of this port. Again satisfying the definition of equivalent ports, the set of connected ports of a port is the same as that for each of its equivalent ports.

B.3.1.3. Specifying Port Properties

A port's properties can be defined either directly (*direct properties*), or through one of its equivalent ports (*inherited properties*). If the port properties are allowed to be simultaneously defined in multiple equivalent ports, a source of potential inconsistency between the specifications is created. It becomes necessary to ensure that all specifications are consistent whenever a change occurs (such as when creating a new interface relation, or changing the properties in one of the equivalent ports). Another simple solution is to allow properties to be defined on only one port among the set of equivalent ports, avoiding duplications of property definitions and hence inconsistency problems. In this case, once the initial choice of the equivalent port is defined, no other equivalent port can be used to define properties. This condition needs to be checked whenever a new interface relation between two ports is created, since the ports become equivalent and it is necessary to ensure that the new set of equivalent ports has only one port definition.

B.3.1.4. General Principles

In the definition of this meta-meta-model, we try to adhere to a few basic principles:

- An element/system is fully defined by its internal definition, whether it is a set of properties or a set of consisting elements and their relations. That is, a system or element is independent of its surroundings. Its properties cannot be defined based on properties of its peer elements nor its parents up in the hierarchy. In other words, it should be possible to remove a system from its current surroundings and place it in another, without changing its internal properties.
- An element's internal and interface definitions should be fully specified through the interface definitions of its direct children elements. In other words, the element does not need any information about the internal properties of its children.

These principles are beneficial in many ways:

- The concept that each element is a system of its own is reinforced, since external changes and reorganisation do not influence that system/element.
- From the user perspective, the concept that the internal elements can be treated as black boxes with a certain interface is reinforced. There is no need to study the direct children's internal definitions in order to define the element's properties or to check its correctness, as long as the internal elements are assumed to be correct. Only the internal elements and their relations are needed.
- Systems can be built and checked independently and then used as elements inside a larger system providing a mechanism for building libraries of reusable elements.
- Constraint rules and mechanisms relating the different modelling entities (views, elements, properties, etc.) can be applied more locally. For example, checking the validity of an element's interface requires only access to the element's direct children without reference to other elements in the system or further elements down or up the hierarchy.

- Once a change is made to an entity, the reapplication of the rules and mechanisms to maintain the model validity is also restricted to a smaller local subset of the system's direct elements. This permits the implementation of more efficient dynamic constraint checking mechanisms.

B.3.1.4.1. Inheritance

Inheritance is the mechanism of specifying a property of a system based on other properties specified elsewhere. It can be viewed as an automation of the manual specification of properties, in the case where only one choice would have been available for a valid model.

The inheritance mechanisms should satisfy the principles specified above. For example, a property of an element can only be inherited from properties specified by its direct children. A port's properties can only be inherited from its direct interfaced port down the hierarchy.

Certain exceptions to the principles specified above may sometimes appear to be made when setting up inheritance mechanisms. The specification of properties among equivalent ports is a typical example (see section B.3.1.3). In that case, it was allowed to specify the port properties at any level among the equivalent ports, and all other equivalent ports (up and down the hierarchy) simply inherited these specifications. This can be interpreted as a violation of the above principle. While it is acceptable to allow the inheritance of the port specifications up the hierarchy (by step-wise inheritance), the inheritance down the hierarchy from a port to its interfaced port is a violation since the element specification is no longer independent of its surrounding environment. In order to satisfy the need that all equivalent ports have equivalent properties, a strict application of the principles means that properties can only be specified at the ports of elementary elements. This solution is however restrictive for the user, and would not be desired.

We hence differentiate between the *inheritance* of the properties in the models which strictly follows the above principles, and the *convenience inheritance* for the user which is more flexible. In the case the property is specified at a high level by the user, this property is actually specified at the equivalent port lowest down in the hierarchy (There is only one such port since each port can only have one direct interfaced port). The properties hence become inherited up the hierarchy by all the equivalent ports. In the case the elementary or any element with an equivalent port, for example, is taken out of its context, its properties remain specified as well. In this way, while the simplification is performed for the user, the model specification still adheres strictly to the above principles.

B.3.1.5. Instantiating a Meta-meta-model

In defining a particular viewpoint (meta-model), the model designer specifies the kind of elements, ports and relations that exist in any model, as well as the rules and constraints governing their use. The following need to be specified:

- The set of composite element types, together with their properties.
- The set of elementary element types, together with their properties.
- The set of relation types between element types, together with their properties.
- The set of port types of each element type
- The rules constraining the kind of models that can be built, by constraining the usage of the above entities.

The choice of these types and constraints is left to the model designer. A common question arising during such a design is whether some aspects of the system are to be modelled as elements or relations. It is often the case that, while in certain models of the system certain aspects are best described as being a part of the system, in other models they are best described as relations between parts. A sound indicator of whether something is to be an element or a relation is that elements are considered systems in their own right and can be further broken down into subparts, while relations are described through simple properties with no decomposition.

B.3.2. Formal Notation

A model can be described mathematically using set notation. This will help define and formalise the rules and conditions for inter-view associations in section B.5. A summary of the following terminologies and notations can be found in Appendix A and Appendix B respectively.

A model *M*, of a certain view, *V*, is defined as an ordered set $M = (E, P, H, G, R_i, \alpha, R_c, \beta)$, where

- *E* is the set of elements of view *V*.
- *P* is the set of ports of view *V*.
- *H* is a binary relation from *E* to *E*, denoting the direct parenthood relationship between element nodes. Considering the parenthood relations between the element nodes, *M* is a directed tree, or an acyclic directed graph, where exactly one node called the root has indegree 0 while all other nodes have indegree 1 [13].

$$H \subseteq \{(c,p): c \in E \wedge p \in E\}$$

- G is a binary relation from P to E, denoting the containment relationship between elements and their interface ports.

$$G \subseteq \{(p,e): p \in P \wedge e \in E\}$$

- R_i is the set of interface relations, and α is a mapping from R_i to ordered pairs of $P \times P$, denoting the interfacing relationship between the ports of the parent element and the ports of its internal elements.

$$Rng\alpha \subseteq \{(p_e, p_i): p_e \in P \wedge p_i \in P\}$$

- R_c is the set of connection relations, and β is a mapping from R_c to unordered pairs of $P \times P$ denoting the connection relationship between ports.

$$Rng\beta \subseteq \{\{p_1, p_2\}: p_1 \in P \wedge p_2 \in P\}$$

B.3.2.1. Further Notations

- The *direct children* of element *e*, *$E_{dc}(e)$*, are defined as the set

$$E_{dc}(e) = \{c \in E : (c,e) \in H\}$$

- Element *c* is said to be a *direct child* of *e* if $c \in E_{dc}(e)$
- Element *p* is said to be a *direct parent* of element *c*, *$e_{dp}(c)$*, if $(c,p) \in H$

Notation: $p = e_{dp}(c) \Leftrightarrow (c,p) \in H$

- The *parents* of element *e*, *$E_p(e)$*, are defined as the set

$$E_p(e) = \{p \in E : (\exists e_1, e_2, ..., e_n \in E : (e,e_1) \in H \wedge (e_1,e_2) \in H \wedge ... \wedge (e_n,p) \in H)\}$$

- Element *n* is said to be a *parent* of *e* if $n \in E_p(e)$
- The *children* of element *e*, *$E_c(e)$*, are defined as the set

$$E_c(e) = \{c \in E : (\exists e_1, e_2, ..., e_n \in E : (e_1,e) \in H \wedge (e_2,e_1) \in H \wedge ... \wedge (c,e_n) \in H)\}$$

- Element *n* is said to be a *child* of *e* if $n \in E_c(e)$
- Element *e* is said to a *elementary*, *$e_l(e)$*, if $E_{dc}(e) = \emptyset$

$$\text{Notation: } e_l(e) \Leftrightarrow E_{dc}(e) = \varnothing$$

- Element *e* is said to be a *root*, *e_r(e)*, if $E_p(e) = \varnothing$

$$\text{Notation: } e_r(e) \Leftrightarrow E_p(e) = \varnothing$$

- Element *e* is said to be the *containing element* of port *p*, $e_g(p)$, if $(p, e) \in G$

$$\text{Notation: } e = e_g(p) \Leftrightarrow (p, e) \in G$$

- The *ports* of element *e*, $P_e(e)$, are defined as the set

$$P_e(e) = \{p \in P : (p, e) \in G\}$$

- Port *p* is said to be an *port* of *e* if $p \in P_e(e)$
- Port *n* is said to be the *direct interfacing port* of port *p*, $p_{di}(p)$, if $(n, p) \in Rng\alpha$

$$\text{Notation: } n = p_{di}(p) \Leftrightarrow (n, p) \in Rng\alpha$$

- Port *n* is said to be the *direct interfaced port* of port *p*, $p_{de}(p)$, if $(p, n) \in Rng\alpha$

$$\text{Notation: } n = p_{de}(p) \Leftrightarrow (p, n) \in Rng\alpha$$

- The *direct connected ports* of port *p*, $P_{dc}(p)$, are defined as the set

$$P_{dc}(p) = \{n \in P : \{n, p\} \in Rng\beta\}$$

- Port *n* is said to be a *direct connected port* of *p* if $n \in P_{dc}(p)$
- The *interfacing ports* of port *p*, $P_i(p)$, are recursively defined as the set

$$P_i(p) = p_{di}(p) \cup P_i(p_{di}(p))$$

- Port *n* is said to be an *interfacing port* of *p* if $n \in P_i(p)$
- The *interfaced ports* of port *p*, $P_e(p)$, are recursively defined as the set

$$P_e(p) = p_{de}(p) \cup P_e(p_{de}(p))$$

- Port *n* is said to be an *interfaced port* of *p* if $n \in P_e(p)$
- The *equivalent ports* of port *p*, $P_{eq}(p)$, are defined as the set

$$P_{eq}(p) = p \cup P_i(p) \cup P_e(p)$$

- Port *n* is said to be an *equivalent port* of *p* if $n \in P_{eq}(p)$

- The *connected ports* of port *p*, P_c *(p)*, are defined as the set

$$P_c(p) = \bigcup_{n \in P_{eq}(p)} \bigcup_{m \in P_{dc}(n)} P_{eq}(m)$$

- Port *n* is said to be an *connected port* of *p* if $n \in P_c(p)$

B.3.2.2. Model Properties

For a valid model *M*, the following properties can be asserted:

- *H* is a function relation since each child has only one direct parent.
- *G* is a function relation since each port is only contained within one parent element.
- Rng α is a one-to-one function relation, since each direct interfacing port can have one, and only one, direct interfaced port and vice versa.
- $\forall(p_e, p_i) \in Rng\alpha, e_g(p_e) = e_{dp}(e_g(p_i))$
- Rng β is a many-to-many relation.
- $\forall(p_1, p_2) \in Rng\beta, e_{dp}(e_g(p_1)) = e_{dp}(e_g(p_2))$

B.4. Case Study Models

B.4.1. Design and Analysis Views

The different system views can be categorised into *design views* and *analysis views*. A design view is used to model and document the design decisions that the developers have made, allowing also for the communication of information between the different developers. Example design views are:

- *Function Structure* view, describing the functionalities of the system and the information flow that exists between them.
- *Function Behaviour* view, describing the behaviour of the system functionalties.
- *Hardware Structure* view, describing the physical components of the system, and their connections.
- *Cabling* view, describing the cables of the system and the components they connect.

- *Power Supply* view, focusing on the power network of the system.

Unlike design models, an analysis model does not document any design decisions made, but simply present specific aspects from the set of design models in a certain way that facilitates the performance of an analysis. So in principle, the same analysis can be performed given the collection of design models of the system, but an analysis view condenses the information by only revealing what is relevant for that analysis. Example analysis views are:

- *Timing Analysis* view, focusing on the timing aspects of the system behaviour.
- *Safety Analysis* view, focusing on the safety aspects of the system behaviour.

Analysis models are extracted from the design views. The process can in many cases be performed automatically; however, there may be cases in which the analyst needs to take certain “analysis decisions” to perform valid analysis. This may be the case when the analysis technique used needs a simplified model of the system and the decision on how to simplify the design models cannot be automated and require the analyst’s choice. For example, in timing analysis, the analyst may need to decide which of the two modes of operations of a certain task to be considered for analysis, if the analysis technique at hand cannot handle different modes of operations.

In most modelling tools, no distinction is made between these view types. Any analysis performed assumes an implicit analysis view, not accessible to the user. In few cases, such as [28], such a distinction is made, where the design data-flow model is first transformed into a fault tree model onto which safety analysis can be performed.

In the following subsections, we exemplify our meta-meta-model using two design views relevant for the case study of section B.2, namely the *Function Structure* and *Hardware Structure* design views. The specification of associated views in section B.5.1.2 is a step towards the definition of analysis views. It remains however to ensure that the analyses discussed in section B.6 make use of these views.

B.4.2. Function Structure

This section defines an instance of the meta-meta-model - the *Function Structure* meta-model, used to specify the structure of the functions to be implemented in a system. Through the ACC case study, we discuss how this model is used to describe the structure of vehicle functionality.

This meta-model is very similar to the traditional data flow diagram [14] adopted in many modern tools such as Matlab/Simulink [15], representing functions as well as the required information flow between them. In this case study, we are not interested in a complete behavioural description of each function, and a structural specification suffices, since the analysis of interest is not concerned with the system's dynamic behaviour. In addition, the links between functions are modelled as first-class elements of their own, and not simply as connection relations between functions, since the data flow between functions is of major concern during function allocation, and it hence becomes necessary to focus the modelling effort on these links.

B.4.2.1. Elements

Two types of elements are defined: *functions* and *communication links*. A function element designates certain functionality that given a certain input, produces a certain output. A communication link element designates a link that transports data between functions.

These element types are arguably similar, taking certain input and producing output. The difference lies in the intention of each type, which is ultimately decided upon by the user. A communication link element differs from a function element in that its main purpose is the data transfer it performs, while its functionality becomes a side effect. The function element's main purpose is to transform its input data to produce some output data, where the transformation is not seen as a transfer of data (See [16] for a detailed discussion of this issue).

Both elements can be either elementary or composite. In describing simple systems, the elements can be elementaries, while composite elements can be used for more complicated descriptions. A composite function element designates an aggregation of other composite and elementary function and communication link elements, providing a certain interface to them. A composite communication link element designates an aggregation of other composite and elementary communication link elements (but not function elements), providing a certain interface to them. It is desired to restrict the content of communication links to not include function elements, since it is argued that communication links should only model communication between functions, and not contain any functionalities.

B.4.2.2. Element Interface

For function and communication link elements, port properties consist of a set of *data* items, where a data item consists of a name, direction (in, out, inout) and type (int, float, etc.). These data items designate a subset of the element's internal data that are externally accessible to other elements.

Connection relations between ports indicate that the input data of one port is the output data of the other. Since ports of function elements can only connect to ports of communication link elements, a connection relation indicates that the connected port of a function exchanges its data via the connected communication link's port. A port connected to more than one port indicates that the data on that port is transmitted through all of the connected ports.

Interface relations indicate that the related port of the internal element is available for external interface.

B.4.2.3. Constraints Summary

For a valid model, the following constraints need to be satisfied:

- A connection relation cannot be setup between two function elements.
- The internal definition of a communication link element can only contain other communication link elements.
- The data properties of related ports should have equal types.
- For a connection relation, the direction of related ports should be opposite.

B.4.2.4. ACC Function Structure Model

Figure 21 illustrates the Function Structure model of the ACC functionality considered in this report. The model is hypothetical and does not necessarily match that adopted at Scania. The highest level in the hierarchical decomposition highlights the control nature of the function, where a control mechanism (*Control*) uses certain sensing of the environment (*Sensing*) to regulate certain actuators that control this environment (*Actuation*). In addition, user interaction is described in the *Human Interface* sub-function.

- For the purposes of this study, the control algorithm can be simply broken down into a decision on the specific target to follow (*Target Selection*), a state machine (*ACC State Machine*) to decide on the mode of the function which is based on user inputs and environment conditions, and a control algorithm (*Distance Control*).
- The control algorithm requires the following properties to be measured from the environment: the vehicle speed (*Speed Sensing*), vehicle yaw rate (*Yaw Rate Sensing*), and the set of nearby vehicles' speeds and distances (*Targets Sensing*). Each such measurement requires some kind of filtering or signal processing.

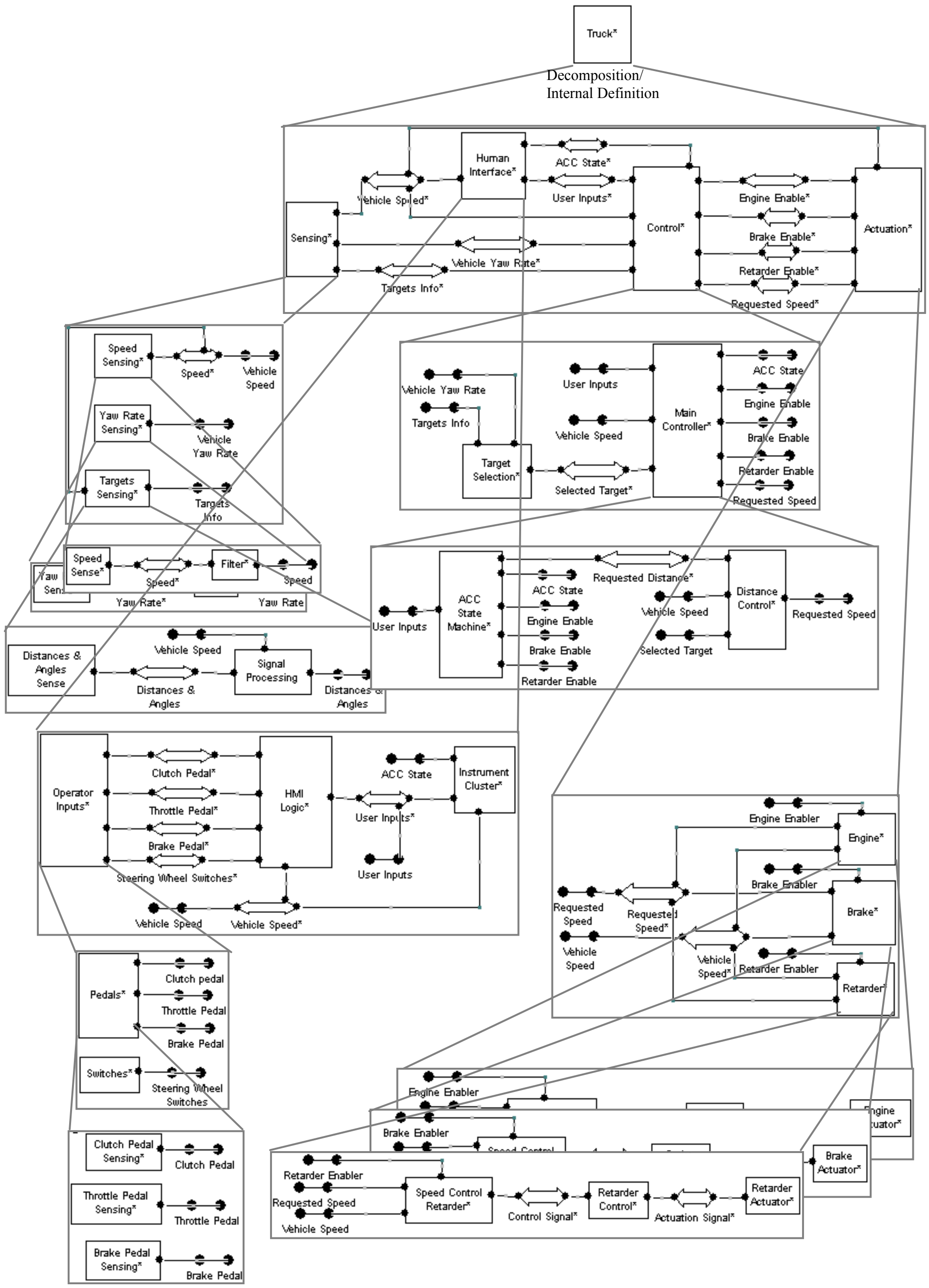


Figure 21. A Function Structure model of the truck ACC functionality.

- The user interaction functionality can be divided into receiving input from the user (*Operator Inputs*), ensuring the validity of any inputs (*HMI Logic*) and feeding back information from the system to the user via displays (*Instrument Cluster*).
- The ACC functionality may actuate the *Engine*, *Brake* and *Retarder* of the truck. Only one of these may be enabled at a time, by requesting a certain vehicle speed to be achieved. Each such request is further broken down into lower level control processes (such as *Speed Control Retarder*, *Retarder Control* and *Retarder Actuator*).

B.4.3. Hardware Structure

This section defines an instance of the meta-meta-model - the *Hardware Structure* meta-model. Through the ACC case study, we describe how this model is used to describe the system's hardware.

The model of the hardware architecture needs to describe the major computational units as well as their connections through which data communication is possible. At the early architectural analysis of this case study, information about the physical location of these units and their connections is sufficient. The accurate physical dimensions are of no interest and we resort to a very simplified geometrical model, specifying approximate unit dimensions. A more accurate model such as that provided by a CAD model could also have been utilised. This is not adopted at this stage, since such models would not contribute to our aim in experimenting with multi-view modelling.

B.4.3.1. Elements

Two types of elements are defined: *hardware units* and *cables*. In describing simple systems, these elements can be elementaries, while composite elements can be used for more complicated descriptions.

An elementary hardware unit element designates a physical block occupying a certain amount of space. It is simply modelled as a 3-D square box and its attributes describe its geometrical dimensions and position. An elementary cable element designates a single cable with a certain geometrical path. Its attributes describe its diameter, density, and its spatial path.

A composite hardware unit element designates an aggregation of other units and cable elements, providing a certain interface to them. Note the abstract nature of these composites. A composite hardware unit is simply an abstract aggregation of a number of physical hardware units and cables, and cannot be viewed as a physical unit itself.

A composite cable element designates an aggregation of cables. A certain length of the cables share a common path, while the extremities can be separated, hence the end-points can have different physical locations. A composite cable is simply a hierarchical management of a number of independent cables which can, but not necessarily have to, be physically bundled together.

B.4.3.2. Element Interface

For hardware unit and cabling elements, port properties consist of a set of *coordinate* items, where a coordinate item specifies a spatial location at which the element can be connected to other elements. A port can be used to specify more than one connection point that can be physically situated in different locations.

Connection relations between ports indicate that the ports' coordinates are physically connected to each other. That is, the connection points of the two ports have the same spatial position. A port connected to more than one port indicates that all connected ports share the same spatial location.

Interface relations indicate that the port of the internal element is available for external connections.

B.4.3.3. Constraints Summary

For a valid model, the following constraints need to be satisfied:

- A connection relation cannot be setup between two ports of hardware unit elements.
- The internal definition of a cable element can only contain other cable elements.
- The connection point properties of two connected ports should have equal values.

B.4.3.4. ACC Hardware Structure Model

Figure 22 shows the complete Scania EE architecture needed to implement the complete functionality set of a truck. The hardware architecture is based on the Controller Area Network (CAN) protocol, with three buses separated by an ECU unit that also acts as a gateway between them. The gateway unit (*COO*) features some software functionality apart from the role of a gateway. ECUs with different levels of system criticality are separated by being placed on different buses. The *Red bus* has ECUs with the highest criticality; ECUs on the *Yellow bus* are estimated to have intermediate criticality; and the ones on the *Green bus* have the lowest level of criticality.

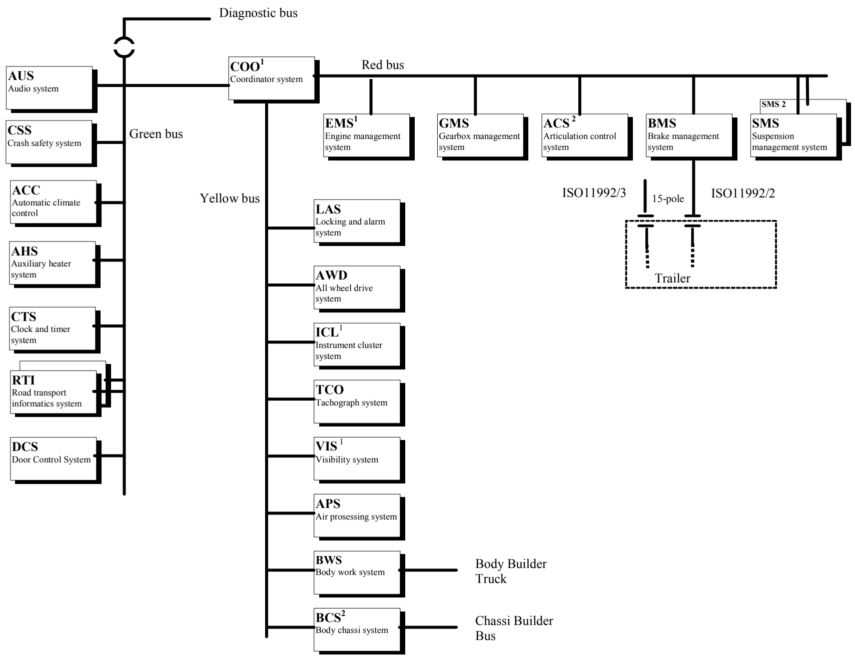

Diagnostic bus
COO[1] Coordinator system
Red bus
AUS Audio system
CSS Crash safety system
Green bus
EMS[1] Engine management system
GMS Gearbox management system
ACS[2] Articulation control system
BMS Brake management system
SMS 2
SMS Suspension management system
ACC Automatic climate control
Yellow bus
ISO11992/3
15-pole
ISO11992/2
LAS Locking and alarm system
AHS Auxiliary heater system
AWD All wheel drive system
Trailer
CTS Clock and timer system
ICL[1] Instrument cluster system
RTI Road transport informatics system
TCO Tachograph system
DCS Door Control System
VIS[1] Visibility system
APS Air prosessing system
BWS Body work system
Body Builder Truck
BCS[2] Body chassi system
Chassi Builder Bus

Figure 22. Scania EE architecture

Figure 23 illustrates a subset of the hardware architecture relevant for the case study considered in this report. This Hardware Structure model is hypothetical and does not necessarily match that adopted at Scania. Additional components such as the *AICC* hardware unit were added to suit the case study. Moreover, components such as sensors and actuators are also defined, providing a more complete hardware specification. The original model is restructured to provide a hierarchical representation. For example, the powertrain management system (*PTMS*) is introduced to group the engine and gearbox management systems (*EMS* and *GMS*). The naming of the ECUs is adopted from the original Scania architecture of figure 22. It would be desired to avoid such naming in the future, since the names are misleading and imply certain functionality, causing bias in the allocation process.

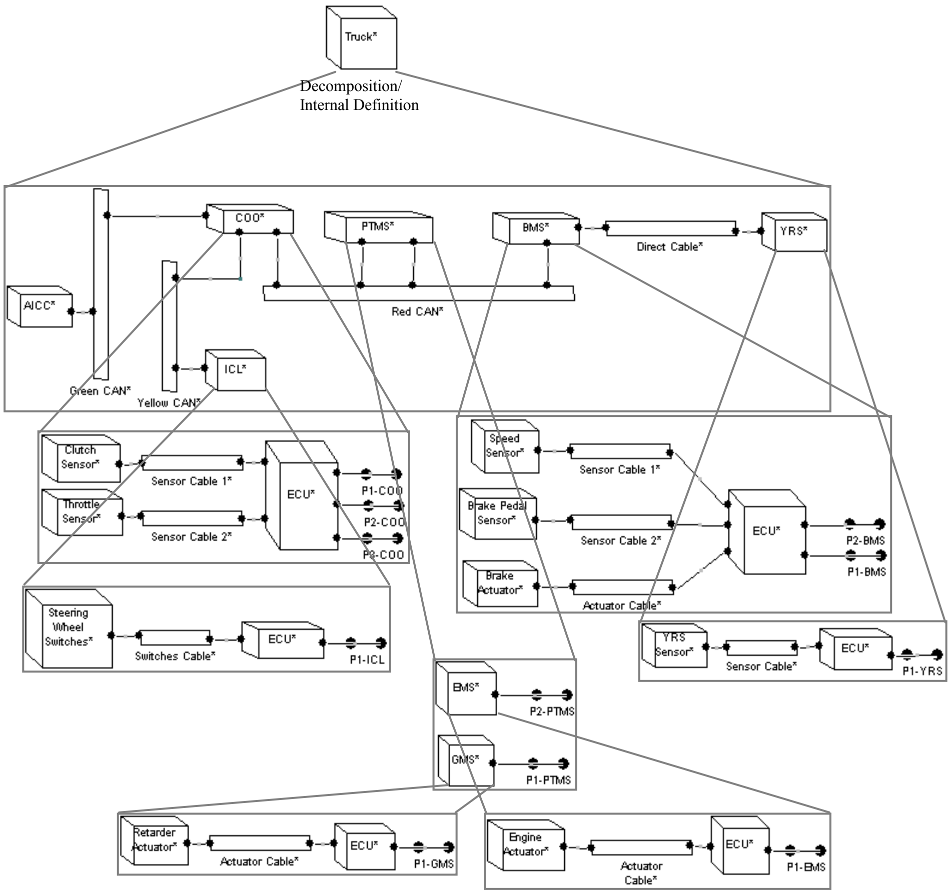


Figure 23. A Hardware Structure model of the truck ACC functionality.

B.4.4. Requirements on View Integration

By specifying the function and hardware architectures as the system's two separate views, the allocation of functions to hardware units and communication links to cables becomes a design decision that lies in between these two views. This allocation step can obviously be treated in a view of its own, with its own model, but as it only deals with relationships between entities of other views this is not needed. Instead the two views can be integrated making use of inter-view relationships.

The simplest and most common solution for integrating views is to flatten the hierarchical structure in either one or both views before inter-view relations are specified. Assuming that both of the views described in section B.3 are flattened, leaf (elementary) functions would be allocated to leaf hardware units. This method obviously fails to make use of the complexity management advantage provided by the hierarchical models during the allocation step. A number of related shortcomings of the method can be identified: Since only leaf entities are related, the context of these, given by their respective hierarchies, is lost during the allocation process. Furthermore, it is difficult to make early coarse design decisions and it becomes necessary to have detailed knowledge about both the particular function and hardware elements by any person performing allocation. Also, if an allocation has been specified and a function is later further decomposed into sub-functions, during a refining design stage, the already existing allocations are lost. In summary, the inter-view allocation is unnecessarily affected by intra-view design changes. All these arguments hold also for the case when only one of the two views is flattened.

Forcing allocation to be done on a leaf level will make the allocation sensitive to changes in either of the two views. What would be desirable is to integrate the different views in a way such that they can both be developed as independently as possible, without affecting the validity of an already chosen allocation. Furthermore, since designers work on different levels of detail in potentially very large systems, one would like to allow allocation decisions to be made on an arbitrary level in the hierarchies. Any decision made would need to be reflected up and down the system hierarchies. This also means that the designer can start with performing rough allocations of a group of functions to a group of hardware units, and then refine the choice down the hierarchy.

Another common approach to view integration is to setup the relationships between the different views based on an import mechanism, where the user in essence maps a complete model into another. Such a mechanism creates a precedence relationship between the views, where one view needs to be first fully developed before the other. In addition, any changes made to the source model are not reflected in the destination until the next transformation is performed, causing inconsistencies between the models. This approach inhibits the possibility of concurrent development between disciplines.

One can also assume a primary view under which the other view is defined. For example, the hardware view can be first defined, and then the functions are distributed over the hardware structure, where each function definition is specified under the hardware units to which it is allocated. This in essence creates a single model structure for the system views. Again, precedence relationship between the views is created, inhibiting concurrent and independent development of the views.

In summary, a model-based view integration environment should satisfy the following:

- **One view – one model.** Preserve the need for a single model for each view of the system since, in most cases, a model user needs only to concentrate on a single aspect at a time.
- **Allocation is inter-view and not intra-view information.** It should therefore not lie in either view, but across views.
- **Preserve the hierarchy.** The inter-view relationships between hierarchically decomposed views should be performed across the hierarchies of the views, independently of the two hierarchies.
- **Independence between the hierarchies.** The choice of hierarchical decomposition within one view should be independent of that specified in another view. Since hierarchy is a tool used to reduce the complexity perceived by a given stakeholder, the use of this tool should not be compromised by the complexity needs of other stakeholders.
- **Concurrent development.** A view development should be performed independently and concurrently of the other views. Each discipline should be able to work independently, yet support for a holistic view should be provided. No precedence should exist in the development of the views.

B.5. Two-View Integration

Similar to the argument in section B.3.1.2, the properties in the different views may be interdependent and hence the multi-view solution is accompanied by the need to setup relations between the views.

To differentiate relations between properties within a view from relations across different views, we refer to the latter as *associations* between properties, while relations hereafter only refer to the former.

This section discusses the mechanisms needed to establish these associations between views for the particular case of integrating a Function Structure with a Hardware Structure view. While these mechanisms are not general enough to be adopted for any kind of inter-view associations, it is believed that they can be easily reused for the mapping of system functionality to the software architecture, or software to hardware allocation. Essentially, the mechanisms can be generalised with little effort to any inter-view information that implies a “implemented by” or “allocated to” relationship. It remains however to test this claim through other case studies in the future.

Setting associations between properties is practically performed through property placeholders, namely elements and ports. Section B.5.1 presents such relationships between elements, and section B.5.2 deals with relationships between ports.

B.5.1. Element Associations

Associating an element in one view to another element in a second view has different implications, depending on the particular views and elements involved. Concerning the case study, the following rules apply when associating elements between the Function Structure and Hardware Structure views:

- Function and communication link elements from the Function Structure view can be associated with hardware unit elements from the Hardware Structure view, indicating that the functional element is physically implemented in that unit.
- Communication link elements can be associated with cable elements, indicating that the communication mechanism designated by the link is performed through the cable.

Associations can be specified between any function and hardware elements, irrespective of whether they are composite or elementary.

Note that an association of a function, *f*, to a hardware unit, *h*, does not necessarily mean that the complete function *f* is implemented on the complete unit *h*, nor that *f* cannot be implemented by other units as well. Instead, the association simply implies that some of the *f* functionality is implemented on some of *h*'s hardware. The remaining *f* functionality may (or may not) be implemented by other hardware units; similarly, the remaining *h* hardware may (or may not) implement other (parts of) functions. This interpretation is important when understanding the element association rules in the following subsections.

When performing design decisions across views, designers would at a given time want to focus on specific parts of the system, at a certain level of abstraction, without being concerned with more detailed design decisions. For example, a designer may wish to specify that the brake system is to be implemented on a certain group of processors, without needing to specify in detail which specific brake sub-functions is to be allocated to which processor. Such a decision can be further refined by others or at a later stage. Conversely, the more detailed allocation design decision of a particular function to a processor must be reflected to the high level functions containing it.

In addition, to satisfy the requirement that views should be developed independently, it is necessary to allow associations between elements of different views to be made across the hierarchy. In other words, an element in a certain view, at a certain depth in its hierarchy is not restricted to be associated to elements in the same depth in another view, instead it can be associated to any valid element throughout the hierarchy.

However, consistency between the high level and the lower level design decisions needs to be maintained. This can be realised by specifying that: A function implemented on a certain hardware unit means that it is also implemented by hardware units containing this hardware unit. Conversely, a unit implementing a certain function, means that this unit also implements (part of) functions that contain this function.

The following subsections discuss how such cross-hierarchy associations ought to be interpreted and managed in order to satisfy these needs.

B.5.1.1. Associated Elements

We define the following, for associations between elements from view V_x to view V_y:

- The *direct associated elements* of element e_x in view V_y, $A_d(e_x, V_y)$, is defined as the set of elements in V_y, directly associated by the user on element e_x. Direct associations are bidirectional meaning that if e_x is associated to e_y, then e_y is also associated to e_x. See section B.5.1.3 for conditions for such a valid set.
- The *inherited associated elements* of element e_x in view V_y, $A_i(e_x,V_y)$, is defined as the set of topmost direct associated elements of e_x's children, excluding those which have already been defined, or generalised, through the direct associated elements of e_x, $A_d(e_x, V_y)$.

$$A_i(e_x,V_y) = \left\{ a \in \bigcup_{n\in E_c(e_x)} A_d(n,V_y) : \left(\neg\exists m \in \bigcup_{n\in E_c(e_x)} A_d(n,V_y) : m \in E_p(a) \right) \wedge \left(\neg\exists m \in A_d(e_x,V_y) : m \in E_p(a) \vee m = a \right) \right\}$$

- The *associated elements* of element e_x in view V_y, $A_a(e_x,V_y)$, consists of the union of its direct associated elements and its inherited associated elements.

$$A_a(e_x,V_y) = A_i(e_x,V_y) \cup A_d(e_x,V_y)$$

Note that the above definitions are specified so that $A_d(e_x,V_y) \cap A_i(e_x,V_y) = \varnothing$.

The associated elements, $A_a(e_x,V_y)$, can be interpreted as the result of a filter applied onto the associated view V_y, in which only the elements associated to e_x and additional associations specified at the more detailed levels are considered.

In figure 24, the *COO* hardware unit is directly associated to the *Main Controller* and *Operator Inputs* functions,

$A_d(COO, V_{FS}) = \{Main\,Controller, Operator\ Inputs\}$; where V_{FS} denotes the Function Structure view.

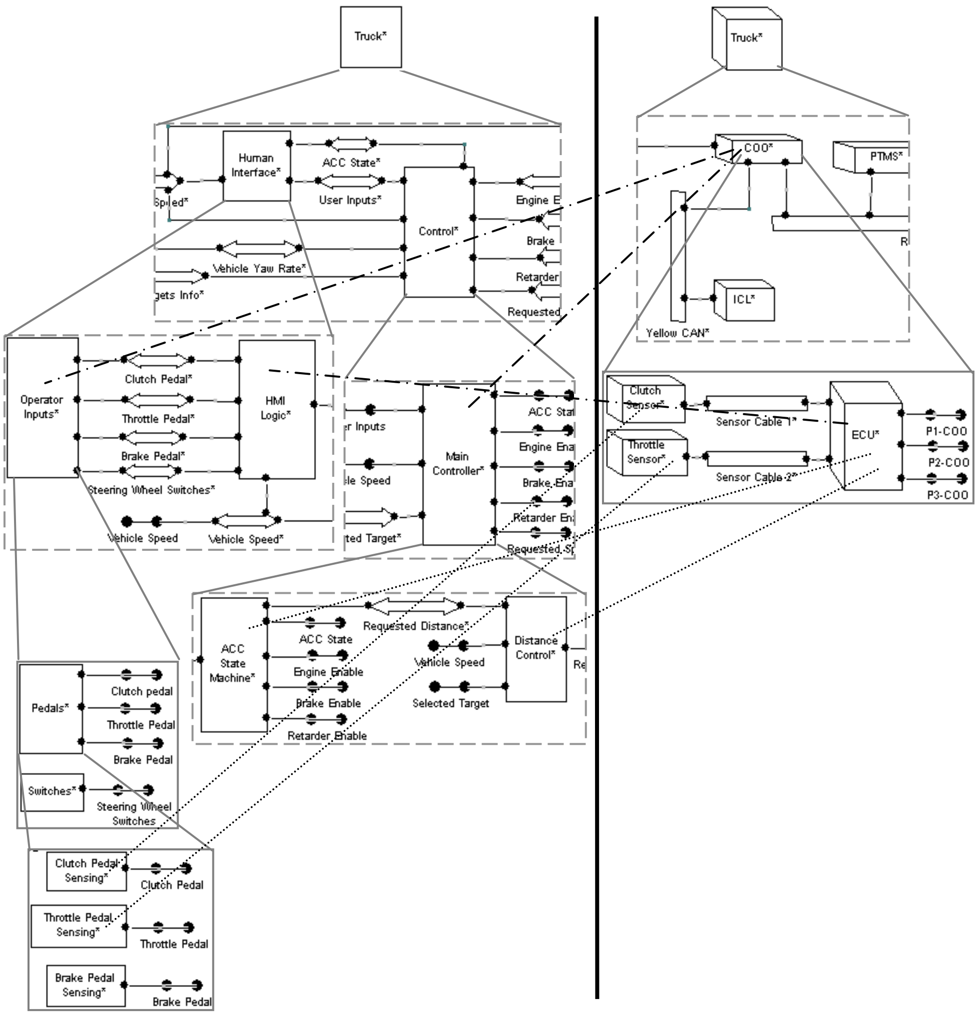


Figure 24. The direct associations of the hardware unit *COO*, as well as some of its child units *ECU*, *Clutch Sensor* and *Throttle Sensor*. The associations from *ECU* to *ACC State Machine* and *Distance Control* specialise that specified to *Main Controller*.

Furthermore, the sub-function *HMI Logic* is associated to the *ECU* unit of *COO*. The association between the *ECU* and *HMI Logic* indirectly implies that the *COO* unit also implements *HMI Logic*. *HMI Logic* is said to be an inherited associated element of *COO*, $A_i(COO,V_{FS}) = \{HMI\ Logic\}$. In integrating these design decisions from the various levels, the *COO* is to (partly) implement the *Main Controller* and *Operator Input* functions, as well as *HMI Logic*, $A_a(COO,V_{FS}) = \{Main\ Controller, Operator\ Inputs, HMI\ Logic\}$.

In refining the above design decisions, the direct association between *COO* and *Main Controller* can be further refined by directly associating the *ECU* hardware unit to the *ACC State Machine* and *Distance Control* functions. This association implies a more detailed specification of the allocation of the *Main Controller*'s functionality to specific hardware units. The associated functions are not considered as inherited associations to *COO*, since they specialise an already existing association, namely the parent *Main Controller*. In a similar refinement step, *Clutch Pedal Sensing* and *Throttle Pedal Sensing* are associated to the *Clutch Sensor* and *Throttle Sensor* sub-units respectively.

Finally, the allocation of functions to *COO* is not considered complete in this case since the allocations to its remaining sub-units (the sensor cables) still need to be specified (see section B.5.1.7 for a discussion on completeness conditions).

As a consequence of the above association definitions, if e_x is associated (directly or indirectly) with the elements e_1, e_2, …. e_n, then e_x's children will in effect only be associated with e_1, e_2, …. e_n, or any of their children. As soon as a child of e_x is associated with an element that is not in this set, this element also becomes an associated element of e_x (unless its parent already is), and hence the above rule still applies. In other words, the children of e_x can either specialise (refine) the parent's associations, or extend them; the propagation of the extended associations up the hierarchy have the same effect as specialisation.

Allocation is strongly related to the design process and can of course be carried out in different ways. The above mechanisms support a process-independent allocation practice. By placing certain restrictions, the allocation practices can be constrained. For example, disallowing the possibilities for association extensions through the sub-systems enforces a top-down approach, where sub-system design can only refine design decisions specified at the higher level.

Given the above definitions, in order to deduce the $A_i(e_x, V_y)$ set, one needs to consider the A_d set of all the children of e_x down the hierarchy. The A_i set of the children can be ignored since these will be reflected anyway by other children down the hierarchy. However, as specified in section B.3.1.4, it would be desired to establish $A_i(e_x, V_y)$ by only considering e_x's direct children.

As proved in Appendix C.1, $A_i(e_x, V_y)$ can be redefined in terms of e_x's direct children only as follows:

$$A_i(e_x,V_y) = \left\{ a \in \bigcup_{n \in E_{dc}(e_x)} A_a(n,V_y) : \left(\neg \exists m \in \bigcup_{n \in E_{dc}(e_x)} A_a(n,V_y) : m \in E_p(a) \right) \wedge \right.$$
$$\left. \left(\neg \exists m \in A_d(e_x,V_y) : m \in E_p(a) \vee m = a \right) \right\}$$

B.5.1.2. Associated Views

As argued in this paper, a system model consists of a set of views. An element in the system hierarchy is also considered a system of its own, and hence its description would need to consist of a set of views. One such view is its internal view which consists of its composing elements. The other views are constructed from the associations made to that element.

We define the *associated view V_y* of an element e_x in view V_x, to consist of the elements from view V_y that are associated to element e_x (taken across the whole hierarchy of V_y). The elements from view V_y are also said to be in the V_y view of e_x. An associated view of the element is a subset of that view for the complete system since the element is only part of the system.

The views of an element are hence its internal view, as well as the set of associated views. This reinforces our concept of system decomposition into small systems, which themselves have multiple views. The designer of that element need only to look at these views for the analysis of the current status of the design since they summarise all the decisions made so far. However, in extending or specialising these decisions, the designer needs access to the complete views.

Considering the earlier example shown in figure 24 and assuming that *COO* (or one of its children) is further associated with the *Clutch Pedal*, *Throttle Pedal* and the *User Inputs* (of both *Truck* and *Human Interface* functions) communication links, figure 25 illustrates the Function Structure associated view, as well as the internal view (Hardware Structure) of *COO*.

Given the independence of the views, a user can choose to focus on a single view of the whole system and ignore all references made to other views, giving a single perspective of the whole system. On the other hand, a user can take an element with all its internal views and treat it as a complete system with many views.

The relations between the associated elements are also included in the associated view. If two ports of two elements that are in the associated view of e_x, have a connection relation between them, then this connection relation is also in the associated view V_y of e_x. In the example of figure 25, the direct connection relations between the ports of *Operator Inputs* with *Clutch Pedal* and *Throttle Pedal* communication links are included in the associated view. Note that connections between ports can be indirect, which is the case when the ports belong to elements in different parts of the V_y hierarchy. For example, in figure 21, the indirect connection between the port of *Main Controller* and the *User Inputs* communication link is included in the associated view.

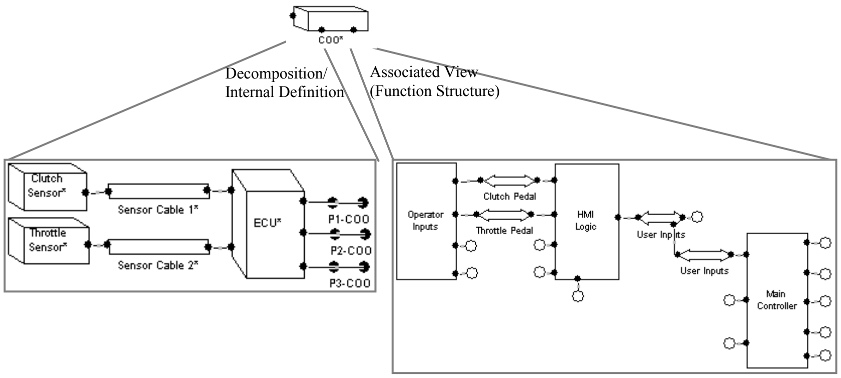


Figure 25. The views of the *COO* hardware unit, consisting of its internal (Hardware Structure) view, as well as the associated Function Structure view.

In the case where there exists a connection relation between two ports and only one of the ports is in the associated view of an element e_x, then it is necessary to indicate that such a connection is missing. This is shown by connecting the existing port to an *associated view interface port*, to indicate that the port needs to connect to other external ports that do not exist in the current (associated) view. In figure 21, an *Operator Inputs*'s port is connected to the *Brake Pedal* communication link, yet *Brake Pedal* is not in the associated view, hence the port is shown as an associated view interface port in figure 25.

The associated view ought to be automatically constructed. Such a mechanism allows a developer to view information in alternative views from its own perspective, defined by its source view (V_x), at a given point in the hierarchy. Note that the elements, ports and relations shown in the associated view V_y of an element e_x are a duplication from the complete view V_y. Changes to these elements are reflected in the complete view V_y. Alternatively, an associated view is only used for visual purposes, and no information ought to be specified in that view. The elements, ports and relations are then considered as 'clones' of the real ones.

B.5.1.3. Validating Element Associations

Naturally, not all associations between elements in different views are permitted. Certain restrictions, which depend on the currently established associations, are imposed.

For element e_x from view V_x to be directly associated to element e_y in view V_y, the following conditions need to be satisfied:

- e_y is not a child of one of the direct associated elements of one of e_x's children.
- Neither e_y, nor any of e_y's parents or children is already directly associated with e_x.

The first condition ensures that associations are specialised down the hierarchy, and that associations do not 'cross-back' up the hierarchy. Referring to figure 24, given that *COO* is directly associated to *Operator Inputs*, it is not possible to specify a direct association between *ECU* (a child of *COO*) and *Human Interface* (*Operator Input*'s parent).

The second condition ensures that direct associations cannot be made to an element as well as its children or parent. Referring to figure 24, given that *COO* is directly associated to *Operator Inputs*, it is not possible to specify a direct association between *COO* and *Pedals* nor *Clutch Pedal Sensor* (Children of *Operator Inputs* down in the hierarchy), nor *Human Interface* (*Operator Input*'s parent).

Formally, the conditions are represented as follows:

$$\begin{aligned}
&\left(E_p(e_y) \cap \bigcup_{n \in E_c(e_x)} A_d(n, V_y) = \varnothing \right) \\
&\wedge \left(e_y \notin A_d(e_x, V_y) \right) \\
&\wedge \left(E_p(e_y) \cap A_d(e_x, V_y) = \varnothing \right) \\
&\wedge \left(E_c(e_y) \cap A_d(e_x, V_y) = \varnothing \right)
\end{aligned}$$

As shown in Appendix C.2, this can be simplified to

$$\begin{aligned}
&\left(E_p(e_y) \cap A_a(e_x, V_y) = \varnothing \right) \\
&\wedge \left(e_y \notin A_d(e_x, V_y) \right) \\
&\wedge \left(E_c(e_y) \cap A_d(e_x, V_y) = \varnothing \right)
\end{aligned}$$

Direct associations are bidirectional meaning that if e_x can be associated to e_y, then e_y should also be associated to e_x. To ensure that this condition is satisfied, the validity check becomes:

$$\begin{aligned}
&\left(E_p(e_y) \cap A_a(e_x, V_y) = \varnothing\right)\\
&\wedge \left(e_y \notin A_d(e_x, V_y)\right)\\
&\wedge \left(E_c(e_y) \cap A_d(e_x, V_y) = \varnothing\right)\\
&\wedge \left(E_p(e_x) \cap A_a(e_y, V_x) = \varnothing\right)\\
&\wedge \left(e_x \notin A_d(e_y, V_x)\right)\\
&\wedge \left(E_c(e_x) \cap A_d(e_y, V_x) = \varnothing\right)
\end{aligned}$$

B.5.1.4. Associating Elements

It may sometimes be desired to find out what elements in view V_y have element e_x as an associated element (direct or inherited). An example of such a need is found in the analysis of section B.6.1.3. We define the *associating elements* of e_x in view V_y, $A_{ai}(e_x, V_y)$, to be such a set. Mathematically, $A_{ai}(e_x, V_y)$ is represented as follows:

$$A_{ai}(e_x, V_y) = \{e_y \in E_y : e_x \in A_a(e_y, V_x)\}$$

Where E_y is the set of elements in view V_y.

Recall that if e_y is an associated element of e_x, it is not necessarily the case that e_x is an associated element of e_y, unless e_x and e_y are directly associated.

Now, rather than searching the entire set of element in V_y, we know that the associating elements of e_x, $A_{ai}(e_x, V_y)$, are constrained to the following subset:

- The elements that have e_x as a direct associated element, $A_d(e_x, V_y)$ (which are the direct associated elements of e_x due to the bidirectionality of element associations).
- For each of the above direct associated elements, their parents up the hierarchy that are also associated to e_x. That is the parents up until, but not including, the parent that is associated to a parent of e_x.

The associating elements of e_x in V_y, $A_{ai}(e_x, V_y)$, can hence be rewritten as:

$$A_{ai}(e_x, V_y) = A_d(e_x, V_y) \cup \left\{ n \in \bigcup_{m \in A_d(e_x, V_y)} E_p(m) : e_x \in A_a(n, V_x) \right\}$$

For example, in figure 24, the associating elements of *Clutch Sensor*, $A_{ai}(\mathit{Clutch\ Sensor}, V_{FS})$, consists of the *Clutch Pedal Sensing* element (its direct associated element), as well as the *Pedals* element (the direct parent of *Clutch Pedal Sensing*). However, the parent *Operator Inputs* is not an associating element to *Clutch Sensor*, since it is associated to the parent of *Clutch Sensor*, namely *COO*.

B.5.1.5. Existence in the Associated View

If neither the element e_x, nor any of its children, have been associated to any element in view V_y, element e_x is defined to be not *exist in associated view V_y*, since from the perspective of view V_y, element e_x simply does not exist.

Element e_x is said to be exist in associated view V_y, $a_{xv}(e_x, V_y)$, if

$$\left(A_d\left(e_x, V_y\right) \neq \varnothing\right) \vee \left(\exists n \in E_c\left(e_x\right): A_d\left(n, V_y\right) \neq \varnothing\right)$$

As shown in Appendix C.3, this is equivalent to

$$A_a\left(e_x, V_y\right) \neq \varnothing$$

$$\text{Notation: } a_{xv}(e_x, V_y) \Leftrightarrow A_a(e_x, V_y) \neq \varnothing$$

For example, consider the association between *Target Sensing* and the *AICC* hardware unit shown in figure 26, noting that none of the children of *Target Sensing* are further associated. In this case, *Signal Processing* is considered to not exist in Hardware Structure associated view, $\neg a_{xv}(\textit{Signal Processing}, V_{HS})$, since it is not associated to any elements in V_{HS}, $A_a\left(\textit{Signal Processing}, V_{HS}\right) = \varnothing$ (V_{HS} denotes the Hardware Structure view).

Note that if an element e_x does not exist in associated view V_y, then none of its children can either, since otherwise the associated elements of e_x would not have been empty in the first place.

$$\neg a_{xv}(e_x, V_y) \Rightarrow \forall n \in E_c(e_x) : \neg a_{xv}\left(n, V_y\right)$$

B.5.1.6. Elementary in Associated View

If the associations of a given element e_x are not further specified by its children, then the element is treated as elementary with respect to the associated view V_y, since it is not possible to further specify the details of the internal elements' associations. In other words, from the perspective of the associated view V_y, the internal elements of e_x, whether e_x is elementary or composite, are not relevant.

We define an element e_x to be *elementary in associated view V_y*, $a_{lv}(e_x, V_y)$, if none of the children of e_x is associated with any elements in view V_y (in other words, none of the children exist in the associated view V_y), yet e_x has associations with at least one element in V_y.

$$\left(\forall n \in E_c\left(e_x\right) : \neg a_{xv}\left(n, V_y\right)\right) \wedge A_d\left(e_x, V_y\right) \neq \varnothing$$

As specified in section B.3.1.4, it would be desired to define *$a_{lv}(e_x, V_y)$* in terms of the direct children of e_x. The above condition can be rewritten as:

$$\left(\forall n \in E_{dc}\left(e_x\right) : \neg a_{xv}\left(n, V_y\right)\right) \wedge A_d\left(e_x, V_y\right) \neq \varnothing$$

Since $\left(\forall n \in E_c\left(e_x\right) : \neg a_{xv}\left(n, V_y\right)\right) \equiv \left(\forall n \in E_{dc}\left(e_x\right) : \neg a_{xv}\left(n, V_y\right)\right)$ as shown in Appendix C.4.

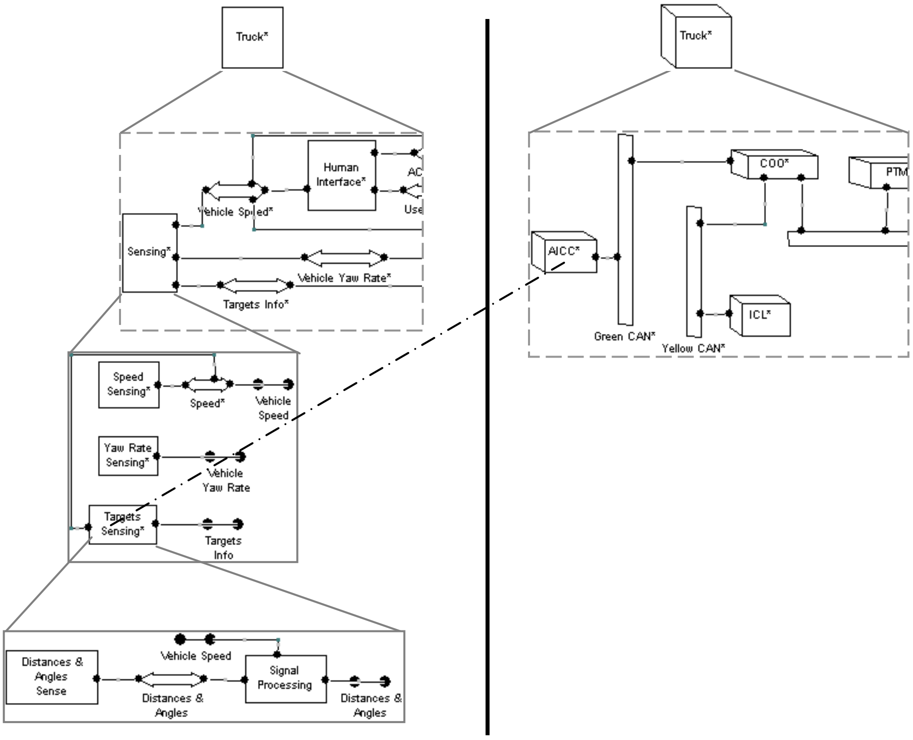


Figure 26. Element association between the *Target Sensing* element and the *AICC* hardware unit.

Note that the definition of e_x as elementary in associated view is only appropriate in the case where e_x exists in associated view V_y.

$$\text{Notation: } a_{lv}(e_x, V_y) \Leftrightarrow \left(\left(\forall n \in E_{dc}\left(e_x\right) : \neg a_{xv}\left(n, V_y\right)\right) \wedge A_d\left(e_x, V_y\right) \neq \varnothing\right)$$

In figure 26, the element *Target Sensing* is considered to be elementary in Hardware Structure associated view, $a_{lv}(\mathit{Target\ Sensing}, V_{HS})$, since none of its children are further associated. On the other hand, in figure 24, the element *Operator Inputs* is considered to be not elementary in Hardware Structure associated view, $\neg a_{lv}(\mathit{Operator\ Inputs}, V_{HS})$, since some of its children, such as *Clutch Pedal Sensing*, are further associated.

B.5.1.7. Completeness Condition

The element association validation checks (section B.5.1.3) ensure that no invalid associations between elements are introduced into the model. However, a given set of valid associations is not necessarily complete, and completeness needs also to be ensured before any analysis of models can be performed. See [17] and section B.6 in this report for a discussion on correctness and completeness.

A feature of the approach described in this report is that associations between elements from different views need not be performed all the way down to the elementary level. For example, in the case where a composite function is to be completely implemented within one hardware unit (composite or elementary), it is sufficient to specify the association between the function and the implementing hardware unit. All sub-functions are implicitly implemented by the same unit. In the case where the hardware unit is a composite, one does not know exactly which sub-unit is to implement which sub-function. This can be considered as a conscious design decision, where, for example, more detailed design is performed externally by a sub-contractor. Nevertheless, the specifications can be considered complete for this function. However, if the association is further refined by one of the sub-functions, it becomes necessary to further specify the allocation of the other sibling sub-functions for a complete specification.

In the example of figure 24, the allocation to the *COO* hardware unit is specified, yet only some of its sub-units (*ECU*, *Clutch Sensor* and *Throttle Sensor*) further specialise this mapping while the mapping of the sub-cables (*Sensor Cable 1* and *Sensor Cable 2*) is not specified. This is hence considered an incomplete allocation specification of *COO*, and needs to be dealt with before any analysis can be performed. A completion of the specification can for example be performed by allocating the *Clutch Pedal* and *Throttle Pedal* Communication links (direct children of *Human Interface*) to these sensor cables.

So, while associations established at the children of an element are appropriately inherited upwards in the hierarchy, associations established at the element can be regarded as requirements on further refinement or specifications of these associations by the children. If the latter associations are not established, the set of associations may be considered incomplete since it cannot be worked out how to further specify the associations on the children elements.

A prerequisite to be able to check for the completeness of associations of an element e_x in view V_y, is that the element e_x exists in associated view V_y, $a_{xv}(e_x, V_y)$. Furthermore, the condition for completeness differs, depending on whether e_x is elementary in associated view V_y or not.

If e_x is elementary in associated view V_y, then e_x is defined to be *completely associated* in V_y, $a_{ca}(e_x, V_y)$.

If e_x is not elementary in associated view V_y, e_x is defined to be completely associated in V_y, $a_{ca}(e_x, V_y)$, if the following conditions are true:

- Each of e_x's direct children exists in associated view V_y.
- For each of e_x's associated elements, $e_a \in A_a(e_x, V_y)$, at least one of e_x's direct children has e_a, or any of its children, as an associated element.

The first condition ensures that if one of the children of e_x exists in associated view V_y (which is the case since e_x is not elementary in associated view V_y), the other children need also to exist in associated view, since it has been established that further refinement of e_x's associations need to be performed, and hence we need to specify each of the children's role in this refinement. The example given above illustrates the need for this condition.

The second condition ensures that any association specified for element e_x is further refined by its children. Considering the example of figure 24, and assuming that the sub-units *Clutch Sensor* and *Throttle Sensor* are not associated to *Clutch Pedal Sensing* and *Throttle Pedal Sensing*, then *COO* is not considered completely associated, since its associated element *Operator Inputs* would not have been specialised by any of *COO*'s direct children.

Note that the conditions above are based on the direct children of element e_x. A precondition for these conditions is that these children have complete associations themselves, which can be specified as a third condition for complete associations.

Formally, if e_x is not elementary in associated view V_y, e_x is said to be completely associated in V_y, $a_{ca}(e_x, V_y)$, if:

$$\begin{aligned} &\forall n \in E_{dc}(e_x) : a_{ca}(n, V_y) \\ \wedge\ &\forall n \in E_{dc}(e_x) : a_{xv}(n, V_y) \\ \wedge\ &\forall n \in A_a(e_x, V_y) : \exists m \in E_{dc}(e_x) : \left(A_a(m, V_y) \cap (n \cup E_c(n)) \neq \varnothing\right) \end{aligned}$$

Note that the association completeness of e_x, does not imply the association completeness of e_x's associated elements, $A_a(e_x, V_y)$. It may be desired to reinterpret the definition of complete association to include the completeness of its associated elements as well. In this case, the following condition is added:

$$\forall n \in A_a(e_x, V_y) : a_{ca}(n, V_x)$$

The condition becomes:

$$\begin{aligned}
&\forall n \in E_{dc}(e_x) : a_{ca}(n, V_y) \\
\wedge\ &\forall n \in E_{dc}(e_x) : a_{xv}(n, V_y) \\
\wedge\ &\forall n \in A_a(e_x, V_y) : \exists m \in E_{dc}(e_x) : \left(A_a(m, V_y) \cap (n \cup E_c(n)) \neq \emptyset\right) \\
\wedge\ &\forall n \in A_a(e_x, V_y) : a_{ca}(n, V_x)
\end{aligned}$$

$$\text{Notation: } a_{ca}(e_x, V_y) \Leftrightarrow \begin{cases} True & \text{if } a_{lv}(e_x, V_y) \\ \\ \forall n \in E_{dc}(e_x) : a_{ca}(n, V_y) & \text{if } \neg a_{lv}(e_x, V_y) \\ \wedge\ \forall n \in E_{dc}(e_x) : a_{xv}(n, V_y) & \\ \wedge\ \forall n \in A_a(e_x, V_y) : \exists m \in E_{dc}(e_x) : & \\ \quad \left(A_a(m, V_y) \cap (n \cup E_c(n)) \neq \emptyset\right) & \\ \wedge\ \forall n \in A_a(e_x, V_y) : a_{ca}(n, V_x) & \end{cases}$$

B.5.1.8. Refined Associated Elements

The associated elements set of an element e_x is based on the direct associations established on that element by the user, as well as any associations inherited from e_x's children.

The associated view, V_y, of element e_x based on these associated elements, $A_a(e_x, V_y)$, provides a fairly high level description of the associations since any refined associations from the children of e_x are not apparent in this view, in the case where a more general association exists.

Given that the children's associations actually refine the associations of e_x, it may be of interest to determine the most refined set of associated elements of e_x. In many cases, only certain children of a specified associated element are effectively associated to e_x (as specified by its children), while other children are associated to another element. This set is referred to as the *refined associated elements* of e_x. It differs from associated elements in that it provides a finer grain set of associated elements. An associated view based on this refined associated set defines a more detailed specification than the associated view as specified in section B.5.1.2.

A prerequisite for establishing the refined associated elements of e_x in view V_y, is that e_x is completely associated in V_y, $a_{ca}(e_x, V_y)$. Furthermore, the refined associated elements set of e_x differs depending on whether e_x is elementary in associated view V_y or not.

If element e_x is elementary in associated view V_y, the refined associated elements of e_x in V_y, $A_{ra}(e_x, V_y)$, is defined as e_x's associated elements.

$$A_{ra}\left(e_x, V_y\right) = A_a\left(e_x, V_y\right)$$

If element e_x is not elementary in associated view V_y, the refined associated elements of e_x in V_y, $A_{ra}(e_x, V_y)$, is defined as the union of the refined associated elements of e_x's direct children, excluding those which have at least one child in the set as well.

$$A_{ra}\left(e_x, V_y\right) = \left\{ a \in \bigcup_{n \in E_{dc}(e_x)} A_{ra}\left(n, V_y\right) : \left(\neg \exists m \in \bigcup_{n \in E_{dc}(e_x)} A_{ra}\left(n, V_y\right) : m \in E_c(a) \right) \right\}$$

$$\text{Notation: } A_{ra}\left(e_x, V_y\right) = \begin{cases} A_a\left(e_x, V_y\right) & \text{if } a_{lv}\left(e_x, V_y\right) \\ \left\{ a \in \bigcup_{n \in E_{dc}(e_x)} A_{ra}\left(n, V_y\right) : \right. & \text{if } \neg a_{lv}\left(e_x, V_y\right) \\ \left. \left(\neg \exists m \in \bigcup_{n \in E_{dc}(e_x)} A_{ra}\left(n, V_y\right) : m \in E_c(a) \right) \right\} & \end{cases}$$

For example, in figure 24, assuming that *COO* is completely associated as suggested in section B.5.1.7, the refined associated elements of the *COO* hardware unit, $A_{ra}(COO, V_{FS})$, consists of *Clutch Pedal Sensing*, *Throttle Pedal Sensing*, *ACC State Machine*, *Distance Control* and *HMI Logic* elements. More elements belong to this set since *COO* and its children are associated to elements not shown in figure 24.

B.5.2. Port Associations

Similar to associations between elements, associations can be specified between the ports across the views. Concerning the case study, in the allocation of functions to hardware units, the association of a function port to a hardware port indicates that the functional communication occurs physically through that hardware port.

For a given element, the association between ports of different views occurs between the element's ports (its interface definition) and the interface ports of the associated view (described in section B.5.1.2). For example, in figure 27, the *COO* hardware unit has three ports in its interface definition connecting to each of the

CAN buses, while its interface in the associated Function Structure view consists of 13 associated view interface ports. So, the function ports of its associated functions (such as port p_3 of element *Operator Inputs* and port p_2 of element *HMI Logic*) need to communicate with their connected ports via one of the three hardware ports.

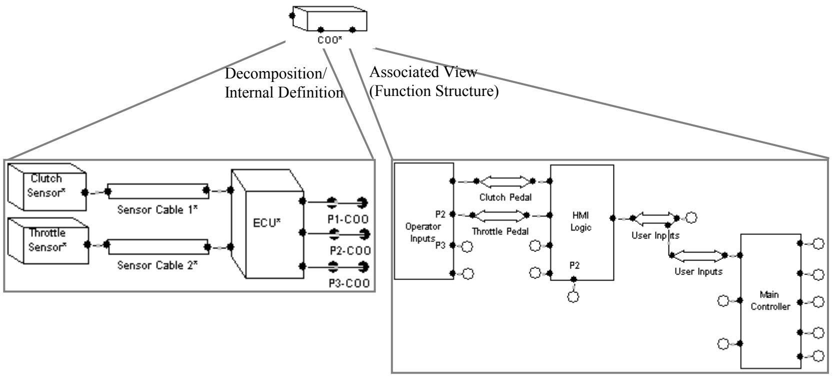


Figure 27. A reproduction of figure 25, highlighting certain port names in the associated view of *COO*, such as p_2 of the *HMI Logic* element.

The *associated ports* of port p_x in view V_y, $A_p(p_x, V_y)$, is defined as the set of associations to ports in V_y, directly specified by the user on port p_x. Port associations are also governed by certain validation and completeness rules. These will be discussed in detail in sections B.5.2.2 and B.5.2.3. In addition to these rules, the following constraint applies for a port p_f (from the Function Structure view) to be associated to port p_h (from the Hardware Structure view):

- p_f can be associated to a maximum of one port from the Hardware Structure view. However, p_h could be associated to any number of ports from the Function Structure view, indicating that more than one communication occur through that same port p_h.

B.5.2.1. The Associated View Interface

As discussed in section B.5.1.2, when viewing the associated view V_y of an element e_x, the relations between the associated elements are also included in V_y. If two ports of two elements that are in the associated view of e_x, have a connection relation between them, then this connection relation is also in the associated view V_y of e_x.

In the case where there exists a connection relation between two ports and only one of the ports, p_y, is in the associated view V_y of e_x, then p_y is said to be not *all* *connected ports associated* in e_x. To indicate that p_y needs to connect to other external ports that do not exist in the associated view V_y of e_x, p_y is connected to an associated view interface port. If all the connected ports of p_y are in the associated view, then p_y needs not interface to any element not associated to e_x, and hence needs not be related to such a port.

We define a port p_y to be all connected ports associated in element e_x, $a_{cpa}(p_y, e_x)$, if all its connected ports, $P_c(p_y)$, (or one of their equivalent ports) have their containing element associated to e_x.

It suffices for one equivalent port of each of the connected ports of p_y to exist in associated view, since a connection to this port implies a connection to all its equivalent ports. A single port from a set of equivalent ports can exist in associated view, given that an associated view cannot contain an element as well as its parent or child element.

Notation: $a_{cpa}(p_y, e_x) \Leftrightarrow \left(\forall p_c \in P_c\left(p_y\right) : \exists p_e \in P_{eq}\left(p_c\right) : e_g\left(p_e\right) \in A_a\left(e_x, V_y\right)\right)$

For example, considering the associations in figure 27, port p_2 of element *Operator Inputs*, $p_{2,OperatorInputs}$, (we denote port p_x of element y as $p_{x,y}$) is an all connected ports associated in element *COO*, $a_{cpa}\left(p_{2,OperatorInputs}, COO\right)$, since all its connected ports, (the port of the communication link *Throttle Pedal*) have their elements also associated to *COO*. On the other hand, port $p_{3,OperatorInputs}$ is not an all connected ports associated in element *COO*, $\neg a_{cpa}\left(p_{3,OperatorInputs}, COO\right)$, since a connected port of $p_{3,OperatorInputs}$, the port of the communication link *Brake Pedal* (see figure 21), does not have its element associated to *COO*.

A precondition to be able to define, $a_{cpa}(p_y, e_x)$, is that the containing element of p_y is an associated element of e_x.

$$e_g\left(p_y\right) \in A_a\left(e_x, V_y\right)$$

B.5.2.2. Port Association Validity Check

In this section, we will incrementally deduce the validity condition for port associations.

First, for a port p_y (of containing element e_y) to be associated to port p_x (of containing element e_x), the following conditions need to be satisfied:

- e_y is an associated element of e_x.
- p_y is not an all connected ports associated in the element e_x.

The first condition simply ensures that the second condition can be validly performed, as required in section B.5.2.1. The second condition ensures that the interface ports of element e_x are associated to ports that need to connect to other external ports that do not exist in associated view V_y of e_x. An all connected ports associated port needs not interface to any element not associated to e_x.

Formally, the condition is represented as follows:

$$e_g(p_y) \in A_a(e_x, V_y) \wedge \neg a_{cpa}(p_y, e_g(p_x))$$

For example, considering the associations in figure 28, port $p_{2,ecu}$ can be associated to port $p_{2,\mathrm{DistanceControl}}$, since:

- The containing element of $p_{2,ecu}$ is associated to the containing element of $p_{2,DistanceControl}$, $ECU \in A_a(Distance\ Control, V_{HS})$;
- And, $p_{2,ecu}$ is not an all connected ports associated in element *Distance Control*, $\neg a_{cpa}(p_{2,ecu}, Distance\ Control)$. This is true since the connected port of $p_{2,ecu}$, $p_{1,SensorCable2}$, is not in the associated view of *Distance Control*.

Similar to element associations, port associations are bidirectional meaning that if p_y can be associated to p_x, then p_x should also be associated to p_y. To ensure that this condition is satisfied, the validity check becomes:

$$\begin{aligned} &(e_g(p_y) \in A_a(e_x, V_y) \wedge \neg a_{cpa}(p_y, e_g(p_x))) \\ \wedge\ &(e_g(p_x) \in A_a(e_y, V_x) \wedge \neg a_{cpa}(p_x, e_g(p_y))) \end{aligned}$$

In the example above, with a similar argument, we can deduce that port $p_{2,DistanceControl}$ can also be associated to port $p_{2,ecu}$. Hence, the association between $p_{2,ecu}$ and $p_{2,DistanceControl}$ remains valid.

Note however that, since elements are associated and inherited across the various hierarchies, it often occurs that element e_y is associated to e_x, yet e_x is not associated to e_y. Hence, guaranteeing the condition for p_y is no guarantee for p_x. The condition may not even be possible to test for p_x if port p_x's element (e_x) is not associated to e_y.

For example, *COO* is in the associated view of *Control* by inheritance. Hence $p_{3,coo}$ can be associated to $p_{1,Control}$ since $COO \in A_a(Control, V_{HS}) \wedge \neg a_{cpa}(p_{3,coo}, Control)$. However, $p_{1,Control}$ cannot be associated to $p_{3,coo}$ since $Control \notin A_a(COO, V_{FS})$. Hence, according the condition above, $p_{3,coo}$ cannot be associated to $p_{1,Control}$ and vice versa.

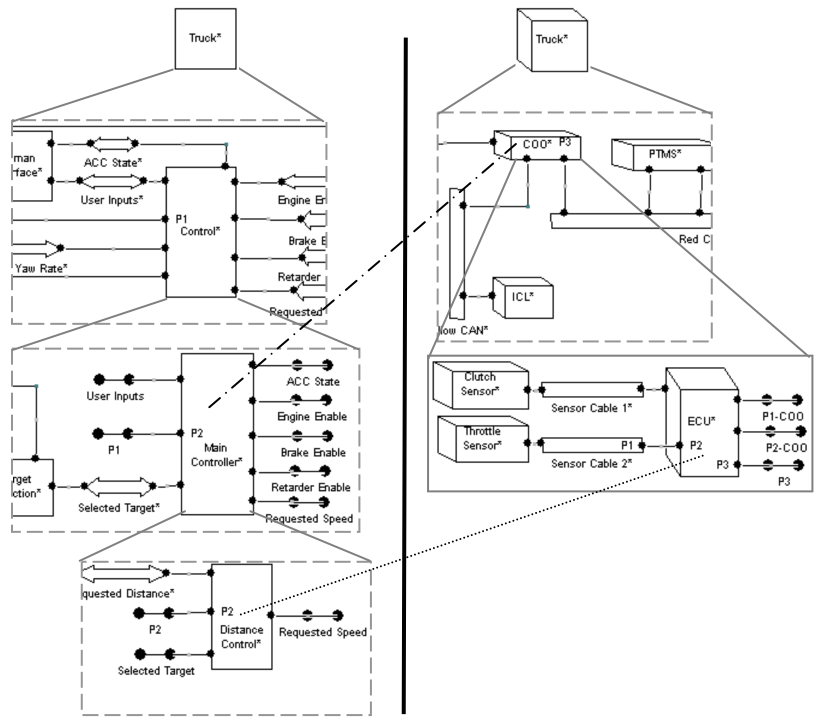


Figure 28. A reproduction of relevant parts from figure 24, focusing on specific direct associations of the hardware unit *COO*, and its child unit *ECU*.

The above example illustrates the case where port p_y is not an all connected ports associated in e_x (satisfying the first part of the condition), but p_x is not even associated to e_y (failing the second part of the condition). Hence, p_x and p_y cannot be associated.

But, in many cases, there may exist an equivalent port of p_x, $p_{x/e}$, which is not all connected ports associated in the associating element e_y. In this case, p_y should be associated to p_x, while $p_{x/e}$ is associated to p_y.

To allow such associations, the validity check changes to become:

$$e_g(p_y) \in A_a(e_x, V_y) \wedge \neg a_{cpa}(p_y, e_g(p_x))$$
$$\wedge\ \exists p_{x/e} \in P_{eq}(p_x):$$
$$e_g(p_{x/e}) \in A_a(e_y, V_x) \wedge \neg a_{cpa}(p_{x/e}, e_g(p_y))$$

With this new condition, and considering the earlier example, $p_{3,coo}$ can be associated to $p_{1,Control}$ (as argued earlier). In addition, the equivalent port of $p_{1,Control}$, $p_{2,MainController}$, can now be associated to $p_{3,coo}$

since $Main\ Controller \in A_a(COO, V_{FS}) \wedge \neg a_{cpa}(p_{2,MainController}, COO)$. Hence, $p_{3,coo}$ is associated to $p_{1,Control}$, and $p_{2,MainController}$ is associated to $p_{3,coo}$.

It is important to remember that upon satisfying this condition, p_y gets associated to p_x, while $p_{x/e}$ (and not p_x) is associated to p_y. In summary, the bidirectionality of associations is extended to allow that if a port p_y is associable to p_x, then p_x, or one of its equivalent ports, can be associated to p_y. This extension should be acceptable since equivalent ports, by definition, are representations of the same properties.

In addition to these rules, equivalent ports that will potentially inherit the associated ports impose further validity conditions that need to be met. This is further discussed in the following subsection.

B.5.2.3. Port Association Inheritance

Equivalent ports must have the same set of associated ports and the rules of inheritance similar to those specified for port properties apply. That is, port associations should be defined on only one port among the set of equivalent ports in order to avoid definition duplications and hence inconsistency problems.

In order to guarantee that for each equivalent port $p_{y/e}$ of p_y that p_x or one of its equivalent ports forms a valid association, the validity check becomes:

$$\begin{aligned}
&e_g(p_y) \in A_a(e_x, V_y) \wedge \neg a_{cpa}(p_y, e_g(p_x)) \\
&\wedge \forall p_{y/e} \in P_{eq}(p_y): \\
&\qquad (\exists p_{x/e} \in P_{eq}(p_x): e_g(p_{y/e}) \in A_a(e_{x/e}, V_y) \wedge \neg a_{cpa}(p_{y/e}, e_g(p_{x/e}))) \\
&\wedge \exists p_{x/e} \in P_{eq}(p_x): \\
&\qquad e_g(p_{x/e}) \in A_a(e_y, V_x) \wedge \neg a_{cpa}(p_{x/e}, e_g(p_y)) \\
&\wedge \forall p_{x/e} \in P_{eq}(p_x): \\
&\qquad (\exists p_{y/e} \in P_{eq}(p_y): e_g(p_{x/e}) \in A_a(e_{y/e}, V_x) \wedge \neg a_{cpa}(p_{x/e}, e_g(p_{y/e})))
\end{aligned}$$

Continuing the previous example, port $p_{3,ecu}$ (an equivalent port of $p_{3,coo}$) can inherit the port association of $p_{2,MainController}$ to $p_{3,coo}$, where an equivalent port of $p_{2,MainController}$, namely $p_{2,DistanceControl}$, is associated to $p_{3,ecu}$ by inheritance, since $Distance\ Control \in A_a(ECU, V_{FS}) \wedge \neg a_{cpa}(p_{2,Dis\tan ceControl}, ECU)$.

Note that the inheritance (and hence the application of the inheritance condition) is only applicable to equivalent ports whose element exist in associated view, since from the associated view perspective, elements that do not exist cannot inherit. The final validity condition becomes:

$$
\begin{aligned}
& e_g(p_y) \in A_a(e_x, V_y) \wedge \neg a_{cpa}(p_y, e_g(p_x)) \\
& \wedge \forall p_{y/e} \in P_{eq}(p_y) : a_{xv}(e_g(p_{y/e}), V_x) : \\
& \qquad (\exists p_{x/e} \in P_{eq}(p_x) : e_g(p_{y/e}) \in A_a(e_{x/e}, V_y) \wedge \neg a_{cpa}(p_{y/e}, e_g(p_{x/e}))) \\
& \wedge \exists p_{x/e} \in P_{eq}(p_x) : \\
& \qquad e_g(p_{x/e}) \in A_a(e_y, V_x) \wedge \neg a_{cpa}(p_{x/e}, e_g(p_y)) \\
& \wedge \forall p_{x/e} \in P_{eq}(p_x) : a_{xv}(e_g(p_{x/e}), V_y) : \\
& \qquad (\exists p_{y/e} \in P_{eq}(p_y) : e_g(p_{x/e}) \in A_a(e_{y/e}, V_x) \wedge \neg a_{cpa}(p_{x/e}, e_g(p_{y/e})))
\end{aligned}
$$

In the above example, the containing element of $p_{3,ecu}$, (*ECU*) exists in the Function Structure associated view, and hence $p_{3,ecu}$ can inherit the association to $p_{2,DistanceControl}$.

As an example of an invalid port association, we return to the association between port $p_{2,ecu}$ and $p_{2,DistanceControl}$ discussed earlier in the previous subsection. Given the new port association validation condition, port $p_{2,DistanceControl}$ can no longer be associated to $p_{2,ecu}$ since for an equivalent port of $p_{2,DistanceControl}$, $p_{2,MainController}$, there exists no equivalent port of $p_{2,ecu}$, to which $p_{2,MainController}$ can be associated by inheritance. As a consequence, port $p_{2,ecu}$ cannot be associated to $p_{2,DistanceControl}$ either.

B.5.2.4. Associable Ports

In summary, we define the *associable ports* of p_x in view V_y, $A_{ap}(p_x, V_y)$, to be the set of ports in V_y that satisfy the port association validity check. These ports can naturally only belong to containing elements that are associated to p_x's containing element. Formally, $A_{ap}(p_x, V_y)$ is represented as follows:

$$
\begin{aligned}
A_{ap}(p_x, V_y) = \Bigg\{ & p_y \in \bigcup_{e_y \in A_a(e_g(p_x))} P_e(e_y) : \\
& e_g(p_y) \in A_a(e_x, V_y) \wedge \neg a_{cpa}(p_y, e_g(p_x)) \\
& \wedge \forall p_{y/e} \in P_{eq}(p_y) : a_{xv}(e_g(p_{y/e}), V_x) : \\
& \qquad (\exists p_{x/e} \in P_{eq}(p_x) : e_g(p_{y/e}) \in A_a(e_{x/e}, V_y) \wedge \neg a_{cpa}(p_{y/e}, e_g(p_{x/e}))) \\
& \wedge \exists p_{x/e} \in P_{eq}(p_x) : \\
& \qquad e_g(p_{x/e}) \in A_a(e_y, V_x) \wedge \neg a_{cpa}(p_{x/e}, e_g(p_y)) \\
& \wedge \forall p_{x/e} \in P_{eq}(p_x) : a_{xv}(e_g(p_{x/e}), V_y) : \\
& \qquad (\exists p_{y/e} \in P_{eq}(p_y) : e_g(p_{x/e}) \in A_a(e_{y/e}, V_x) \wedge \neg a_{cpa}(p_{x/e}, e_g(p_{y/e}))) \Bigg\}
\end{aligned}
$$

B.5.3. Maintaining Model Integrity

The following *actions* can be performed on a model by the user:

- Create and delete elements
- Create and delete ports
- Create, delete and modify properties
- Create and delete relations (interface or connection)
- Create and delete associations (element or port)

Validity checks (such as those described in sections B.5.1.3 and B.5.2.2) prevent any action from invalidating the model. In case the user wishes to perform such a violating action, certain modifications need to be performed prior to the originally intended modification.

The port and element association validity checks guarantee the model validity when attempting to create a new association. This however does not guarantee the validity of established associations at all times.

For example, while the port association validity check prevents invalid port associations, we have not considered other actions that the user can perform that makes existing port associations invalid. In a way, it is so far assumed that port associations are performed once all elements, ports, port relations and element associations are already established, and none will be modified in the future. Such a restriction on the order of performing actions within a model is not desired.

According to the port validity check in section B.5.2.2, a port p_y (of containing element e_y) can no longer be associated to port p_x (of containing element e_x) if one of the following becomes true:

- e_y becomes no longer associated (direct or inherited) to e_x, $e_y \notin A_a(e_x, V_y)$. This may be caused by the following actions:
    a. The direct association between e_y and e_x is deleted.
    b. A parent of e_y is directly associated to e_x, causing e_y to no longer be an inherited associated element of e_x.
- p_y becomes an a l l c o n n e c t e d p o r t s a s s o c i a t e d in e_x, $a_{cpa}(p_y, e_x)$. That is, all the connected ports of p_y become associated to e_x, $\forall p_i \in P_c(p_y) : e_g(p_i) \in A_a(e_x, V_y)$. This may be caused by the following actions:
    a. The containing elements of all connected ports are associated to e_x.

  b. The ports whose elements are not associated to e_x are deleted.
  c. Connection relations to ports whose elements are not associated to e_x are deleted.
  d. Interface relations are deleted, indirectly deleting connections to ports whose elements are not associated to e_x.

- One of p_y's equivalent ports, which exist in associated view V_x, can no longer be associated to p_x or one of its equivalent ports, for similar reasons/actions as above, or if caused by the following action:
  a. An interface relation is created between p_y and another port, creating a new set of equivalent ports to p_y.
- One of p_y's equivalent ports becomes exist in associated view V_x, and the port cannot be associated to p_x or one of its equivalent ports. This may be caused by the following actions:
  a. The port's containing element is associated to an element in V_x.
  b. An interface relation is created between p_y and another port, creating a new set of equivalent ports to p_y.
- Given the bidirectionality of port associations, port p_x can no longer be associated to port p_y for similar reasons/actions as above.

So in principle, any user action that causes the above conditions to be satisfied, should be prevented in order to maintain the model validity.

However, in many cases, such modifications are predictable and hence the mechanism of *induced actions* is introduced, automating the process and modifying the model in order to maintain its validity. These modifications are specified as actions themselves, possibly triggering further actions.

Considering the example of port associations above, actions can be automatically performed in order to re-establish the model integrity, by deleting the existing invalid port associations once any of the above actions are performed. However, in certain cases, it is not possible to perform such induced actions since more than a single option is available to ensure validity. For example, in case where two ports are made equivalent and each of the ports is associated to other ports, it is not possible to decide automatically which of the redundant port specifications ought to be deleted. Such a decision ought to be left to the user instead.

In summary, to keep a model valid when being modified, one of two alternative mechanisms can be adopted:

- *Validity checks* - performed before an action can be taken, that prevent the user from performing certain actions that may jeopardise the model correctness or consistency.
- *Induced automatic actions* - performed as a consequence of a certain user action in order to re-establish the model integrity.

It is not always clear whether to introduce validity checks, preventing invalid actions from occurring, or whether further actions can be induced returning the model to a valid state. Validity checks are simplest to implement since they simply decline the user from performing a certain action unless other actions are performed first, keeping the model correct. Automatic actions, on the other hand, facilitate the work needed to be performed by the user, with the slight risk that the user may be left unaware of any such actions.

The general principle adopted is that induced actions are performed in case there exists a single obvious choice (with obvious consequences) available to the user in order to keep the model valid. In certain situations, restoring validity can be performed in many different ways, and hence a validity check is setup to prevent the action from occurring in the first place and leaving it to the user to make a choice.

As illustrated earlier with port associations, by analysing the dependencies between user actions and the various model aspects (such as element association validity, port association validity, etc.), the consequences of each user action on each of these aspects can be established. For example, a consequence of deleting element e_x from the model is the need to induce the following actions:

- Delete any direct element associations to e_x. This action affects the directly associated element of e_x, $A_d(e_x, V_y)$.
- Re-evaluate the inherited associated elements (and redraw the associated view) of the associating elements of e_x, $A_{ai}(e_x, V_y)$.
- Delete each of the ports of e_x. (This action leads to further induced actions to maintain the validity of port associations, etc.)

In the implemented tool (section B.7), we have systematically defined the consequences of each such user action on the validity of each aspect of the model, and defined the necessary induced actions that need to be performed in order to maintain model validity. These actions can themselves trigger further induced actions. It remains however an effort for future work to formalise these actions.

B.6. Cross-view Analysis

As well as domain-specific analyses that can be performed within a view, certain analyses require information from multiple views, and are hence of interest for the proposed view integration environment. The approach advocated in this paper allows a designer to treat an element of the system as a system of its own, with its own set of views. By allowing the multi-view approach to propagate at each level in the system hierarchies, the same analysis that can be performed at the system level can also be easily performed at the sub-system (element) level.

Three categories of analysis can be identified:

- Correctness analysis
- Completeness analysis
- Keyfigure calculations

Correctness analyses are used to check if any incorrectness or inconsistencies exist in a model. It is generally preferable to perform dynamic correctness checks, detecting and preventing any incorrectness from being introduced into the model as soon as they occur. The validity checks in sections B.5.1.3 and B.5.2.2 are examples of correctness analysis.

Compared to the dynamic correctness checks, certain checks cannot be performed at random instances since not enough information is yet specified by the user to perform the analysis, while the lack of information cannot be flagged as an error. These completeness checks can be triggered by the user once it is believed the model to be complete. The analysis in section B.5.1.7 is an example of a completeness check.

A keyfigure analysis produces a summary of the system properties being modelled. These properties were not specified by the user directly, but emerged from the combination of other properties. Prior to any keyfigure analysis, a completeness check needs to be performed that establishes whether enough information is available for the analysis to be performed. Different keyfigure analyses may require different completeness analyses since a different set of information may be needed.

In [10], the various keyfigure analyses of interest for the design of the EE architecture are discussed. Examples of cross-view keyfigure analyses that can be performed for any element are:

- The number of hardware units and cables needed to realise a given function element.
- The cable length or weight needed for a given function.

- Given a certain function, statistics on the other functions that share some of its resources.

For a given Function or Hardware Structure element, these keyfigure values can be easily calculated based on the associated view of the element. For example, given the associated view in figure 25 of the *COO* hardware unit, one can easily calculate the required utilisation on *COO*, given the execution times and rates of execution of each of the allocated function elements.

The following subsection provides an extended example of cross-view keyfigure analysis relevant for the case study of section B.2.

B.6.1. Complete Cabling Paths for Communication

This analysis checks that any Function Structure element that needs to communicate through their connected Communication Links, can do so, given its specified allocations to hardware units and cables. The analysis can be performed on the complete system, as well as any sub-system (element).

Prior to introducing this analysis, certain terms need to be first defined. For this discussion, the function and hardware unit elements are termed as *container* elements, while the communication link and cable elements are termed as *linker* elements.

B.6.1.1. Internally Linked Ports

The *internally linked ports* of port p, $P_{il}(p)$, is defined as the set of ports of the containing element, $e=e_g(p)$, where $p_x \in P_{il}(p)$ implies that p_x is internally connected to p through a set of internal linker elements only, connected together to form a path from p_x to p.

The $P_{il}(p)$ set differs, depending on the property of e:

- If e is an elementary linker element, $P_{il}(p)$ is the remaining ports of e, since all the element's ports share the internal buffer of the elementary. Considering the Function Structure model in the example of figure 29, $P_{il}(p_{1,CL11}) = \{p_{2,CL11}, p_{3,CL11}\}$.

  $$P_{il}(p) = P_e(e_g(p)) - p$$

- If e is an elementary container element, then there exists no internally linked ports, since e performs a functional transformation between its ports, and not simply a data transfer. In the example of figure 29, $P_{il}(p_{1,F111}) = \varnothing$.

$$P_{il}(p)=\varnothing$$

- If *e* is a composite element, and *p* has no direct interfaced port, $p_{de}(p)$, then there exists no internally linked ports since *p* is not even related to any internal ports of *e* to further link through. In the example of figure 29, $P_{il}(p_{3,CL12})=\varnothing$.

$$P_{il}(p)=\varnothing$$

- If *e* is a composite element, and *p* has a direct interfaced port, $p_{de}(p)$, then

$$P_{il}(p)=P_{el}(p_{de}(p))$$

where the *externally linked ports* of port p_i, $P_{el}(p_i)$, is defined as the set of ports of the parent element, $e_i=e_{dp}(e_g(p_i))$, where $p_y \in P_{el}(p_i)$ implies that p_y is related to p_i through a set of linker elements, connected together to form a path from p_y to p_i. $P_{el}(p_i)$ consists of the union of:

- The direct interfacing port of each of the internally linked ports of p_i.
- The externally linked ports of the direct connected ports of each of the internally linked ports of p_i.

$$P_{el}(p_i)=\bigcup_{n\in P_{il}(p_i)} p_{di}(n) \cup \left(\bigcup_{n\in P_{il}(p_i)} \bigcup_{m\in P_{dc}(n)} P_{el}(m) \right)$$

In the example of figure 29, $P_{il}(p_{1,F1})=\{p_{3,F1},p_{6,F1}\}$. However, $P_{il}(p_{2,F1})=\varnothing$, since the set of linker elements is broken by the direct child of *F11*, namely *F111*.

$$P_{il}(p)=\begin{cases} P_e(e_g(p))-p & \text{if } e_l(e_g(p)) \wedge \text{linker}(e_g(p)) \\ \varnothing & \text{if } e_l(e_g(p)) \wedge \text{container}(e_g(p)) \\ \varnothing & \text{if } \neg e_l(e_g(p)) \wedge p_{de}(p)=nil \\ P_{el}(p_{de}(p)) & \text{if } \neg e_l(e_g(p)) \wedge p_{de}(p)\neq nil \end{cases}$$

Notation:

$$\text{where } P_{el}(p_i)=\bigcup_{n\in P_{il}(p_i)} p_{di}(n) \cup \left(\bigcup_{n\in P_{il}(p_i)} \bigcup_{m\in P_{dc}(n)} P_{el}(m) \right)$$

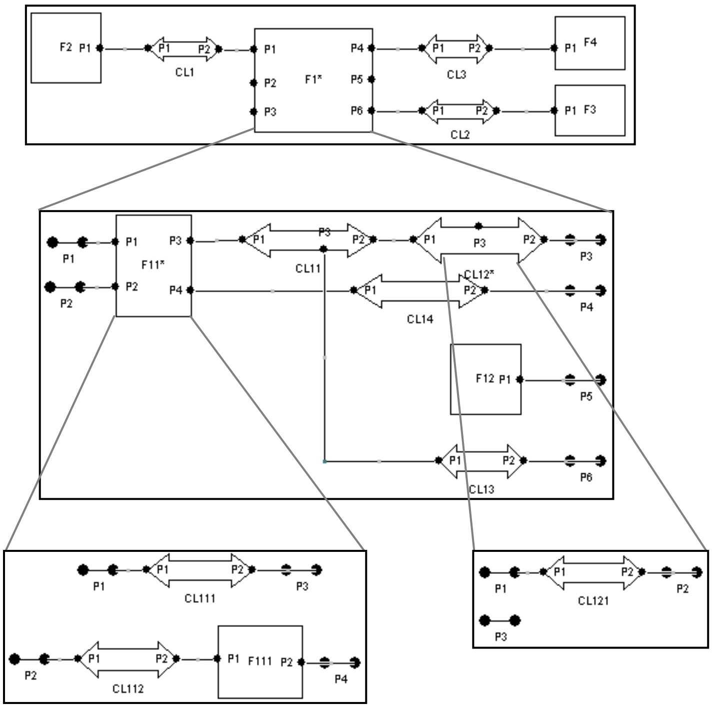


Figure 29. A hypothetical Function Structure model to illustrate internally linked ports.

B.6.1.2. Communicating Ports

Two ports, p_1 and p_2, are defined to be *communicating ports*, $p_{cp}(p_1, p_2)$, if a continuous path of only linker elements exists between them, in which the ports along the path are either directly connected or internally linked.

$p_{cp}(p_1, p_2)$ differs depending on whether p_1 and p_2 are connected or not.

If p_1 and p_2 are connected, then they are said to not be communicating ports, since we expect at least one linker element between p_1 and p_2. In the example of figure 29, the ports $p_{1,F1}$ and $p_{2,CL1}$ are not communicating ports, $\neg p_{cp}\left(p_{1,F1}, p_{2,CL1}\right)$, since $p_{1,F1}$ and $p_{2,CL1}$ are directly connected.

If p_1 and p_2 are not connected, then p_{cp} (p_1, p_2) is true if one of the following is true:

- p_2 is internally linked to p_1. In the example of figure 29, the ports $p_{1,F1}$ and $p_{6,F1}$ are communicating ports, $p_{cp}\left(p_{1,F1}, p_{6,F1}\right)$, since $p_{6,F1} \in P_{il}\left(p_{1,F1}\right)$.

- $\exists p \in P_{il}(p_1)$ such that p is a communicating port with p_2, $p_{cp}(p, p_2)$, or p_2 is connected to p, $p_2 \in P_c(p)$. In the example of figure 29, the ports $p_{1,F1}$ and $p_{1,CL2}$ are communicating ports, $p_{cp}(p_{1,F1}, p_{1,CL2})$, since $\exists p \in P_{il}(p_{1,F1})$, namely $p_{6,F1}$, such that $p_{1,CL2} \in P_c(p_{6,F1})$. Extending this example further, it can be deduced that the ports $p_{1,F1}$ and $p_{2,CL2}$ are communicating ports, $p_{cp}(p_{1,F1}, p_{2,CL2})$, since $\exists p \in P_{il}(p_{1,F1})$, namely $p_{6,F1}$, such that $p_{cp}(p_{6,F1}, p_{2,CL2})$.
- $\exists p \in P_c(p_1)$ such that p is a communicating port with p_2, $p_{cp}(p, p_2)$. In the example of figure 29, the ports $p_{2,CL1}$ and $p_{6,F1}$ are communicating ports, $p_{cp}(p_{2,CL1}, p_{6,F1})$, since $\exists p \in P_c(p_{2,CL1})$, namely $p_{1,F1}$, such that $p_{cp}(p_{1,F1}, p_{6,F1})$ (as discussed earlier, $p_{6,F1} \in P_{il}(p_{1,F1})$).

In summary, $p_{cp}(p_1, p_2)$ is true if

$$\begin{aligned} & p_2 \in P_{il}(p_1) \\ \vee\ & \exists p \in P_{il}(p_1) : \left(p_{cp}(p, p_2) \vee p_2 \in P_c(p)\right) \\ \vee\ & \exists p \in P_c(p_1) : p_{cp}(p, p_2) \end{aligned}$$

As a final example, by combining all these conditions together, and performing the test on ports across the hierarchy, it can be deduced that the ports $p_{1,F2}$ and $p_{1,CL13}$ are communicating ports, $p_{cp}(p_{1,F2}, p_{1,CL13})$.

$$\text{Notation: } P_{cp}(p_1, p_2) = \begin{cases} \text{false} & \text{if } p_2 \in P_c(p_1) \\ \\ p_2 \in P_{il}(p_1) & \text{if } p_2 \notin P_c(p_1) \\ \vee\ \exists p \in P_{il}(p_1) : \left(p_{cp}(p, p_2) \vee p_2 \in P_c(p)\right) & \\ \vee\ \exists p \in P_c(p_1) : p_{cp}(p, p_2) & \end{cases}$$

Two ports, p_1 and p_2, are defined to be *communicating ports in associated view* of element e_x, $p_{cp,av}(p_1, p_2, e_x)$, if they are communicating ports, considering only ports whose containing elements are in the associated view of e_x. Naturally, a precondition for this test is that the containing elements of p_1 and p_2 are associated to e_x.

B.6.1.3. The Complete Cabling Path Analysis

In the current implementation of the analysis, it is assumed that a Function Structure port is associated to a single Hardware Structure port, $\left|A_p(p, V_{hs})\right| = 1$.

The completeness test for this analysis is that the Function Structure element *f* has complete associations, *$a_{ca}(f, V_{hs})$*. Failing this condition implies that there exists missing associations and hence such a cross-view analysis cannot be performed.

The condition for completeness differs, depending on whether *f* is elementary in associated view *V_y* or not.

If *f* is elementary in associated view *V_{hs}*, *$a_{lv}(f, V_{hs})$*, then *f* is defined to have *complete cabling paths for communication*, *$f_{ccp}(f)$*, since all its children are implicitly associated to the same hardware elements, within which the communication occurs internally.

If *f* is not elementary in associated view *V_{hs}*, *$\neg a_{lv}(f, V_{hs})$*, *f* is defined to have complete cabling paths for communication, *$f_{ccp}(f)$*, if the following conditions are satisfied:

- Each port, *p_f*, of each of *f*'s direct children, is associated to a hardware port, if the *p_f*'s associable ports set, *$A_{ap}(p_f, V_{hs})$*, is not empty. A non-empty associable ports set of *p_f* implies that *p_f* itself is not an all connected ports associated in one of the associating elements of *$e_g(p_f)$*, *$A_{ai}(e_g(p_f), V_{hs})$*. *p_f* hence needs to be associated to one of the associable ports in order to communicate to its unassociated connected ports.

$$\forall p_f \in \bigcup_{n \in E_{dc}(f)} P_e(n) : A_{ap}(p_f, V_{hs}) \neq \emptyset :$$
$$A_p(p_f, V_{hs}) \neq \emptyset$$

- For each pair, *p_1* and *p_2*, of directly connected ports of *f*'s direct children that have associations to hardware ports, the pair of associated hardware ports are connected. We need not handle a port that has no associable ports, since its containing element, and that of its directly connected ports (which have also no associable ports), would be associated to the same hardware element, within which the communication occurs internally.

$$\forall p_1, p_2 \in \bigcup_{n \in E_{dc}(f)} P_e(n) : A_p(p_1, V_{hs}) \neq \emptyset \wedge A_p(p_2, V_{hs}) \neq \emptyset \wedge p_2 \in P_{dc}(p_1) :$$
$$\left(A_p(p_1, V_{hs}) \in P_c\left(A_p(p_2, V_{hs})\right)\right)$$

- For each pair, *p_1* and *p_2*, of internally linked ports of *f*'s direct children that have associations to hardware ports, the pair of associated hardware ports are communicating ports in associated view of *f*. It is necessary to make sure that the ports are communicating by only considering the elements and ports of the associated view, to ensure that the element *f* is completely defined using its own set of views, independently of other views and elements in the system.

$$\forall p_1, p_2 \in \bigcup_{n \in E_{dc}(f)} P_e(n) : A_p(p_1, V_{hs}) \neq \varnothing \wedge A_p(p_2, V_{hs}) \neq \varnothing \wedge p_2 \in P_{il}(p_1) : \\ \left(p_{cp,av}\left(A_p(p_1, V_{hs}), A_p(p_2, V_{hs}), f\right)\right)$$

Note that the condition is defined such that it only deals with the direct children of element *f*, with no consideration of the children further down the hierarchy. This definition is in line with the inheritance argument presented in section B.3.1.4. For this reason, the communication completeness check for *f*, does not guarantee the communication completeness of its children. A complete check can be performed by recursively running the same test through the hierarchy.

$$\text{Notation: } f_{ccp}(f) = \begin{cases} \text{true} & \text{if } a_{lv}(f, V_{hs}) \\ \forall p_f \in \bigcup_{n \in E_{dc}(f)} P_e(n) : & \text{if } \neg a_{lv}(f, V_{hs}) \\ \quad A_{ap}(p_f, V_{hs}) \neq \varnothing : & \\ \quad A_p(p_f, V_{hs}) \neq \varnothing & \\ \wedge\ \forall p_1, p_2 \in \bigcup_{n \in E_{dc}(f)} P_e(n) : & \\ \quad A_p(p_1, V_{hs}) \neq \varnothing \wedge A_p(p_2, V_{hs}) \neq \varnothing \wedge p_2 \in P_{dc}(p_1) : & \\ \quad \left(A_p(p_1, V_{hs}) \in P_c\left(A_p(p_2, V_{hs})\right)\right) & \\ \wedge\ \forall p_1, p_2 \in \bigcup_{n \in E_{dc}(f)} P_e(n) : & \\ \quad A_p(p_1, V_{hs}) \neq \varnothing \wedge A_p(p_2, V_{hs}) \neq \varnothing \wedge p_2 \in P_{il}(p_1) : & \\ \quad \left(p_{cp,av}\left(A_p(p_1, V_{hs}), A_p(p_2, V_{hs}), f\right)\right) & \end{cases}$$

For example, consider the simple example in figure 30, showing the associations between the child elements of the *Speed Sensing* function element and the *BMS* hardware unit (See figure 21 and figure 23). In this example, the *Speed Sense* and *Filter* function elements are associated to the *Speed Sensor* and *ECU* child elements of *BMS* respectively.

Now, for the *Speed Sense* element to be able to communicate with *Filter* via the *Speed* communication link, it is necessary to associate *Speed* to the *Sensor Cable* in the hardware view. In addition, the port associations ought to be performed as shown in the figure.

Any other choice of element or port associations would not be satisfactory. For example, it can be easily realised that it would not be acceptable to associate the *Speed* communication link to the *Actuator Cable* of *BMS*. While such an element association is valid and can be performed, no valid port association can thereafter be specified for which the *Speed Sensing* function can have complete cabling paths for communication, f_{ccp}*(Speed Sensing)*.

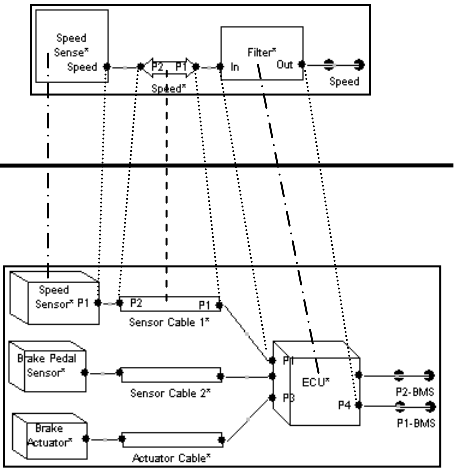


Figure 30. Element and port associations between the child elements of the *Speed Sensing* function element and the *BMS* hardware unit.

Similarly, it would not be acceptable to associate the port $p_{in,Filter}$ to port $p_{3,ecu}$, while ensuring f_{ccp}*(Speed Sensing)*. Such a port association would violate the second condition for path completeness since the port $p_{in,Filter}$ would be associated to a port, $p_{3,ECU}$, which is not connected to the associated port of the connected port to $p_{in,Filter}$, $p_{1,Speed}$. That is, $\left(A_p\left(p_{in,Filter}, V_{hs}\right) \notin P_c\left(A_p\left(p_{1,Speed}, V_{hs}\right)\right)\right)$

Now, consider the more elaborate example in figure 31, showing the associations between the child elements of the *Human Interface* function element and the hardware elements onto which it is desired to implement them. It is desired to establish whether *Human Interface* has complete cabling paths for communication, f_{ccp}*(Human Interface)*. However, the discussion in this section will be limited to the communication path formed by *Operator Inputs*, *Brake Pedal* and *HMI Logic* elements only.

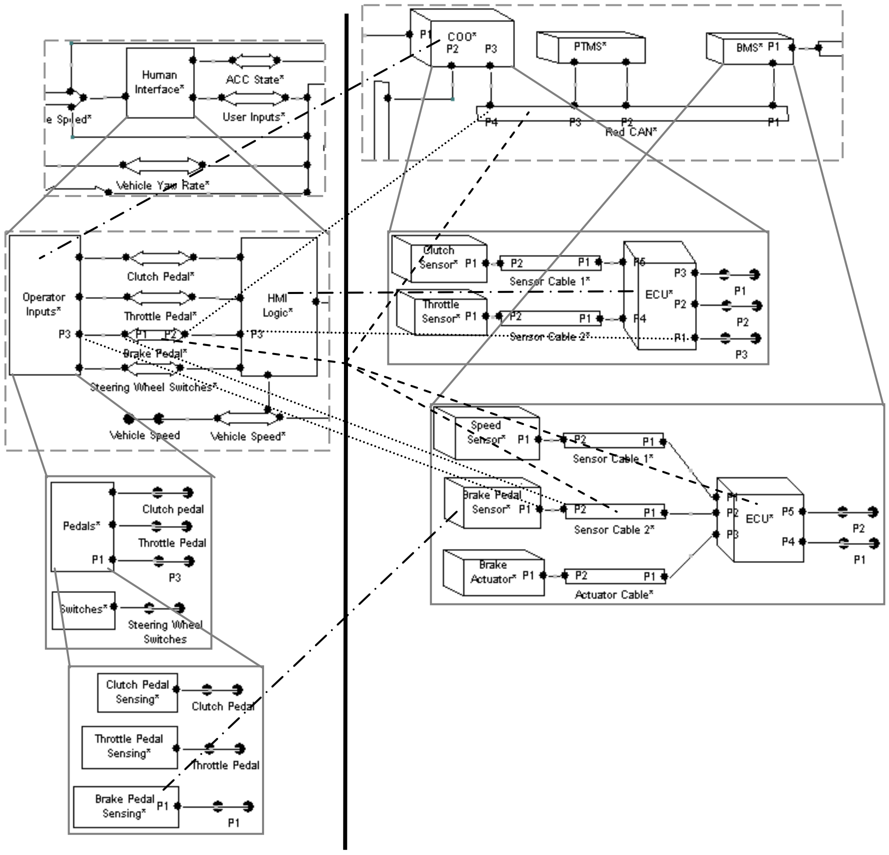


Figure 31. Element and port associations between the child elements of the *Human Interface* function element and the hardware elements onto which it is desired to implement them.

The *Operator Inputs* and *HMI Logic* functions are associated to the *COO* and the *ECU* unit of *COO* (*COO/ECU*) respectively. In addition, the child of *Operator Inputs*, *Brake Pedal Sensing*, is associated to the *Brake Pedal Sensor* hardware unit of the *BMS* hardware unit (*BMS/Brake Pedal Sensor*). This later association also implies that *Operator Inputs* is associated to the *Brake Pedal Sensor* hardware unit by inheritance.

Now, given that port $p_{1,BrakePedalSensing}$ is equivalent to $p_{3,OperatorInputs}$, the only possible association to $p_{3,OperatorInputs}$ would be to $p_{1,BrakePedalSensor}$. Given that restriction, for *Operator Inputs* and *HMI Logic* to be able to communicate via the *Brake Pedal* communication link, *Brake Pedal* needs to be associated to *BMS/Sensor Cable*, *BMS/ECU* as well as *Red CAN*. In this way, a communication path between *BMS/Brake Pedal Sensor* and *COO/ECU* is provided. The Hardware Structure associated view of *Brake Pedal* becomes as shown in figure 32.

In addition, the port associations ought to be performed as shown in the figure. Any other choice of port associations would have not been satisfactory. For example, associating $p_{3,HMILogic}$ to $p_{3,COO/ECU}$ would not satisfy the second condition for path completeness since this port $p_{3,COO/ECU}$ is not connected to $p_{4,RedCAN}$ (the associated port of the connected port to $p_{3,HMILogic}$, $p_{2,BrakePedal}$).

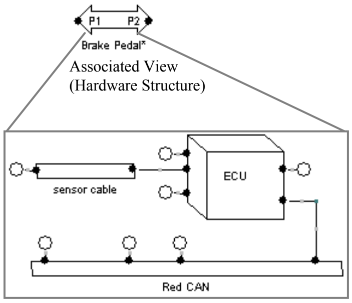


Figure 32. The Hardware Structure associated view of the *Brake Pedal* element.

Finally, consider the internally linked ports $p_{1,BrakePedal}$ and $p_{2,BrakePedal}$. According to the third condition for complete communication paths, the associations to these ports ($p_{2,SensorCable2}$ and $p_{4,RedCAN}$) should be communicating in the associated view of *Human Interface*. But as can be seen in figure 32, this is not the case due to the hardware unit *BMS/ECU*. One remedy to this problem, is to further detail the internal definition of *BMS/ECU*, in which a cable is setup between the ports $p_{2,BMS/ECU}$ and $p_{4,BMS/ECU}$.

B.7. Tool implementation

In order to investigate the feasibility of the inter-view mechanisms introduced in this report, a prototype tool was implemented in the Dome prototyping environment [12], in which views, as well as, inter-view design information and analysis, could be performed.

The integration of views is easier when all views are specified within a single tool. However, different tools are typically used by an organisation to specify the various views of the system. The approach is hence expected to deal with views specified in separate domain-specific tools. A central tool integration and management system can then be used to perform the inter-view information specification and analysis. To prove and test this concept, a partial implementation of the approach has been developed based on the MDM platform [18]. The

Simulink [15] and Dome [12] tools were used for the specification of the Function Structure and Hardware Structure views respectively. A generic inter-view association mechanism is then used to perform element associations between the two tools. The implementation is limited to the associations between elements, while port associations remain the subject of future work.

Some ideas from the suggested solution have also been partly implemented in the industrial analysis tool [10] of the case study of section B.2. The tool is able to evaluate different architectural solutions, based on the keyfigure analysis mentioned in section B.6. The case study presented in this report forms a small subset of the functionality studied in the industrial case study, which covered the complete EE architecture of a set of truck variants. An important contribution of the study was the division of the available dataset into different views, thereby facilitating the desired analysis as well as the possibility to perform multiple allocation strategies without needing to re-model the system functionality. While the implementation is based on our meta-meta-model, the cross-hierarchy associations were not adopted.

B.8. Related Work

The use of the view notion and related concepts (such as viewpoint, model and roles) in high level modelling and framework standards is discussed in [24], concluding that 'in addition to accommodating multiple perspectives, views are used in standards to: examine and define content, expose content to enable interoperability, reduce apparent complexity, provide focus, enable modularity of process, and enforce "need to know" restrictions'. One such standard is the IEEE-1471 [1]. This standard addresses the content and organisation of architectural descriptions of software-intensive systems. In the standard, concepts such as stakeholders, concerns, viewpoint, view and model and the relationships among them, form a fundamental basis for the organisation of these descriptions. No specific views are specified in the standard and although it is specified that consistency among views shall be recorded, how such consistency can be achieved is not specified.

The need to separate the captured design information into different views is gaining increased recognition and is found in many modern engineering modelling languages and tools (such as [2], [3], [4] and [5]). In addition, most modelling approaches adopt some form of decomposition techniques in describing each of the supported views [19]. In combining these two techniques, it becomes essential to integrate the various hierarchical views, through the specification of inter-view design information, in order to form a consistent and complete system definition.

When integrating the system views, modelling approaches (such as [20], [21], [22], [23] and [3]) normally provide the simple mechanism to reference a component from one view to another component in another view. For example, it may be possible to specify the software components in the software view that are to be allocated to a specific processor in the hardware view. Many of these approaches only allow the establishment of relationships at the leaf of their hierarchies ([22], [23] and [3]). In this way, the complexity of interrelating the system views across their hierarchies is simply avoided. However, the advantages gained in using hierarchical descriptions within a view are then lost during view integration, forcing developers to work at the lowest levels of abstractions.

In the few cases where references can be specified across the hierarchies (such as [20] and [21]), the semantics of such references are restricted to the context of the specific system part at which they are specified. Views are hence only loosely tied at the points at which the references are specified. It would instead be desired to obtain a tighter integration by propagating these references across the system hierarchies. For example, having specified the allocation of certain software components onto hardware components, mechanisms ought to be provided that use this information to facilitate the more refined allocation of software to hardware at a more detailed level of abstraction of the system.

From the software engineering domain, the work presented in [16] also deals with the documentation of software architectures, in which the concept of views plays a central role. The work categorises a specific set of views found in common use. Similar to the meta-meta-model suggested in this report, in describing each view, the set of elements, relations, their properties and a topology that can be defined in the view are described. The views are grouped into different styles, which are themselves grouped into viewtypes forming a hierarchy. For each view, the relationships to other views across this hierarchy are described, by stating the relations between the different elements in the views to each other. While stating that certain relations may be quite complex (such as the allocation of modules to components), no guidelines are given on how this complexity should be handled.

In [25], an environment in which domain-specific components can be composed to develop large applications is presented. The approach recognises that since domains are developed independently, they may contain similar concepts defined in different ways; and domain composition needs to identify and define relations between these concepts. Two types of relations can be established: general associations and correspondence relating similar or overlapping concepts. The approach is model-based in that components are modelled in different domains, using domain-specific languages, and the composition is performed at the model level before code generation is performed. The approach is focused on software applications where each component/domain results in source code that need to be integrated. While the approach deals with system decomposition into different domains, the decomposition mechanisms within each domain are not considered.

Aspect Oriented Programming (AOP) [26] is another approach within the software engineering community where a system specification is separated between its functional components and its other properties that affect the system semantics and performance. AOP deals with the cross-cutting of the hierarchical decomposition of a system into components, with the various non-functional aspects of the system such as its error handling and performance aspects. This cross-cutting is necessary since the aspects must compose differently from the functional decomposition, yet the different compositions must be coordinated. An aspect weaver is then used to integrate and coordinate the co-composition of the aspects with the functional components. In this approach, while the functional decomposition is hierarchical, the remaining aspects are not.

A framework and a set of techniques for the view integration of the existing views in UML with other architectural views is presented in [27]. The framework allows the mapping of architectural components/connectors to the classes of the design view. This mapping is closely related to the hardware to functionality allocation approach discussed in this report. However, the suggested mapping deals with a flat structure in each view, and assumes that a design class can only be mapped to a single architectural element. In addition, once the mapping is performed, conformance analysis can be automated in order to identify mismatches between the architectural view of a system and its design view, based on a set of constraints rules. For example, it becomes possible to check that class interactions belonging to different components are appropriately constrained to the architectural topology adopted. Such analysis is similar to the correctness and completeness check analysis presented in section B.6.1.

B.9. Conclusion

In this paper, the need for a systematic approach to multi-view integration is discussed. The establishment of inter-view design information is common practice in many modern design tools. The approach presented here takes advantage of such information in order to tightly interweave the views' hierarchies. In this way, the system views are reflected to a stakeholder within a given domain at a sufficient level of abstraction and detail that makes him/her appreciate the information provided.

Through the use of a case study, model integration is investigated for a particular type of inter-view relationships (function to hardware allocation). The resulting approach maintains the principle of hierarchical design within, as well as between the views, by systematically integrating the two generally accepted complexity reduction techniques of hierarchical decomposition and multi-viewing. Rules and mechanisms were developed to ensure the completeness and correctness of any inter-view design decisions. Additional mechanisms allow a developer within a given domain to view the other aspects of the system from his/her own perspective, making view integration a good basis for information sharing. The proposed approach promotes the independent development of the views, allowing developers from each discipline to work concurrently, yet providing support for a holistic view.

Allocation is strongly related to the design process and can of course be carried out in different ways. The defined allocation inheritance rules permit the specialisation (refinement) of allocation specifications performed higher up in the hierarchies, as well as their extensions at the lower levels, propagating the extended associations up to the higher levels. Such mechanisms support a process-independent allocation practice. By placing certain restrictions, the allocation practices can be constrained. For example, disallowing the possibilities for association extensions through the sub-systems provides a top-down approach, where sub-system design can only refine design decisions specified at the higher level.

The approach also reinforces the principle that a part of the complete system is a system of its own, with its own set of views. This provides the possibilities to perform cross-view analysis on the complete system as well as its individual parts, since all relevant inter-view relationships established across the system are propagated.

To investigate the approach's feasibility, various tool implementations were performed. Less focus has so far been placed on scalability and implementation efficiency considering many views and large systems. Future developments would need to address these issues appropriately.

Even though it is based on simple concepts, using the approach is suspected to require a new mind-set. This places certain doubts on whether the approach actually facilitates the developer's work. From the limited gained experiences, the ability to focus on specific parts of the system design, as well as inheriting and extending other decisions made elsewhere in the system, is rewarding. This however does depend on good feedback and support by the integration tool. In the worst case, the approach advocated here can be seen as an experiment, or an initial step, towards other possibilities of view integration.

While specific to the allocation of system functions to hardware, it is believed that the mechanisms can be applied to other types of relationships such as that of mapping software components to hardware. No claim can be made that these mechanisms are general enough to handle all types of relationships. However, it is intended to expand on this work in order to cover many of the relationships identified in [19] such as dependencies and refinement. In addition, the ability to perform inter-view associations over a larger number of views is a challenge to handle in future developments.

A systematic approach when implementing these relationships should allow a reuse of many of the concepts already explored. What is essential is to provide mechanisms that reflect design decisions between design teams from the various disciplines, and across the different levels of abstractions. This provides a good basis for an information sharing environment enabling model-based, multidisciplinary development.

B.10. Acknowledgements

This work has been supported by the Swedish Strategic Research Foundation, through the SAVE project. Special thanks go to Martin Törngren for his reviews of this report, as well as Ola Larses for his reviews and help in the Scania case study.

B.11. References

[1] IEEE, ANSI/IEEE Standard 1471-2000, "Recommended practice for architectural description of software-intensive systems", September 2000.

[2] UML, OMG Unified Modelling Language Specification, V1.5, March 2003.

[3] Kruchten, P. B. "The 4+1 View Model of Architecture" IEEE Software, Volume 12, Issue 6 November 1995, pp 42-50.

[4] GME, "A Generic Modelling Environment, GME 4 User's Manual" Version 4.0, Institute for Software Integrated Systems, Vanderbilt University, 2004.

[5] Redell O., El-khoury J. and Törngren M., "The AIDA toolset for design and implementation analysis of distributed real-time control systems" Microprocessors and Microsystems. Volume 28, Issue 4, 20 May 2004, Pages 163-182.

[6] Grundy J., Hosking J. and Mugridge W.B., "Inconsistency management for multiple-view software development environments", Software Engineering, Volume 24, Issue 11, 1998.

[7] Skyttner L. General Systems Theory: Ideas and Applications. World Scientific Publishing Co. Singapore. ISBN 981-02-4175-5. 2001.

[8] Weinberg G. M., An Introduction to General Systems Thinking. Dorset House Publishing; Silver anniversary edition, 2001, ISBN 0932633498.

[9] Larses O. and Adamsson N. "Drivers for Model Based Development" Proceedings of the 8th International Design Conference on Design, Dubrovnik, May 2004.

[10] Larses, O., “Applying quantitative methods for architecture design of embedded automotive systems”, Proceedings of INCOSE International Symposium, 2005.
[11] MOF, Meta Object Facility (MOF) Specification, V1.4, April 2002.
[12] Dome, “Dome Guide” Version 5.2.2, http://www.htc.honeywell.com/dome/index.htm, 1999, accessed November 2005.
[13] Råde L. and Westergren B., “Beta Mathematics Handbook”, second edition, Chartwell-Bratt Ltd, ISBN 0-86238-140-1, 1990.
[14] Cooling J., Software Engineering for Real-time Systems. Pearson Education Limited, ISBN 0201596202, 2003.
[15] Simulink, Mathworks, http://www.mathworks.com/products/simulink/, accessed November 2005.
[16] Clements P., Bachman F., Bass L., Garlan D., Ivers J., Little R., Nord R. and Stafford J., “Documenting software architectures: Views and beyond”, Addison Wesley, ISBN 0-201-70372-6, 2002.
[17] Maier M. W., Emery D. and Hilliard R., “ANSI/IEEE 1471 and systems engineering” Systems Engineering. Volume 7, Issue 3, pp 257-270, 2004.
[18] El-khoury J., Redell O. and Törngren M., “A tool integration platform for multi-disciplinary development”, 31st Euromicro Conference on Software Engineering and Advanced Applications, 2005.
[19] El-khoury J., Chen D. and Törngren M., “A survey of modelling approaches for embedded computer control systems (Version 2.0)” Technical report, ISRN/KTH/MMK/R-03/36-SE, TRITA-MMK 2003:36, ISSN 1400-1179, Department of Machine Design, KTH, 2003.
[20] Core, Vitech Corporation, http://www.vtcorp.com/core/productline.html/, accessed November 2005.
[21] Herzog, E. and Törne, A., “Information modelling for system specification representation and data exchange” Proceedings of the Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, pp 136 – 143, April 2001.
[22] Hatley D., Hruschka P. and Pirbhai I., Process for system architecting and requirements engineering. Dorset House, New York, 2000.
[23] Loureiro G., Leaney P. G. and Hodgson M., “A systems engineering framework for integrated automotive development” Systems Engineering. Volume 7, Issue 2, pp: 153-166, 2004.
[24] Martin R. and Robertson E., “Views in the enterprise domain”, Views, Aspects and Roles Workshop, 2005.
[25] Estublier J., Ionita A. D. and Vega G., “A domain composition approach”, International Workshop on Applications of UML/MDA to Software System, 2005.

[26] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C. V., Loingtier J. M., Irwin J., "Aspect-Oriented Programming", European Conference on Object-Oriented Programming (ECOOP), 1997.
[27] Egyed A. and Medvidovic N., "Extending architectural representation in UML with view integration", 2nd International Conference on the Unified Modelling Language (UML), 1999.
[28] Papadopoulos Y. and Grante C, Techniques and tools for automated safety analysis & decision support for redundancy allocation automotive systems, 27th Annual International Computer Software and Applications Conference, 2003

Appendix

Appendix A Terminology

A.1 Single-view Modelling

analysis view – A view used to present specific aspects from the set of design views in a certain way that facilitates the performance of an certain analysis.

attributes - A placeholder used to represent a single property of an element, port or relation.

child element – of element e_x is an element lower down in e_x's hierarchy, forming a part of e_x's internal definition. There may exist more than one child element of e_x.

composite element – A more elaborate description of an element where the properties of the system are decomposed into smaller, less complex, interacting elements, in which each element contains a subset of the original system properties.

connected ports - of port p_x, $P_c\ (p_x)$, is the set of direct connected ports of p_x and each of their equivalent ports, together with the direct connected ports of the equivalent ports of p_x.

connection relation – a relation established between a port of an element and a port of another peer element, implying a certain dependency between their properties.

containing element – of a port p_x, $e_g(p_x)$, is the element for which the port presents an interface.

design view – a view used to model and document the design decisions made by developers.

direct child element – of element e_x is a child element of e_x which exists directly one level down in e_x's hierarchy. There may exist more than one child element of e_x.

direct connected port - of port p_x is the port in a connection relation with p_x. There may exists more than one direct connected port of a single port p_x.

direct interfaced port – of port p_x, $p_{de}(p_x)$, is the port of the internal element in which p_x is a direct interfacing port. There may only be one direct interfaced port of a port p_x.

direct interfacing port – of port p_x, $p_{di}(p_x)$, is the port in an interface relation, in which p_x is a port of an internal element. There may only be one direct interfacing port of a port p_x.

direct parent element – of element e_x, $e_{dp}(c)$, is a parent element of e_x which exists directly one level up in e_x's hierarchy. There exists a maximum of one direct parent of e_x.

direct properties – of a port p_x are properties defined directly on it by the user.

element – a placeholder of properties describing the represented system

elementary element - element e_x is defined to be *elementary*, $e_l(e_x)$, if e_x contains no child elements. e_x has a simple description where the properties can be specified as a set of attributes.

equivalent ports - of a port p_x, $P_{eq}(p_x)$, is the combined sets of its interfacing ports and interfaced ports, as well as p_x itself.

inherited properties – of a port p_x are properties defined through one of p_x's equivalent ports (the inheriting equivalent port of p_x).

inheriting equivalent port – of a port p_x is the equivalent port of p_x in which the properties are directly defined.

interface (external) definition – of element e_x reveals only those properties of e_x that need to be shared with the system environment.

interface relation - a relation between an element's port and a port of one of its internal elements, externally indicating that the internal port is externally accessible.

interfaced ports – of port p_x, $P_e(p_x)$, is the direct interfaced port of p_x, together with its interfaced ports.

interfacing ports – of port p_x, $P_i(p_x)$, is the direct interfacing port of p_x, together with its interfacing ports.

internal (white-box) definition – of element e_x deals with e_x's complete set of properties, which consists of its set of internal elements.

internal element – see child element

parent element – of element e_x is the composite element higher up in e_x's hierarchy, in which e_x is a child element. There may exist more than one parent element of e_x.

port – forms part of the interface definition of its containing element and acts as a placeholder for a subset of its element's externally accessible properties. Two representations of a port can be defined: an *internal port representation* which is a representation of the port as seen from the containing element's internal definition; an *external port representation* which is a representation of the port as seen from the containing element's interface definition.

property placeholder – an element or a port.

root element – of view V_x, $e_r(V_x)$, is the single element within V_x which has no parent elements.

A.2 Two-View Integration

all connected ports associated - port p_y is defined to be all connected ports associated in element e_x, $a_{cpa}(p_y, e_x)$, if all its connected ports, $P_c(p_y)$, (or one of their equivalent ports) have their containing element associated to e_x.

associable ports – of port p_x in view V_y, $A_{ap}(p_x, V_y)$, is the set of ports in V_y that satisfy the port association validity check, and can hence be associated to p_x.

associated elements - of element e_x in view V_y, $A_a(e_x, V_y)$, consists of the union of its direct associated elements and its inherited associated elements.

associated ports – of port p_x in view V_y, $A_p(p_x, V_y)$, is the set of associations to ports in V_y, directly specified by the user on port p_x.

associated view - V_y of element e_x in view V_x is a subset of the complete view V_y for the complete system. It consists of the elements from view V_y that are associated to element e_x (taken across the whole hierarchy of V_y).

associated view interface port – of port p_y is an interface port to p_y, presented in the associated view of element e_x, in the case where p_y is not an all connected ports associated port, indicating that certain connections to p_y are missing in the associated view.

associating elements - of element e_x in view V_y, $A_{ai}(e_x, V_y)$, is the set of elements in view V_y have element e_x as an associated element (direct or inherited).

association - a relation between property placeholders across different views

completely associated – element e_x is defined to be completely associated in view V_y, $a_{ca}(e_x, V_y)$, if given the set of associated elements specified for e_x, no further refinement of these associations are needed by e_x's children in order to complete the system specification.

direct associated elements - of element e_x in view V_y, $A_d(e_x, V_y)$, is the set of associations to elements in V_y, directly specified by the user on element e_x.

elementary in associated view - element e_x is defined to be elementary in associated view V_y, $a_{lv}(e_x, V_y)$, if none of the children of e_x is associated with any elements in view V_y, yet e_x has associations with at least one element in V_y.

exist in associated view - element e_x is defined to be exist in associated view V_y, , $a_{xv}(e, V_y)$, if either e_x, or one of its children, have been associated to at least one element in view V_y.

inherited associated elements - of element e_x in view V_y, $A_i(e_x, V_y)$, is the set of (top most) direct associated elements of e_x's children, excluding those which have already been defined, or generalised, through the direct associated elements of e_x, $A_d(e_x, V_y)$.

refined associated elements – of element e_x in view V_y, $A_{ra}(e_x, V_y)$, is the most refined set of associated element of e_x, based on the associated elements if e_x's direct children.

A.3 Example Views - Function structure and Hardware Structure

cable – an element designating a physical cable with a certain geometrical path.

communicating ports - Two ports, p_1 and p_2, are defined to be communicating ports, $p_{cp}(p_1, p_2)$, if a continuous path of purely linker elements exists between them, in which the ports along the path are either directly connected or internally linked.

communicating ports in associated view - Two ports, p_1 and p_2, are defined to be communicating ports in associated view of element e_x, $p_{cp,av}(p_1, p_2, e_x)$, if they are communicating ports, considering only ports whose containing elements are in the associated view of e_x.

communication link – an element designating a link that transports data between functions.

complete cabling paths for communication – the Function Structure element f is defined to have complete cabling paths for communication, $f_{ccp}(f)$, if all of f's direct children can communicate to each other through their connected communication links, given their associated hardware units and cables.

container element – a function or hardware unit element.

function – an element designating certain functionality that given a certain input, produces a certain output.

hardware unit – an element designating a physical block occupying a certain amount of space.

internally linked ports - of port *p*, $P_{il}(p)$, is the set of ports of the containing element that are internally connected to *p* through a set of internal purely linker elements, connected together to form a path from to *p*.

linker element – a communication link or cable element.

Appendix B Notations

$a_{ca}(e_x, V_y)$	element e_x is completely associated in view V_y
$a_{cpa}(p_y, e_x)$	port p_y is all connected ports associated in element e_x
$a_{lv}(e_x, V_y)$	element e_x is elementary in associated view V_y
$a_{xv}(e_x, V_y)$	element e_x is exist in associated view V_y
$A_a(e_x, V_y)$	Associated elements of element e_x in view V_y
$A_{ai}(e_x, V_y)$	associating elements of element e_x in view V_y
$A_{ap}(p_x, V_y)$	Associable ports of port p_x in view V_y
$A_d(e_x, V_y)$	direct associated elements of element e_x in view V_y
$A_i(e_x, V_y)$	inherited associated elements of element e_x in view V_y
$A_p(p_x, V_y)$	Associated ports of port p_x in view V_y
$A_{ra}(e_x, V_y)$	refined associated elements of element e_x in view V_y
$e_{dp}(e_x)$	direct parent element of element e_x
$e_g(p_x)$	Containing element of port p_x
$e_l(e_x)$	element e_x is elementary
$e_r(V_x)$	root element of view V_x
$E_{dc}(e_x)$	direct children elements of element e_x
$E_p(e_x)$	Parent elements of element e_x
$E_c(e_x)$	Children elements of element e_x
$f_{ccp}(f)$	the Function Structure element f has complete cabling paths for communication
$p_{cp}(p_1, p_2)$	ports, p_1 and p_2, are communicating ports
$p_{cp,av}(p_1, p_2, e_x)$	ports, p_1 and p_2, are communicating ports in associated view of element e_x
$p_{de}(p_x)$	Direct interfaced port of port p_x
$p_{di}(p_x)$	Direct interfacing port of port p_x
$p_{x,e}$	port p_x of element e

$P_c\ (p_x)$	Connected ports of port p_x
$P_{dc}(p)$	Direct connected ports of port p_x
$P_e(p_x)$	interfaced ports of port p_x
$P_e(e_x)$	ports of element e_x
$P_{eq}(p_x)$	Equivalent ports of a port p_x
$P_{el}(p_x)$	externally linked ports of port p_x
$P_i(p_x)$	Interfacing ports of port p_x
$P_{il}(p_x)$	internally linked ports of port p_x
V_{FS}	Function Structure view
V_{HS}	Hardware Structure view

Appendix C Proofs

C.1 Proof 1

Let

$$A_i(e_x,V_y)=\left\{a\in\bigcup_{n\in E_c(e_x)}A_d(n,V_y):\left(\neg\exists m\in\bigcup_{n\in E_c(e_x)}A_d(n,V_y):m\in E_p(a)\right)\right.$$
$$\left.\wedge\left(\neg\exists m\in A_d(e_x,V_y):m\in E_p(a)\vee m=a\right)\right\} \quad [1]$$

$$A_a(e_x,V_y)=A_i(e_x,V_y)\bigcup A_d(e_x,V_y) \quad [2]$$

And

$$B_i(e_x,V_y)=\left\{a\in\bigcup_{n\in E_{dc}(e_x)}B_a(n,V_y):\left(\neg\exists m\in\bigcup_{n\in E_{dc}(e_x)}B_a(n,V_y):m\in E_p(a)\right)\right.$$
$$\left.\wedge\left(\neg\exists m\in A_d(e_x,V_y):m\in E_p(a)\vee m=a\right)\right\} \quad [3]$$

$$B_a(e_x,V_y)=B_i(e_x,V_y)\bigcup A_d(e_x,V_y) \quad [4]$$

We need to prove that

$$\left(A_i(e_x,V_y)=B_i(e_x,V_y)\right)\wedge\left(A_a(e_x,V_y)=B_a(e_x,V_y)\right) \quad [5]$$

1. Considering all the elementary elements e_x of the model tree, M, [5] is true since $E_{dc}(e_x)\equiv E_c(e_x)\equiv\varnothing$

Hence,

$$\forall e_x\in\{a\in E:E_{dc}(a)=\varnothing\}:\left(\left(A_i(e_x,V_y)=B_i(e_x,V_y)\right)\wedge\left(A_a(e_x,V_y)=B_a(e_x,V_y)\right)\right) \quad [6]$$

2. Considering the nodes of the M tree one level up in the hierarchy (that is $\{e_x\in E:\forall n\in E_{dc}(e_x):E_{dc}(n)=\varnothing\}$), [5] is true since $E_{dc}(e_x)\equiv E_c(e_x)$.

Hence,

$$\forall e_x\in\{a\in E:E_{dc}(a)=E_c(a)\}:\left(\left(A_i(e_x,V_y)=B_i(e_x,V_y)\right)\wedge\left(A_a(e_x,V_y)=B_a(e_x,V_y)\right)\right) \quad [7]$$

3. Now, assume that for a given e_{x2}, $\forall e_{x1}\in E_c(e_{x2})$, condition [5] is true.

That is:

$$\forall e_{x1} \in E_c(e_{x2}): \left(\left(A_i(e_{x1},V_y) = B_i(e_{x1},V_y)\right) \wedge \left(A_a(e_{x1},V_y) = B_a(e_{x1},V_y)\right)\right) \qquad [8]$$

Given this assumption, we now proof the condition true for *e_{x2}* itself.

$$B_i(e_{x2},V_y) = \left\{ a \in \bigcup_{n\in E_{dc}(e_{x2})} B_a(n,V_y) : \left(\neg\exists m \in \bigcup_{n\in E_{dc}(e_{x2})} B_a(n,V_y) : m \in E_p(a)\right) \wedge C \right\}$$

Where $C = \left(\neg\exists m \in A_d(e_{x2},V_y) : m \in E_p(a) \vee m = a\right)$

$$B_i(e_{x2},V_y) = \left\{ a \in \bigcup_{n\in E_{dc}(e_{x2})} \left(A_i(n,V_y) \bigcup A_d(n,V_y)\right) : \left(\neg\exists m \in \bigcup_{n\in E_{dc}(e_{x2})} \left(A_i(n,V_y) \bigcup A_d(n,V_y)\right) : m \in E_p(a)\right) \wedge C \right\}$$

[From [8], since $\forall n \in E_{dc}(e_{x2}) : B_a(n,V_y) = A_a(n,V_y) = A_i(n,V_y) \bigcup A_d(n,V_y)$]

$$B_i(e_{x2},V_y) = \left\{ a \in \bigcup_{n\in E_{dc}(e_{x2})} A_i(n,V_y) \bigcup \bigcup_{n\in E_{dc}(e_{x2})} A_d(n,V_y) : \left(\neg\exists m \in \bigcup_{n\in E_{dc}(e_{x2})} A_i(n,V_y) \bigcup \bigcup_{n\in E_{dc}(e_{x2})} A_d(n,V_y) : m \in E_p(a)\right) \wedge C \right\}$$

$$\begin{aligned} B_i(e_{x2},V_y) = \Bigg\{ a \in \Bigg(&\bigcup_{n\in E_{dc}(e_{x2})} \Big\{ b \in \bigcup_{m\in E_c(n)} A_d(m,V_y) : \\ &\left(\neg\exists p \in \bigcup_{m\in E_c(n)} A_d(m,V_y) : p \in E_p(b)\right) \wedge C \Big\} \Bigg) \bigcup \bigcup_{n\in E_{dc}(e_{x2})} A_d(n,V_y) : \\ &\Bigg(\neg\exists m \in \Bigg(\bigcup_{n\in E_{dc}(e_{x2})} \Big\{ b \in \bigcup_{m\in E_c(n)} A_d(m,V_y) : \\ &\left(\neg\exists p \in \bigcup_{m\in E_c(n)} A_d(m,V_y) : p \in E_p(b)\right) \wedge C \Big\} \Bigg) \\ &\bigcup \bigcup_{n\in E_{dc}(e_{x2})} A_d(n,V_y) : m \in E_p(a) \Bigg) \wedge C \Bigg\} \end{aligned}$$

$$B_i(e_{x2},V_y)=\Big\{a\in\Big\{b\in\bigcup_{m\in E_{oc}(e_{x2})}A_d(m,V_y):$$
$$\Big(\neg\exists p\in\bigcup_{q\in E_c(x)}A_d(q,V_y):p\in E_p(b)\Big)\wedge C\Big\}\cup\bigcup_{n\in E_{dc}(e_{x2})}A_d(n,V_y):$$
$$\Big(\neg\exists m\in\Big\{b\in\bigcup_{m\in E_{oc}(e_{x2})}A_d(m,V_y):$$
$$\Big(\neg\exists p\in\bigcup_{q\in E_c(x)}A_d(q,V_y):p\in E_p(b)\Big)\wedge C\Big\}$$
$$\cup\bigcup_{n\in E_{dc}(e_{x2})}A_d(n,V_y):m\in E_p(a)\Big)\wedge C\Big\}$$

Where x is the parent of m that is also the direct child of e_{x2};

and $E_{oc}(e_{x2})=E_c(e_{x2})-E_{dc}(e_{x2})$

Now, let

$$Y(e_{x2},V_y)=\Big\{b\in\bigcup_{m\in E_{oc}(e_{x2})}A_d(m,V_y):\Big(\exists p\in\bigcup_{q\in E_c(x)}A_d(q,V_y):p\in E_p(b)\Big)\wedge C\Big\} \quad [9]$$

We have

$$Y(e_{x2},V_y)'=\bigcup_{n\in E_{oc}(e_{z2})}A_d(n,V_y)-Y(e_{x2},V_y), \quad [10]$$

since $\bigcup_{n\in E_{oc}(e_{x2})}A_d(n,V_y)\supset Y(e_{x2},V_y)$

$B_i(e_{x2},V_y)$ can be rewritten as:

$$B_i(e_{x2},V_y)=\Big\{a\in\Big(Y(e_{x2},V_y)'\cup\bigcup_{n\in E_{dc}(e_{x2})}A_d(n,V_y)\Big):$$
$$\Big(\neg\exists m\in\Big(Y(e_{x2},V_y)'\cup\bigcup_{n\in E_{dc}(e_{x2})}A_d(n,V_y)\Big):m\in E_p(a)\Big)\wedge C\Big\} \quad [11]$$

We first prove:

$$\forall a\in Y(e_{x2},V_y):\Big(\exists m\in Y(e_{x2},V_y)':m\in E_p(a)\Big) \quad [12]$$

Consider such an $a\in Y(e_{x2},V_y)$:

$$\begin{aligned}
& a \in Y(e_{x2}, V_y) \\
& \Rightarrow \exists p \in \bigcup_{q \in E_c(x)} A_d(q, V_y) : p \in E_p(a) \quad [\text{Definition of } Y(e_{x2}, V_y)] \\
& \Rightarrow \exists p \in \bigcup_{q \in E_{oc}(e_{x2})} A_d(q, V_y) : p \in E_p(a) \quad [E_c(x) \subset E_{oc}(e_{x2})]
\end{aligned} \qquad [13]$$

$p \in Y(e_{x2}, V_y)' \vee p \in Y(e_{x2}, V_y)$,

since $p \in \bigcup_{q \in E_{oc}(e_{x2})} A_d(q, V_y)$, and $\bigcup_{q \in E_{oc}(e_{x2})} A_d(q, V_y) \supset Y(e_{x2}, V_y)$.

If $p \in Y(e_{x2}, V_y)'$, then we found a $p \in Y(e_{x2}, V_y)'$, such that $p \in E_p(a)$, and hence proving expression [12].

If $p \in Y(e_{x2}, V_y)$, then

$$\begin{aligned}
& p \in Y(e_{x2}, V_y) \\
& \Rightarrow \exists v \in \bigcup_{q \in E_c(x)} A_d(q, V_y) : v \in E_p(p) \qquad [\text{Definition of } Y(e_{x2}, V_y)] \\
& \Rightarrow \exists v \in \bigcup_{q \in E_{oc}(e_{x2})} A_d(q, V_y) : v \in E_p(p) \qquad [E_c(x) \subset E_{oc}(e_{x2})]
\end{aligned}$$

This is similar to expression [13], where p replaces a, v replaces p, with $p \in E_p(a)$, and $v \in E_p(p)$.

So, by repeating the above argument, we can either deduce the following statements:

$$\exists v \in \bigcup_{q \in E_{oc}(e_{x2})} A_d(q, V_y) : v \in E_p(p), \exists u \in \bigcup_{q \in E_{oc}(e_{x2})} A_d(q, V_y) : u \in E_p(v), \quad \ldots$$

if $v \in Y(e_{x2}, V_y), u \in Y(e_{x2}, V_y)$, etc.

(Where $p \in E_p(a), v \in E_p(p),\ u \in E_p(v), \ldots$)

Or prove expression [12] if $v \in Y(e_{x2}, V_y)', u \in Y(e_{x2}, V_y)'$, since we would have found a $v/u \in Y(e_{x2}, V_y)'$, such that $v/u \in E_p(a)$.

(Note that $v \in E_p(a)$, since $v \in E_p(p) \in E_p(E_p(a)) \in E_p(a)$)

This sequence is repeated along the parents of a (p, v, u, s, …, r) until either expression [12] is satisfied at some point in the hierarchy, or the root of the tree, r, is reached. In the worst case where the sequence reaches the root r, we similarly get

$$\exists t \in \bigcup_{q \in E_{oc}(e_{x2})} A_d(q, V_y) : t \in E_p(r)$$

But, since no such t can exist since r is the root of the tree, we conclude that $r \notin Y(e_{x2}, V_y)$, and it must be the case that $r \in Y(e_{x2}, V_y)'$, also satisfying expression [12].

Therefore, in all cases, expression [12] is satisfied.

Now, reconsider the equation for $B_i(e_{x2}, V_y)$ in [11]:

$$B_i(e_{x2}, V_y) = \left\{ a \in \left(Y(e_{x2}, V_y)' \cup \bigcup_{n \in E_{dc}(e_{x2})} A_d(n, V_y) \right) : \right.$$
$$\left. \left(\neg \exists m \in \left(Y(e_{x2}, V_y)' \cup \bigcup_{n \in E_{dc}(e_{x2})} A_d(n, V_y) \right) : m \in E_p(a) \right) \wedge C \right\}$$

One can add the $Y(e_{x2}, V_y)$ set to the set of elements to choose from in the expression for $B_i(e_{x2}, V_y)$, since these added elements will not satisfy the condition of the $B_i(e_{x2}, V_y)$ set: $\left(\neg \exists m \in \left(Y(e_{x2}, V_y)' \cup \bigcup_{n \in E_{dc}(e_{x2})} A_d(n, V_y) \right) : m \in E_p(a) \right) \wedge C$, since from [12], we know that for $\forall a \in Y(e_{x2}, V_y)$, the expression $\left(\exists m \in Y(e_{x2}, V_y)' : m \in E_p(a) \right)$ is true.

Therefore, [11] can be rewritten as:

$$\begin{aligned}B_i(e_{x2},V_y) &= \Bigg\{a \in \Bigg(Y(e_{x2},V_y) \cup Y(e_{x2},V_y)' \cup \bigcup_{n\in E_{dc}(e_{x2})} A_d(n,V_y)\Bigg): \\ &\qquad \Bigg(\neg\exists m \in \Bigg(Y(e_{x2},V_y)' \cup \bigcup_{n\in E_{dc}(e_{x2})} A_d(n,V_y)\Bigg): m \in E_p(a)\Bigg) \wedge C\Bigg\} \\ &= \Bigg\{a \in \Bigg(\bigcup_{n\in E_{oc}(e_{x2})} A_d(n,V_y) \cup \bigcup_{n\in E_{dc}(e_{x2})} A_d(n,V_y)\Bigg): \\ &\qquad \Bigg(\neg\exists m \in \Bigg(Y(e_{x2},V_y)' \cup \bigcup_{n\in E_{dc}(e_{x2})} A_d(n,V_y)\Bigg): m \in E_p(a)\Bigg) \wedge C\Bigg\} \\ &= \Bigg\{a \in \bigcup_{n\in E_c(e_{x2})} A_d(n,V_y): \\ &\qquad \Bigg(\neg\exists m \in \Bigg(Y(e_{x2},V_y)' \cup \bigcup_{n\in E_{dc}(e_{x2})} A_d(n,V_y)\Bigg): m \in E_p(a)\Bigg) \wedge C\Bigg\}\end{aligned} \quad [14]$$

We now prove:

$$\forall a \in B_i(e_{x2},V_y): (\neg\exists m \in Y(e_{x2},V_y): m \in E_p(a)) \quad [15]$$

Assume the inverse of [15]. That is:

$$\exists a \in B_i(e_{x2},V_y): (\exists m \in Y(e_{x2},V_y): m \in E_p(a)) \quad [16]$$

For this *a*, we know that $(\exists m \in Y(e_{x2},V_y): m \in E_p(a))$

$$\begin{aligned}& m \in Y(e_{x2},V_y) \\ \Rightarrow\ & \exists p \in \bigcup_{q\in E_c(x)} A_d(q,V_y): p \in E_p(m) \quad [\text{Definition of } Y(e_{x2},V_y)] \\ \Rightarrow\ & \exists p \in \bigcup_{q\in E_{oc}(e_{x2})} A_d(q,V_y): p \in E_p(m) \quad [E_c(x) \subset E_{oc}(e_{x2})]\end{aligned} \quad [17]$$

$p \in Y(e_{x2},V_y)' \vee p \in Y(e_{x2},V_y)$,

since $p \in \bigcup_{q\in E_{oc}(e_{x2})} A_d(q,V_y)$, and $\bigcup_{q\in E_{oc}(e_{x2})} A_d(q,V_y) \supset Y(e_{x2},V_y)$.

But, $p \notin Y(e_{x2},V_y)'$, Since

$$\begin{aligned}& p \in E_p(m) && [\text{from } [17]] \\ \Rightarrow\ & p \in E_p(E_p(a)) && [\mathrm{m} \in \mathrm{E_p}(a), \text{from } [16]] \\ \Rightarrow\ & p \in E_p(a)\end{aligned}$$

and

$$\begin{aligned}
& a \in B_i(e_{x2}, V_y) \\
& \Rightarrow \left(\neg \exists m \in \left(Y(e_{x2}, V_y)' \cup \bigcup_{n \in E_{dc}(e_{x2})} A_d(n, V_y) \right) : m \in E_p(a) \right) \wedge C \quad [\text{From } [14]] \\
& \Rightarrow \left(\neg \exists m \in Y(e_{x2}, V_y)' : m \in E_p(a) \right)
\end{aligned}$$

That is, if $\left(\neg \exists m \in Y(e_{x2}, V_y)' : m \in E_p(a) \right)$ and $p \in E_p(a)$, then $p \notin Y(e_{x2}, V_y)'$.

Therefore,

$$p \in Y(e_{x2}, V_y)$$

Now,

$$\begin{aligned}
& p \in Y(e_{x2}, V_y) \\
& \Rightarrow \exists v \in \bigcup_{q \in E_c(x)} A_d(q, V_y) : v \in E_p(p) \quad [\text{Definition of } Y(e_{x2}, V_y)] \\
& \Rightarrow \exists v \in \bigcup_{q \in E_{oc}(e_{x2})} A_d(q, V_y) : v \in E_p(p) \quad [E_c(x) \subset E_{oc}(e_{x2})]
\end{aligned}$$

This is similar to expression [17], where, where *p* replaces *m*, *v* replaces *p* with, $p \in E_p(m)$ and $v \in E_p(p)$.

So, by repeating the above argument, the following statements can be deduced:

$$\exists v \in \bigcup_{q \in E_{oc}(e_{x2})} A_d(q, V_y) : v \in E_p(p),\ \exists u \in \bigcup_{q \in E_{oc}(e_{x2})} A_d(q, V_y) : u \in E_p(v),\ \ldots$$

where $v \in E_p(p),\ u \in E_p(v), \ldots$

This sequence is repeated along the parents of *a* (*m*, *p*, *v*, *u*, …, *r*) until the root of the tree, *r*, is reached, and concluding that

$$\exists t \in \bigcup_{q \in E_{oc}(e_{x2})} A_d(q, V_y) : t \in E_p(r)$$

But, since no such *t* can exist since *r* is the root of the tree, we conclude that assumption [16] is false.

Hence [16]'s inverse, [15] is true.

Now, reconsider the equation for $B_i(e_{x2}, V_y)$ in [14]:

$$B_i(e_{x2},V_y)=\Big\{a\in\bigcup_{n\in E_c(e_{x2})}A_d(n,V_y):$$
$$\Big(\neg\exists m\in\Big(Y(e_{x2},V_y)'\cup\bigcup_{n\in E_{dc}(e_{x2})}A_d(n,V_y)\Big):m\in E_p(a)\Big)\wedge C\Big\}$$

We know from [15] that for $\forall a\in B_i(e_{x2},V_y)$, $(\neg\exists m\in Y(e_{x2},V_y):m\in E_p(a))$ is true.

Therefore, [14] can be rewritten:

$$B_i(e_{x2},V_y)$$
$$=\Big\{a\in\bigcup_{n\in E_c(e_{x2})}A_d(n,V_y):$$
$$\Big(\neg\exists m\in\Big(Y(e_{x2},V_y)'\cup\bigcup_{n\in E_{dc}(e_{x2})}A_d(n,V_y)\Big):m\in E_p(a)\Big)\wedge C$$
$$\wedge(\neg\exists m\in Y(e_{x2},V_y):m\in E_p(a))\Big\}$$
$$=\Big\{a\in\bigcup_{n\in E_c(e_{x2})}A_d(n,V_y):$$
$$\Big(\neg\exists m\in\Big(Y(e_{x2},V_y)\cup Y(e_{x2},V_y)'\cup\bigcup_{n\in E_{dc}(e_{x2})}A_d(n,V_y)\Big):m\in E_p(a)\Big)\wedge C\Big\}$$
$$=\Big\{a\in\bigcup_{n\in E_c(e_{x2})}A_d(n,V_y):$$
$$\Big(\neg\exists m\in\Big(\bigcup_{n\in E_{oc}(e_{x2})}A_d(n,V_y)\cup\bigcup_{n\in E_{dc}(e_{x2})}A_d(n,V_y)\Big):m\in E_p(a)\Big)\wedge C\Big\}$$
$$=\Big\{a\in\bigcup_{n\in E_c(e_{x2})}A_d(n,V_y):$$
$$\Big(\neg\exists m\in\bigcup_{n\in E_c(e_{x2})}A_d(n,V_y):m\in E_p(a)\Big)\wedge C\Big\}$$
$$=A_i(e_{x2},V_y) \qquad [18]$$

Now,

$$\begin{aligned} B_a(e_{x2},V_y) &= B_i(e_{x2},V_y) \cup A_d(e_{x2},V_y) && [\text{from } [4]] \\ &= A_i(e_{x2},V_y) \cup A_d(e_{x2},V_y) && [\text{from } [18]] \\ &= A_a(e_{x2},V_y) \end{aligned}$$ [19]

Combining [18] and [19], we get

$$\left(A_i(e_{x2},V_y) = B_i(e_{x2},V_y)\right) \wedge \left(A_a(e_{x2},V_y) = B_a(e_{x2},V_y)\right)$$

We have now proved that [5] is true for *e*$_{x2}$, assuming [5] is true for $\forall e_{x1} \in E_c(e_{x2})$ ([8]).

And, given that [5] is true for the leafs of the model ([6] and [7]), then by induction, this proves [5] for $\forall e_x \in E_x$

C.2 Proof 2

Prove that

$$\begin{pmatrix} \left(E_p(e_y) \cap \bigcup_{n \in E_c(e_x)} A_d(n,V_y) = \varnothing\right) \\ \wedge \left(e_y \notin A_d(e_x,V_y)\right) \\ \wedge \left(E_p(e_y) \cap A_d(e_x,V_y) = \varnothing\right) \\ \wedge \left(E_c(e_y) \cap A_d(e_x,V_y) = \varnothing\right) \end{pmatrix}$$

$$\Leftrightarrow$$

$$\begin{pmatrix} \left(E_p(e_y) \cap A_a(e_x,V_y) = \varnothing\right) \\ \wedge \left(e_y \notin A_d(e_x,V_y)\right) \\ \wedge \left(E_c(e_y) \cap A_d(e_x,V_y) = \varnothing\right) \end{pmatrix}$$

We first prove that

$$\begin{pmatrix} \left(E_p(e_y) \cap \bigcup_{n \in E_c(e_x)} A_d(n, V_y) = \emptyset\right) \\ \wedge \left(e_y \notin A_d(e_x, V_y)\right) \\ \wedge \left(E_p(e_y) \cap A_d(e_x, V_y) = \emptyset\right) \\ \wedge \left(E_c(e_y) \cap A_d(e_x, V_y) = \emptyset\right) \end{pmatrix}$$

$$\Leftrightarrow$$

$$\begin{pmatrix} \left(E_p(e_y) \cap A_i(e_x, V_y) = \emptyset\right) \\ \wedge \left(e_y \notin A_d(e_x, V_y)\right) \\ \wedge \left(E_p(e_y) \cap A_d(e_x, V_y) = \emptyset\right) \\ \wedge \left(E_c(e_y) \cap A_d(e_x, V_y) = \emptyset\right) \end{pmatrix} \qquad [1]$$

Now,

$$\left(E_p(e_y) \cap \bigcup_{n \in E_c(e_x)} A_d(n, V_y) = \emptyset\right)$$

$$\Rightarrow \neg\exists x \in E_p(e_y) : x \in \bigcup_{n \in E_c(e_x)} A_d(n, V_y)$$

$$\Rightarrow \neg\exists x \in E_p(e_y) : x \in A_i(e_x, V_y) \qquad \left[\text{Since } A_i(e_x, V_y) \subset \bigcup_{n \in E_c(e_x)} A_d(n, V_y)\right]$$

$$\Rightarrow \left(E_p(e_y) \cap A_i(e_x, V_y) = \emptyset\right)$$

Hence,

$$\begin{pmatrix} \left(E_p(e_y) \cap \bigcup_{n \in E_c(e_x)} A_d(n, V_y) = \emptyset\right) \\ \wedge \left(e_y \notin A_d(e_x, V_y)\right) \\ \wedge \left(E_p(e_y) \cap A_d(e_x, V_y) = \emptyset\right) \\ \wedge \left(E_c(e_y) \cap A_d(e_x, V_y) = \emptyset\right) \end{pmatrix}$$

$$\Rightarrow$$

$$\begin{pmatrix} \left(E_p(e_y) \cap A_i(e_x, V_y) = \emptyset\right) \\ \wedge \left(e_y \notin A_d(e_x, V_y)\right) \\ \wedge \left(E_p(e_y) \cap A_d(e_x, V_y) = \emptyset\right) \\ \wedge \left(E_c(e_y) \cap A_d(e_x, V_y) = \emptyset\right) \end{pmatrix} \qquad [2]$$

Considering the RHS of (1),

$$\begin{pmatrix} \left(E_p(e_y) \cap A_i(e_x, V_y) = \varnothing\right) \\ \wedge \left(e_y \notin A_d(e_x, V_y)\right) \\ \wedge \left(E_p(e_y) \cap A_d(e_x, V_y) = \varnothing\right) \\ \wedge \left(E_c(e_y) \cap A_d(e_x, V_y) = \varnothing\right) \end{pmatrix}$$

$$\Rightarrow$$

$$\begin{pmatrix} \left(\neg\exists x \in A_i(e_x, V_y) : x \in E_p(e_y)\right) \\ \wedge \left(e_y \notin A_d(e_x, V_y)\right) \\ \wedge \left(E_p(e_y) \cap A_d(e_x, V_y) = \varnothing\right) \\ \wedge \left(E_c(e_y) \cap A_d(e_x, V_y) = \varnothing\right) \end{pmatrix} \qquad [3]$$

Now, assume that

$$\exists a \in \bigcup_{n \in E_c(e_x)} A_d(n, V_y) : a \in E_p(e_y) \qquad [4]$$

$a \in A_i(e_x, V_y) \vee a \in A_i(e_x, V_y)'$,since $a \in \bigcup_{n \in E_c(e_x)} A_d(n, V_y)$,
and $\bigcup_{n \in E_c(e_x)} A_d(n, V_y) \supset A_i(e_x, V_y)$.

But $a \notin A_i(e_x, V_y)$, since from [3], we have $\neg\exists x \in A_i(e_x, V_y) : x \in E_p(e_y)$, and from [4] we have $a \in E_p(e_y)$.

Therefore,

$$a \in A_i(e_x, V_y)'$$

From the definition of $A_i(e_x, V_y)$ (section B.5.1.1), we get that for $a \in A_i(e_x, V_y)'$

$$\neg\begin{pmatrix} \left(\neg\exists m \in \bigcup_{n \in E_c(e_x)} A_d(n, V_y) : m \in E_p(a)\right) \\ \wedge \left(\neg\exists m \in A_d(e_x, V_y) : m \in E_p(a) \vee m = a\right) \end{pmatrix}$$

That is,

$$\begin{gathered} \left(\exists m \in \bigcup_{n \in E_c(e_x)} A_d(n, V_y) : m \in E_p(a)\right) \\ \vee \left(\exists m \in A_d(e_x, V_y) : m \in E_p(a) \wedge m \neq a\right) \end{gathered} \qquad [5]$$

Considering the second predicate of [5]:

$$\exists m \in A_d(e_x, V_y) : m \in E_p(a) \wedge m \neq a$$
$$\Rightarrow \exists m \in A_d(e_x, V_y) : m \in E_p(E_p(e_y)) \wedge m \neq a \qquad [\text{from } [4], \text{ we have } a \in E_p(e_y)]$$
$$\Rightarrow \exists m \in A_d(e_x, V_y) : m \in E_p(e_y) \wedge m \neq a$$

But, this is false since it is given in [3] that $E_p(e_y) \cap A_d(e_x, V_y) = \varnothing$

Hence [5] becomes:

$$\left(\exists m \in \bigcup_{n \in E_c(e_x)} A_d(n, V_y) : m \in E_p(a)\right)$$

This is similar to assumption [4], where *m* replaces *a* with, $a \in E_p(e_y)$ and $m \in E_p(a)$.

So, by repeating the argument above, the following statements can be deduced:

$$\left(\exists p \in \bigcup_{n \in E_c(e_x)} A_d(n, V_y) : p \in E_p(m)\right), \left(\exists q \in \bigcup_{n \in E_c(e_x)} A_d(n, V_y) : q \in E_p(p)\right), \ldots,$$

where $p \in E_p(m), q \in E_p(p), \ldots$

This sequence is repeated along the parents of *e* (a, *m*, *p*, *q*, …, *r*) until the root of the tree, *r*, is reached, and concluding that

$$\left(\exists v \in \bigcup_{n \in E_c(e_x)} A_d(n, V_y) : v \in E_p(r)\right)$$

But, since no such *v* can exist since *r* is the root of the tree, we conclude that assumption [4] is false.

That is

$$\neg \exists a \in \bigcup_{n \in E_c(e_x)} A_d(n, V_y) : a \in E_p(e_y)$$

or

$$E_p(e_y) \cap \bigcup_{n \in E_c(e_x)} A_d(n, V_y) = \varnothing \qquad [6]$$

Now, [6] is proven true based on [3], and we hence can write:

$$\begin{pmatrix} \left(\neg\exists x \in A_i\left(e_x,V_y\right): x \in E_p\left(e_y\right)\right) \\ \wedge \left(e_y \notin A_d\left(e_x,V_y\right)\right) \\ \wedge \left(E_p\left(e_y\right) \cap A_d\left(e_x,V_y\right) = \varnothing\right) \\ \wedge \left(E_c\left(e_y\right) \cap A_d\left(e_x,V_y\right) = \varnothing\right) \end{pmatrix}$$

$$\Rightarrow$$

$$\left(E_p\left(e_y\right) \cap \bigcup_{n \in E_c\left(e_x\right)} A_d\left(n,V_y\right) = \varnothing\right)$$

$\therefore$

$$\begin{pmatrix} \left(E_p\left(e_y\right) \cap A_i\left(e_x,V_y\right) = \varnothing\right) \\ \wedge \left(e_y \notin A_d\left(e_x,V_y\right)\right) \\ \wedge \left(E_p\left(e_y\right) \cap A_d\left(e_x,V_y\right) = \varnothing\right) \\ \wedge \left(E_c\left(e_y\right) \cap A_d\left(e_x,V_y\right) = \varnothing\right) \end{pmatrix}$$

$$\Rightarrow$$

$$\left(E_p\left(e_y\right) \cap \bigcup_{n \in E_c\left(e_x\right)} A_d\left(n,V_y\right) = \varnothing\right)$$

$\therefore$

$$\begin{pmatrix} \left(E_p\left(e_y\right) \cap A_i\left(e_x,V_y\right) = \varnothing\right) \\ \wedge \left(e_y \notin A_d\left(e_x,V_y\right)\right) \\ \wedge \left(E_p\left(e_y\right) \cap A_d\left(e_x,V_y\right) = \varnothing\right) \\ \wedge \left(E_c\left(e_y\right) \cap A_d\left(e_x,V_y\right) = \varnothing\right) \end{pmatrix}$$

$$\Rightarrow$$

$$\begin{pmatrix} \left(E_p\left(e_y\right) \cap \bigcup_{n \in E_c\left(e_x\right)} A_d\left(n,V_y\right) = \varnothing\right) \\ \wedge \left(e_y \notin A_d\left(e_x,V_y\right)\right) \\ \wedge \left(E_p\left(e_y\right) \cap A_d\left(e_x,V_y\right) = \varnothing\right) \\ \wedge \left(E_c\left(e_y\right) \cap A_d\left(e_x,V_y\right) = \varnothing\right) \end{pmatrix} \qquad [7]$$

Combining [2] and [7], we get:

$$\begin{pmatrix} \left(E_p(e_y) \cap A_i(e_x, V_y) = \varnothing\right) \\ \wedge \left(e_y \notin A_d(e_x, V_y)\right) \\ \wedge \left(E_p(e_y) \cap A_d(e_x, V_y) = \varnothing\right) \\ \wedge \left(E_c(e_y) \cap A_d(e_x, V_y) = \varnothing\right) \end{pmatrix}$$

$$\Leftrightarrow$$

$$\begin{pmatrix} \left(E_p(e_y) \cap \bigcup_{n \in E_c(e_x)} A_d(n, V_y) = \varnothing\right) \\ \wedge \left(e_y \notin A_d(e_x, V_y)\right) \\ \wedge \left(E_p(e_y) \cap A_d(e_x, V_y) = \varnothing\right) \\ \wedge \left(E_c(e_y) \cap A_d(e_x, V_y) = \varnothing\right) \end{pmatrix}$$

Hence, we prove [1].

Now,

$$\left(\left(E_p(e_y) \cap A_i(e_x, V_y) = \varnothing\right) \wedge \left(E_p(e_y) \cap A_d(e_x, V_y) = \varnothing\right)\right)$$

$$\Leftrightarrow$$

$$\left(E_p(e_y) \cap A_a(e_x, V_y) = \varnothing\right)$$

Since $A_a(e_x, V_y) = A_i(e_x, V_y) \cup A_d(e_x, V_y)$

Hence,

$$\begin{pmatrix} \left(E_p(e_y) \cap \bigcup_{n \in E_c(e_x)} A_d(n, V_y) = \varnothing\right) \\ \wedge \left(e_y \notin A_d(e_x, V_y)\right) \\ \wedge \left(E_p(e_y) \cap A_d(e_x, V_y) = \varnothing\right) \\ \wedge \left(E_c(e_y) \cap A_d(e_x, V_y) = \varnothing\right) \end{pmatrix}$$

$$\Leftrightarrow$$

$$\begin{pmatrix} \left(E_p(e_y) \cap A_a(e_x, V_y) = \varnothing\right) \\ \wedge \left(e_y \notin A_d(e_x, V_y)\right) \\ \wedge \left(E_c(e_y) \cap A_d(e_x, V_y) = \varnothing\right) \end{pmatrix}$$

C.3 Proof 3

Prove that

$$\left(A_d\left(e_x,V_y\right)\neq\varnothing\right)\vee\left(\exists n\in E_c\left(e_x\right):A_d\left(n,V_y\right)\neq\varnothing\right)\equiv\left(A_a\left(e_x,V_y\right)\neq\varnothing\right)$$

First,

$$\begin{aligned}&\left(A_d\left(e_x,V_y\right)\neq\varnothing\right)\vee\left(\exists n\in E_c\left(e_x\right):A_d\left(n,V_y\right)\neq\varnothing\right)\\ &\equiv\neg\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\neg\left(\exists n\in E_c\left(e_x\right):A_d\left(n,V_y\right)\neq\varnothing\right)\right)\\ &\equiv\neg\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\left(\forall n\in E_c\left(e_x\right):A_d\left(n,V_y\right)=\varnothing\right)\right)\\ &\equiv\neg\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\left(\bigcup_{n\in E_c\left(e_x\right)}A_d\left(n,V_y\right)=\varnothing\right)\right)\end{aligned}$$ [1]

We now prove that

$$\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\left(\bigcup_{n\in E_c\left(e_x\right)}A_d\left(n,V_y\right)=\varnothing\right)\right)$$
$$\Leftrightarrow$$
$$\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\left(A_i\left(e_x,V_y\right)=\varnothing\right)\right)$$ [2]

First, given the definition of A_i in section B.5.1.1:

$$\begin{aligned}&\bigcup_{n\in E_c\left(e_x\right)}A_d\left(n,V_y\right)=\varnothing\\ &\Rightarrow A_i(e_x,V_y)=\left\{a\in\varnothing:\left(\neg\exists m\in\bigcup_{n\in E_c\left(e_x\right)}A_d\left(n,V_y\right):m\in E_p\left(a\right)\right)\right.\\ &\qquad\left.\wedge\left(\neg\exists m\in A_d\left(e_x,V_y\right):m\in E_p\left(a\right)\vee m=a\right)\right\}\\ &\Rightarrow A_i(e_x,V_y)=\varnothing\end{aligned}$$

Hence,

$$\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\left(\bigcup_{n\in E_c\left(e_x\right)}A_d\left(n,V_y\right)=\varnothing\right)\right)\Rightarrow\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\left(A_i\left(e_x,V_y\right)=\varnothing\right)\right)$$ [3]

Second,

$$\begin{aligned}
&\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\left(A_i\left(e_x,V_y\right)=\varnothing\right)\right)\\
&\Rightarrow\left\{a\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):\left(\neg\exists m\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):m\in E_p(a)\right)\right.\\
&\qquad\qquad\left.\wedge\left(\neg\exists m\in\varnothing:m\in E_p(a)\vee m=a\right)\right\}\\
&\quad=\varnothing\\
&\Rightarrow\left\{a\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):\left(\neg\exists m\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):m\in E_p(a)\right)\right\}=\varnothing\\
&\Rightarrow\left(\forall a\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):\left(\exists m\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):m\in E_p(a)\right)\right)\\
&\quad\vee\left(\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right)=\varnothing\right)
\end{aligned}\qquad[4]$$

Now, assume that

$$\forall a\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):\left(\exists m\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):m\in E_p(a)\right)\qquad[5]$$

And consider an *a* such that $a\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right)$.

$$\begin{aligned}
&a\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right)\\
&\Rightarrow\exists m\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):m\in E_p(a) &&[\text{from }[5]]\\
&\Rightarrow\exists p\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):p\in E_p(m) &&\left[\text{from }[5],\text{ since } m\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right)\right]\\
&\Rightarrow\exists q\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):q\in E_p(p) &&\left[\text{from }[5],\text{ since } p\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right)\right]\\
&\quad\ldots
\end{aligned}$$

Note that $m\in E_p(a)$, $p\in E_p(m)$, $q\in E_p(p)$, etc.

This sequence is repeated along the parents of *a* (*m*, *p*, *q*, …, *r*) until the root of the tree, *r*, is reached, concluding that

$$r\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right)$$

and

$$\exists v\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):v\in E_p(r)$$

But since no such *v* can exist, we can conclude that [5] is not valid.

Therefore, [4] becomes

$$\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\left(A_i\left(e_x,V_y\right)=\varnothing\right)\right)$$
$$\Rightarrow\left(\bigcup_{n\in E_c\left(e_x\right)}A_d\left(n,V_y\right)=\varnothing\right)$$

Hence,

$$\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\left(A_i\left(e_x,V_y\right)=\varnothing\right)\right)\Rightarrow\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\left(\bigcup_{n\in E_c\left(e_x\right)}A_d\left(n,V_y\right)=\varnothing\right)\right) \qquad [6]$$

Combining [3] and [6], we get

$$\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\left(\bigcup_{n\in E_c\left(e_x\right)}A_d\left(n,V_y\right)=\varnothing\right)\right)$$
$$\Leftrightarrow$$
$$\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\left(A_i\left(e_x,V_y\right)=\varnothing\right)\right)$$

and thus proving [2].

Combining [1] and [2], we get:

$$\begin{aligned}
&\left(A_d\left(e_x,V_y\right)\neq\varnothing\right)\vee\left(\exists n\in E_c\left(e_x\right):A_d\left(n,V_y\right)\neq\varnothing\right)\\
&\equiv\neg\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\left(\bigcup_{n\in E_c\left(e_x\right)}A_d\left(n,V_y\right)=\varnothing\right)\right) \qquad [\text{from}\,[1]]\\
&\equiv\neg\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\left(A_i\left(e_x,V_y\right)=\varnothing\right)\right) \qquad [\text{from}\,[2]]\\
&\equiv\left(A_d\left(e_x,V_y\right)\cup A_i\left(e_x,V_y\right)\right)\neq\varnothing\\
&\equiv A_a\left(e_x,V_y\right)\neq\varnothing
\end{aligned}$$

C.4 Proof 4

Prove that

$$\left(\forall n\in E_c\left(e_x\right):\neg a_{xv}\left(n,V_y\right)\right)\equiv\left(\forall n\in E_{dc}\left(e_x\right):\neg a_{xv}\left(n,V_y\right)\right)$$

First,

$$\left(\forall n\in E_c\left(e_x\right):\neg a_{xv}\left(n,V_y\right)\right)\Rightarrow\left(\forall n\in E_{dc}\left(e_x\right):\neg a_{xv}\left(n,V_y\right)\right)$$
$$\left[\text{Since}\,E_{dc}\left(e_x\right)\subset E_c\left(e_x\right)\right] \qquad [1]$$

Second,

$$\begin{aligned}
&\left(\forall n \in E_{dc}(e_x) : \neg a_{xv}(n, V_y)\right) \\
&\Rightarrow \left(\forall n \in E_{dc}(e_x) : \left(\forall m \in E_c(n) : \neg a_{xv}(m, V_y)\right)\right) \\
&\qquad\qquad \left[\text{From section1.6}, \neg a_{xv}(e_{x1}) \Rightarrow \forall n \in E_c(e_{x1}) : \neg a_{xv}(n, V_y)\right] \\
&\Rightarrow \left(\forall n \in E_{oc}(e_x) : \neg a_{xv}(n, V_y)\right) \\
&\Rightarrow \left(\forall n \in E_{oc}(e_x) : \neg a_{xv}(n, V_y)\right) \wedge \left(\forall n \in E_{dc}(e_x) : \neg a_{xv}(n, V_y)\right) \\
&\qquad\qquad \begin{bmatrix} (A \Rightarrow B) \equiv (A \Rightarrow (B \wedge A)) \\ \text{and} \quad E_{oc}(e_x) = E_c(e_x) - E_{dc}(e_x) \end{bmatrix} \\
&\Rightarrow \left(\forall n \in E_c(e_x) : \neg a_{xv}(n, V_y)\right)
\end{aligned}$$

Hence,

$$\left(\forall n \in E_{dc}(e_x) : \neg a_{xv}(n, V_y)\right) \Rightarrow \left(\forall n \in E_c(e_x) : \neg a_{xv}(n, V_y)\right) \qquad [2]$$

Combining [1] and [2], we get

$$\left(\forall n \in E_c(e_x) : \neg a_{xv}(n, V_y)\right) \Leftrightarrow \left(\forall n \in E_{dc}(e_x) : \neg a_{xv}(n, V_y)\right)$$

Paper-C

Model Data Management – Towards a common solution for PDM/SCM systems

El-khoury Jad

Published as
Twelfth International Software Configuration Management Workshop (SCM-12), 2005.

Abstract

Software Configuration Management and Product Data Management systems have been developed independently, but recently the need to integrate them to support multidisciplinary development environments has been recognised. Due to the difference in maturity levels of these disciplines, integration efforts have had limited success in the past. This paper examines how the move towards model-based development in software engineering is bringing the discipline closer to hardware development, permitting a tighter integration of their data management systems. An architecture for a Model Data Management system that supports model-based development is presented. The system aims to generically handle the models produced by the different tools during the development of software-intensive, yet multidisciplinary, products. The proposed architecture builds on existing technologies from the mature discipline of mechanical engineering, while borrowing new ideas from the software domain.

C.1. Introduction

Organisations involved in the development of large and complex products need to deal with a large amount of information, created and modified during the development and product life cycle. To support this need, an organisation normally adopts some kind of product management environment. Many such management solutions are currently available, and it is generally the case that each tends to focus on a specific class of products, determined by the major engineering domain involved in the product development. The development of software-intensive products relies on Software Configuration Management (SCM) systems, while mechanical system development uses Product Data Management (PDM) systems.

In the development of products that involve the collaboration of various engineering disciplines, a number of these management environments come into simultaneous use. This is necessary since developers from each discipline require the specific support provided by its corresponding system. An automotive system is a typical such product, where traditional engineering disciplines such as control, software, mechanical and electrical engineering, need to interact to meet the demands for dependable and cost-efficient integrated systems.

Considering the central role these environments take in controlling the development process as well as facilitating the communication between developers, integrating them becomes essential for the successful integration of the efforts of all disciplines involved. In multidisciplinary development, allowing the environments to run unsynchronised creates a source of inconsistencies and conflicts between the disciplines. In other words, it is equally important to provide (where possible) a common set of support mechanisms and principles within, as well as between, the disciplines.

While most of the general facilities provided by these solutions overlap, variations in the details exist due to the differing needs of the domains. This leads to complications and difficulties when attempting to integrate them [1]. In this paper, we discuss how the move towards a model-based development approach in software engineering is bringing it closer to the hardware engineering discipline, allowing for a tighter integration of their management systems. We advocate a common model-based management system that borrows from the technologies of each of these disciplines. In the next section, we discuss the differences between conventional SCM and PDM tools and investigate the effect of adopting model-based development in software engineering in bringing these solutions closer. Section C.3 presents a management system architecture that takes advantage of this change. This is followed by a discussion of related work in the area of PDM/SCM integration, before concluding the paper in section C.5.

C.2. Model-based Development – Bringing Software Development towards Hardware Development

Model-based development (MBD) refers to a development approach whose activities emphasise the use of models, tools and analysis techniques for the documentation, communication and analysis of decisions taken at each stage of the development lifecycle. Models can take many forms such as, (but not limited to,) graphical, textual and prototype models. It is essential however that the models contain sufficient and consistent information about the system, allowing reproducible and reliable analysis of specific properties to be performed.

With the maturity of the software discipline, the need to move towards a more model-based development approach is being recognized. This need is exemplified in (but certainly not limited to) the OMG efforts [2][3], and the wide range of tools supporting them.

In this section, we will investigate how the adoption of model-based development in software engineering can help bridge a gap between software and hardware development, leading towards a common solution for the data management of multidisciplinary products. In [1], three crucial factors for a successful integration of PDM and SCM are presented: processes, tools and technologies and people. We follow this categorisation in this investigation. New challenges facing such a common solution are also discussed.

C.2.1. Processes

The difference in the development process of software and hardware products has been most influential in the divergence between their management tools. The more mature hardware development expects support during the complete product life cycle from the early concept design phases down to manufacturing and post-production phases [4]. All product data from all these phases is expected to be handled and related through the PDM system. In comparison, as with any new discipline, early software development occurred in a relatively more ad-hoc manner with no, or little, early design and analysis phases. Consequently, these early phases were beyond the scope of SCM tools [4][5], and SCM was only expected to manage the large amount of source files produced during the implementation phase of software development.

In software engineering, the application of the model-based approach throughout the complete development process implies the need to handle different kinds of documentation from the early design and analysis stages, as well as implementation. Conventional SCM tools have so far incorporated these additional documents by simply treating them as files, without differentiating them from source code files. However, one cannot claim that SCM handles the development process appropriately, since no distinction is made between the types of documents produced during the different development phases. For this to be possible, we argue that the development process itself needs to be reflected in the product information model.

In [1], it is argued that the life cycle processes of the software and hardware development should be integrated for the successful integration of PDM/SCM. The challenges for such integration and a simple solution are then suggested. What seems to be missing in the discussion is how process integration would be beneficial for the integration of PDM and SCM systems. Studying the functionalities of PDM/SCM, one can see that such systems simply provide the infrastructure to enforce a given process (see section C.2.2) and play no direct role in integrating the development processes. Instead, PDM/SCM functionalities focus on the product data produced. For this reason, while process integration may be desired within an organisation, for the purpose of integrating PDM/SCM systems, it is even more important to focus on the integration of the outcomes/artefacts produced at each phase of the product lifecycle. The ultimate goal is the tight integration of the hardware and software components of the final product, and not the process of getting there.

C.2.2. Tools and Technologies

This category is further divided into six basic functionalities expected of PDM/SCM systems: data representation, version management, management of distributed data, product structure management, process support and document management.

C.2.2.1. Data Representation

A major difference between PDM and SCM lies in the kind of data that the support tools are expected to handle [6]. In hardware development, the need to provide a seamless workflow from design to manufacturing phases has forced PDM systems to not only handle the documents produced, but much of their internal contents (metadata) as well. A detailed information model of the product data is an integral part of a PDM system [7]. Software development, on the other hand, has so far adopted a file-based approach, only managing the files produced during development, and where the only relations handled between the files is that of the file system itself (a small amount of meta-data is also handled such as file author and modification date). The internal structure of these files and the semantical relationships between them has so far been outside the scope of SCM tools. PDM can be interpreted as managing product representations, while SCM manages the final product itself [8].

With the maturity of the software discipline, and its move towards a more model-based development approach, many documents (analysis models, uses cases, etc.) will be produced during development. These documents act as models representing certain aspects of the product and will not necessarily end up in the final product. Nevertheless, the different types of documents need to be identified in the management system and related to specific development stages.

The information stored in the documents is interrelated. For this reason, SCM systems supporting model-based development would need to, not only manage the files storing the models, but also the internal content of these models, allowing fine-grained relationships between the document contents to be setup. An information model of the complete information space contained in the models need to be an integral part of a SCM system.

In a model-based development approach, developers need to be shielded from the file structures used to store the models built, allowing them to focus on the models and their structures. This strategy is adopted by many modern modelling tools that may use database systems to store and hide models, and a modern management system should follow in this track.

C.2.2.2. Version Management

In PDM systems, revisions of an object are manually managed by the user and form a sequential series, with no possibility of performing parallel changes. In contrast, versions in SCM systems form a graph structure, with the possibility to perform branching in the development, followed by merging of the branched tracks. Due to these differences, the later approach facilitates concurrent engineering, which is limited in the former.

Accepting that SCM systems need to focus on modelling items, and not only the files storing these items, it becomes essential for version management functionality in SCM systems to similarly focus on the contents provided in these files. Instead of differentiating between the lines of text in different versions of a file, it is differences between the modelling items in different versions of a model that need to be identified and managed.

Since conventional SCM systems do not handle the internal semantics of files, it has also been out of its scope to ensure that parallel changes to the same item (file) are consistent upon a merge. SCM simply provides the mechanism to branch and merge changes made to unrelated lines of text. The burden is placed on the user to ensure that merged changes from different development tracks are consistent semantically. It was hence relatively easy to provide such semantic-free functionality.

Model-based version management becomes a challenge for SCM systems. Complexity arises due to the different kinds of modelling items that may exist in a model compared to the single type (lines of text) that are conventionally handled. It is no longer possible to provide the exact versioning functionalities for all kinds of documents in the system. In the best case, customisation of a generic mechanism will allow the reuse of much of this functionality.

An additional challenge is to ensure consistent parallel changes to the models stored in the files during version management. While lines of text in a file can be treated individually, modelling items in a model are generally tightly interrelated. Changes to one item may have implications on other items in the model. This implies that even though each individual set of changes in two parallel change tracks is semantically valid, merging these changes into a consistent set is not as simple as the union of the changes since the relations between the modelling items need to be taken into account. For example, in a class diagram, one track of changes may have deleted a certain class, while in another track a new association is created between that class and another. In merging these changes, it is first necessary to establish if the deleted class needs to be reintroduced before allowing the presence of the new association.

In dealing with this problem, an SCM system can adopt the approach of PDM of disallowing parallel changes and in this way preventing the problem from occurring in the first place. Another approach is to develop branch/merge mechanisms that work on model structures, maintaining support of concurrent development of models for software developers. A successful implementation of the latter approach can also be beneficial for hardware development, where the possibility to concurrently develop models becomes possible, leading the way for new development processes.

The need for concurrent changes to the same source code files partly originates from the less mature adhoc development of earlier software systems before software “engineering” became a discipline. It is argued that a structured model-based development approach would reduce the need for parallel access to the same product data and hence the former approach becomes more appropriate. In the case where concurrent changes remain a necessity, the latter approach needs to be supported.

Nevertheless, branch/merge mechanisms in SCM remain a necessity for the management of product variants. However, in model-based development, this implicit management of variants should be made more explicit, by representing variants in the product information model.

As discussed in section C.4, the model-based approach to versioning and branching/merging is gaining ground in the SCM community. In this paper, we advocate taking advantage of this new trend in the integration of PDM and SCM systems.

C.2.2.3. Management of Distributed Data

The need to manage geographically distributed data seems to be common for both disciplines, with the difference being in the technical solution provided by the management systems. PDM systems provided a more limiting functionality by not allowing concurrent access to distributed data. This difference is closely related to that discussed in the previous subsection, and synchronising the earlier difference will naturally lead to the synchronisation of this functionality. Technically, a common solution will choose either the currently adopted PDM or SCM solution based on whether concurrent access is desired or not.

In a model-based approach to distributed data management, the functionality would focus on the management of distributed fine-grained model data items and not the files storing these items.

C.2.2.4. Product Structure Management

In hardware systems, the physical structure of the final product is the single predominant structure. This structure is used throughout the development phases as a basis for the information model to which all other information is related. Conventional SCM systems do not explicitly support the structure of the product, focusing instead on the directory structure of the files it manages.

In a model-based approach to software development, an SCM system would need to focus on the internal structures of the models stored in the files instead. Unlike hardware products, when using models throughout the development phases, the software structure will vary widely, and hence the product structure management functionality of a model-based SCM needs to handle many different parallel structures. Relationships between these structures will also need to be taken into account.

Given the possibility to manage multiple structures, it becomes easier to also manage products resulting from the integrated effort of hardware and software development. Each discipline would be able to maintain its own structure. The possibility to set up relationships between the structures results in a tight integration of hardware and software components.

C.2.2.5. Process Support

As mentioned in section C.2.1, it is necessary to integrate the process support functionalities of PDM and SCM systems. Software and hardware development would need to follow different development processes, and this functionality should be able to support each of the chosen processes, yet based on common fundamental mechanisms: workflow management, user assignment, approach rule mechanisms, etc. As mentioned in [1], such functionality is already quite similar in PDM and SCM systems.

C.2.2.6. Document Management

Document management is an integral part of PDM systems, and such functionality is missing in conventional SCM systems. The need for document management by software developers is apparent, and hence a common efficient support ought to be technically feasible.

C.2.3. People and Cultural Behaviours

In [9], some of the differences in the terminologies used by software and hardware engineers are highlighted. These differences are attributed to the differences in the development phases generally focused on by these disciplines. For example, in software engineering, “design” is traditionally defined as building a model of the system up to the point at which coding begins. In hardware development, however, “design” would also include broader activities such as requirements and testing activities.

In adopting a model-based approach in both disciplines, and as a by-product of integrating the outcomes of each of the phases of the development processes as advocated earlier, it becomes necessary to integrate the meaning of some of the terminology used.

An important function of models is communication. While models are domain-specific and can only be understood in details by engineers of the specific disciplines, such models can be still used to communicate certain aspects of the design to other engineers, if presented at the right level of abstraction. If models from the various disciplines can be successfully interrelated to form a consistent whole view of the system through a common management system, such interrelations can also act as interaction points between the disciplines, reducing any risks of inconsistencies and conflicts.

C.2.4. Conclusion

The fundamental differences between SCM and PDM systems stem from the different needs of the disciplines they aim to support. As software development becomes increasingly model-based, and requires support throughout its development life cycle, its needs become closer to those of hardware development. In particular, the process management and information modelling functionalities expected of SCM systems come closer to those provided by PDM systems for hardware development.

This leads the way for an easier and more effective integrated management platform satisfying the needs of both disciplines using a common set of mechanisms. The management functionality ought to take advantage of the commonality between the disciplines – the use of models – in the development process by focusing on models and their internal content as central entities. This allows the same model-based functionalities to be used by both disciplines. We term such an approach as Model Data Management (MDM).

C.3. Model Data Management

In this section, we present an architecture for a Model Data Management (MDM) system that aims to generically support and control different kinds of models produced from a set of different tools and disciplines.

C.3.1. Tool Architecture

The envisaged architecture is shown in figure 33. The platform consists of two main parts: A set of tool-specific adaption layers and a data repository with mechanisms to handle this data. The data repository stores the data for each of the tools. To perform this role in a generic way, the data from the different tools is expected to be presented in a neutral form, and this functionality is provided by the adaption layer. Triggered either by a tool or the repository, the corresponding adaption layer permits the data flow between a tool and the repository, in a predefined format. The following subsections will further discuss these components.

Given its maturity, we aim to base the proposed MDM system on a configurable PDM system. The major advantage of using a PDM system is the possibility to define information models, with a high level query language to access and modify the model data in the repository. These facilities generally do not exist in conventional SCM systems. In addition, it is envisaged that the development of the remaining MDM functionalities is made easier given the already developed functionalities of PDM such as the support for distributed development, change management, workflow control, etc. The adoption of a PDM system is not indispensable and one can envisage building an independent MDM that supports both disciplines.

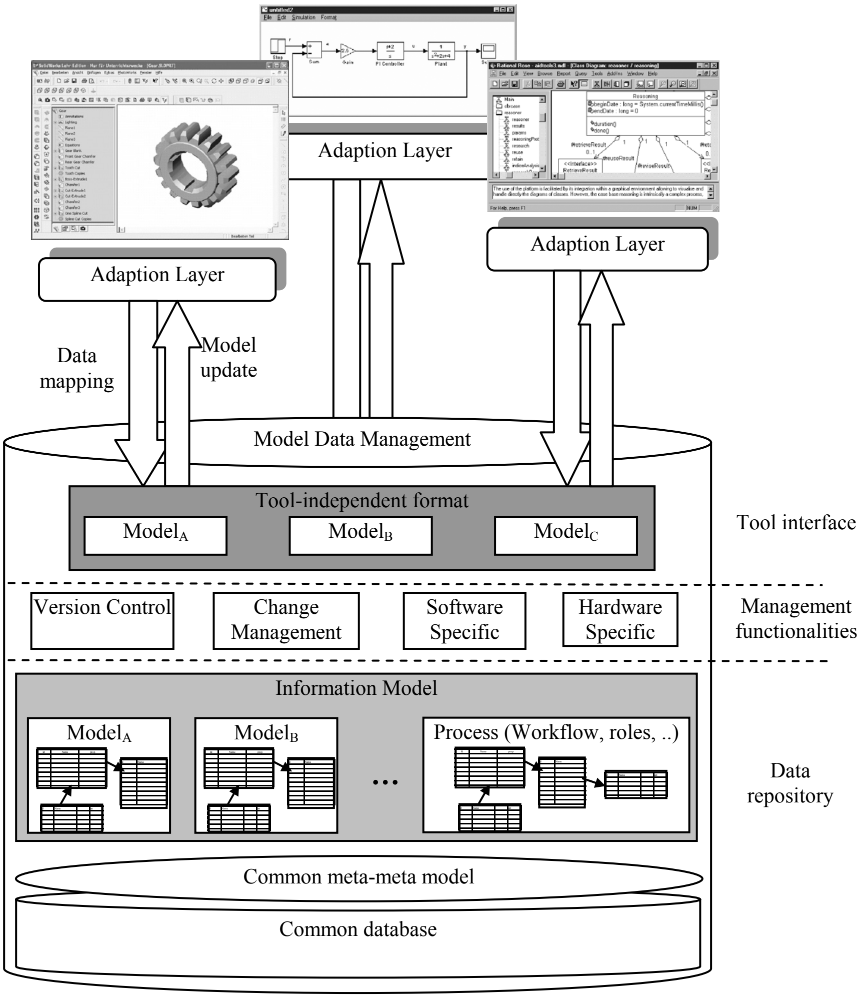


Figure 33. The major components of the MDM architecture. (Note that the graphical tools are mock-ups shown here for illustration purposes only.)

C.3.1.1. Data Repository

The data repository stores the data from each of the tools integrated into the platform. Tool data can be separated into graphical and model data [10] and both types of data need to be managed by the system, giving full control over the models.

It is important to note that the data repository is not expected to be the primary storage medium for each integrated tool, and to which each tool implementation needs to conform. Similar to the a-posteriori approach in [24], an integrated tool is self-sustained, and is only a-posteriori integrated through an adaption layer (See next subsection).

The content of a model is generally defined using a specific meta-model that reflects its internal structure and constraints of how modelling elements can be combined to form a valid model. In many tools such as in Simulink [11], a meta-model is implicitly assumed, while others, such as any UML tool [3], are strongly based on a given meta-modelling framework.

This meta-model acts as a basis for the data schema used by a tool to internally manage and store the model contents. Similarly, the MDM system managing an integrated model needs to map the corresponding meta-model onto the data schema of the repository. Since different types of models assume a different meta-model, each model type would occupy a separate space in the repository with a different data structure. However, in order to simplify the specification of a schema for each integrated model, a meta-meta-model is adopted as a basis for the repository. This meta-meta-model is instantiated to reflect a given meta-model, which is then further instantiated when mapping the internal data of its tool to the information model of the repository.

We adopt a simple meta-meta-model which generalises among established meta-meta-modelling languages such as MoF [12], Dome [13] and GME [14], and based on a broad survey of modelling languages for embedded computer systems [15]. A model can be generally viewed as consisting of a hierarchical structuring of modelling objects that may possess properties; ports defining interfaces of these objects; and relationships (such as associations, inheritance and refinement) between ports. Modelling languages differ in the kinds of objects that can be specified, their relationships and the kind of properties they possess. When integrating a particular model, a meta-model is instantiated by defining the kind of objects, ports and relations that exist in a model. (Note that the main aim is not to suggest yet another meta-meta-model that claims to cover any modelling language. A simple, generalised meta-meta-model was adopted, allowing focus to be placed on the PDM/SCM integration aspects of the platform.)

Figure 34 shows a UML class diagram of the object types, attributes and relations defining the generic meta-meta-model. As an example, the lower part of the figure illustrates the meta-model of a Data Flow Diagram (DFD) [21] model as interpreted by the Simulink tool [11], which is defined by specialising the generic objects.

In this approach, the granularity at which the MDM system operates on the models is controlled by the definition of the meta-model, implemented in the adaption layer. MDM mechanisms will understand the model semantics down to the level at which the elements, ports and properties are defined. Finer semantics within these entities are not the concern of MDM. For example, if a property of an element is defined as a blob of text, an MDM functionality cannot be expected to interpret the detailed semantics of this property.

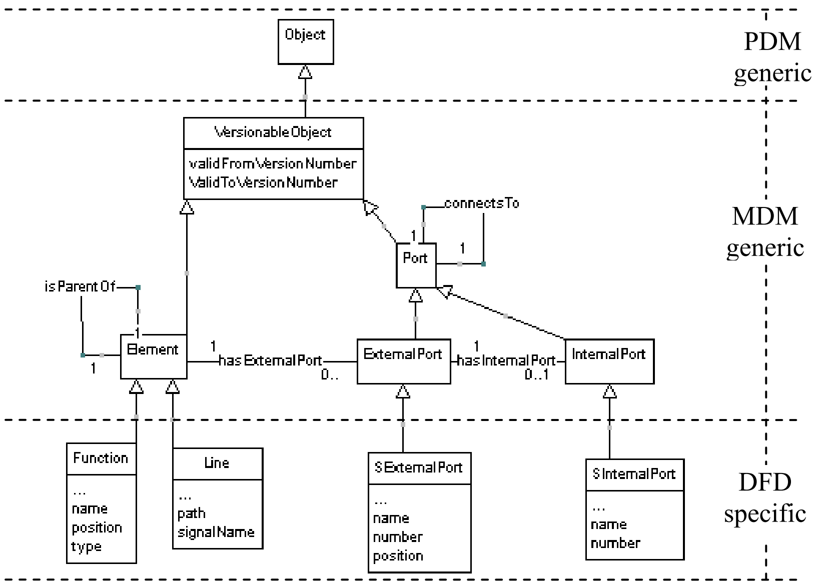


Figure 34. The PDM information model implementing the meta-meta model, and showing how a tool-specific meta-model is defined.

Adopting a common meta-meta-model between the models is not sufficient if there is a need to integrate the various model contents into a whole. For this to be possible, a unified information model of the set of models is necessary, specifying more detailed semantics of the models and their interrelations. While such information models are standardised for hardware products [16], no such standard model is currently available that also encompasses models from the software discipline. In [17], ongoing work on how such integration can be achieved is presented.

C.3.1.2. Tool Interface

Access to the tool data and the mapping of this data to the repository is performed by an *adaption layer*. An adaption layer is developed for each tool to be integrated into the MDM system. This layer isolates the tool-specific issues allowing MDM to operate generically on many tools implementing different technologies. The adaption layer fulfils three purposes. As discussed in the previous section, it maps the specific meta-model of its designated tool onto the repository.

Second, the adaption layer maps the tool-specific format used internally to manage the model data to a generic format of the repository. In this way, the management functionalities can operate uniformly using a single format.

Different technologies are available for a tool to internally store its model data. A tool can use either a computer file system to store model data in a file, or a database management system. Various standards exist that specify how data should be handled using these technologies, yet one cannot assume that tools will not implement their own solutions.

In a set of tools in which the tools adopt a combination of technologies (standard or not), it becomes necessary to translate these technologies onto a common format, in order to make the interface to the MDM platform generic. This role is fulfilled by the adaption layer, making the tool-specific data technology transparent to the rest of the platform. The adaption layer translates the format used by its designated tool to the chosen format of the repository. In the platform advocated in this paper, we adopt the data neutral XML format to interface the adaption layer to the repository.

Third, the adaption layer accommodates the different techniques used to gain access to the tool's internal data. Different tools use different technologies to provide automated access to its internal data. In the simplest case, the adaption layer can access and interpret the textual file produced by the tool. A tool can also provide 'export' functionality, an Application Programming Interface (API), or a query language.

For a potential tool to be integrated into the MDM system, specific automation support is expected. In order to allow fine-grained accessibility to parts of models and the manipulation of models, a modelling tool whose models are to be managed need to:

- Provide access to the model data either through an API or using parsable text [16].
- Provide fine-grained mechanisms for the construction and modification of models through an API.

Again, in this a-posteriori approach, an integrated tool needs neither to conform to the meta-meta-model, nor format, nor data access approach adopted by the MDM platform. Such demands would create a tight, undesired, dependency between the integrated tools and the platform. It is the adaption layer's role to map these technologies to those of the platform.

C.3.1.3. Management Functionalities

MDM functionalities ought to generically store and handle models from the various tools and disciplines. The functionalities of the union of typical SCM and PDM tools would include: Version management, product structure management, build management, change management, release management, workflow and process management, document management, concurrent development, configuration management and workspace management [4].

The model-based approach to data management unifies the disciplines by unifying the kinds of objects it manages – models. The management functionalities should focus on the models and their contents, transparent of the file structure used to store them.

While as much of the functionalities can be shared by the disciplines, discipline-specific functionalities need still to be supported such as build management for the software discipline. In certain cases, it may also be desired to provide different solutions of the same functionality for different disciplines. For example, software development might require the complex version control mechanisms and concurrent development normally provided by SCM systems, while hardware development is satisfied with sequential revision control. There should be no problem providing different solutions in MDM, depending on the kind of data items the functionality operates on. It would however be advantageous to base the different solutions on generic mechanisms for reusability purposes. The different solutions ought to be also based on the same user interface and terminology.

In order to test the proposed architecture, we have investigated in details the version management functionality of models. This functionality is termed Model Version Control (MVC). While version control is needed in both domains, the functionality differs between SCM and PDM systems (section C.2.2). This allows us to investigate how far such mechanisms can be aligned between the disciplines. Version control is also critical since it will put to the test the other crucial factors discussed in section C.2, such as the possibility of having a common product structure and data representation. A short summary of MVC is presented in section C.3.2.

C.3.2. Model Version Control Implementation

The MDM platform is currently developed using the Matrix PDM system [19]. As shown in figure 34, it was necessary to specialise the meta-meta model provided by the tool in order to perform the desired version control algorithm.

A simple model version control functionality (MVC) has been implemented. This should be seen as an exploration of the integration possibilities of model-based development. The implementation borrows from the general ideas from the fine-grained version control algorithms such as [5] and [20]. However, instead of using conventional databases, we base our implementation on the MDM architecture.

The algorithm supports the versioning of any model that can be mapped to the meta-meta-model assumed in the platform. In the current implementation, data flow diagram (DFD) [21] models from the Matlab/Simulink [11] tool and Hardware Structure Diagram models [22] in the Dome [13] tool, are handled.

Even though each tool's models contain a different kind of modelling objects, with different set of properties, MVC operates generically on all kinds of models, owing to the adoption layer which presents the model instances using a common format and structure.

Compared to version control mechanisms in conventional SCM systems, the major difference with the model-based approach is that entities have relations between them that also need to be handled. Such relations do not exist between files in the file-based approach. In file-based version control, the versioning of an entity (file) is done independently, and does not affect the versioning of other entities in the system, since no relations exist between them. In contrast, the versionable objects of a model are interrelated and creating a new version of an object might influence the versions of others.

Similar to file-based SCM systems, by only saving the changes between versions of a model, this algorithm maintains efficient storage in the repository and avoids the duplication of information. In addition, comparison between different versions of a model can be efficiently deduced.

MVC provides mechanisms that allow a user to save and extract any part of the system model. In a 'checkin' operation, changes to the model since the last checkin operation are saved in the repository. When performing a 'checkout' operation, the specified element is reconstructed for a given version, together with its subparts, forming an XML document of the information in the repository. This document is then further transformed by the adaption layer to create a tool-specific format that can be used by the tool. The details of these operations are performed transparently to the user, allowing him/her to interface with the modelling tool's interface and format. Further details on the implemented algorithm can be found in [18].

C.3.3. Integration versus Unification

As an alternative to integrating PDM and SCM systems as proposed in [4] and [6], the MDM architecture ought to be interpreted as a unified solution that aims to support the needs of both disciplines, assuming model-based development.

The need to move from the file-based approach of SCM to focus on models instead, makes much of the mechanisms currently available technically obsolete. So, the only advantage of maintaining both systems using the integration approach would be to maintain the user interface and terminologies software engineers are accustomed to. Integration techniques struggle with trying to synchronise and balance between the two disciplines.

Instead, the unification approach imposes new common mechanisms with common terminology that are expected to be accepted by both disciplines. Naturally, this approach faces more resistance from established developers and disciplines. However, the shift to model-based development would require a paradigm change that the software community may have to face anyway.

Failures in PDM/SCM integration efforts due to cultural differences [1] ought to be seen as integration problems in the organisation itself that have to be dealt with. In the best case, a unified approach can only bring the conflicts to the surface to be dealt with appropriately.

Accepting the resistance and time it takes tool vendors to change, integration may be the first step, but the future is unification.

C.4. Related Work

SCM systems targeting models, instead of file objects, are increasingly appearing in the literature ([5], [20] and [23]). In these approaches, an information model of the documents to be handled is assumed, allowing for the management of the internal information stored in the documents, as well as the specification of relations between information from different documents. While focused on software models, these approaches are helpful since the mechanisms can be extended to apply to any kind of models throughout the development lifecycle. The MVC implementation advocated in this paper is inspired by these approaches, broadening their use for more general model types. More importantly, basing the implementation on the facilities already available in PDM systems, instead of using conventional databases, helps in the integration with the mechanisms in the discipline of hardware development.

In [4], three techniques of integrating PDM and SCM systems are proposed. Of these, the full integration technique was considered ideal and most desired. In the full integration solution, the systems' functionalities are separated from their own repositories, and reintegrated into a common repository with a common information model. A common user interface is also built on top, in order to give all users a common look and feel. However, it is argued that full integration is difficult to implement using today's tools due to the tight integration of the tools' components. All the suggested approaches accept the status quo of software and hardware development and consequently needed to deal with fundamental differences. This lead to limited integration success. Rejecting the status quo and focusing on the commonality between the disciplines (model-based development), should instead lead to a smoother integration.

In [6], a configuration management system is suggested that can be applied to both software and hardware design documents. The system also allows for relationships, such as dependencies, to be established between documents. However, the entities handled by the system are file documents with no fine-grained management of their content.

C.5. Conclusion

In multidisciplinary development, the integration of the various management systems used by different disciplines is of critical value. An integrated environment allows the efforts of all developers to be well communicated and reduces any risks of inconsistencies and conflicts between them.

Due to the difference in maturity levels of these disciplines, such integration efforts has had limited success in the past. Specifically, the implementation-centred development approach of software systems expected a coarse-grained support from SCM systems, where documents are the smallest entities managed, while ignoring the internal model semantics contained within them. In comparison, mechanical development expects the handling of the detailed product data by their corresponding PDM systems using standard information models.

However, with the move towards model-based development, where the use of models becomes the central activity in making, communicating and documenting de-sign decisions, disciplines share a common need to handle the same kind of entities – models. In this way, management systems can be brought closer together.

This paper presented an architecture for a Model Data Management (MDM) system that aims to provide the support functionalities expected of a model-based development environment. The system aims to generically handle and control the various kinds of models produced by the different tools during the development of software-intensive, yet multidisciplinary, products. The proposed architecture builds on existing technologies from the more mature discipline of mechanical engineering, while borrowing new ideas from the software engineering discipline.

To illustrate the MDM solution, an initial implementation of a Model Version Control (MVC) functionality was performed, allowing for the fine-grained version management of two types of models from two different tools. MVC permits stakeholders to perform design activities in terms of models, where they can organise, share and modify their models, transparent to the underlying file structure. A simplified version control functionality has been realised. The ability to perform branches and merges in the changes of an element is a very important feature of version control, specifically desired in software development. This is needed in order to study different design alternative, provide product variants, or deal with a bug fix from an earlier release. MVC needs to handle this functionality in the future.

The major aim of the current platform implementation has been to experiment and illustrate the concepts discussed in this paper. While the current implementation has only been validated through the use of a small case study, a more commercial size case study would be needed to appropriately validate the usability of this approach. This remains to be done in the future.

The advantage of MDM over conventional PDM/SCM systems is the inclusion of the internal content of its supported models, allowing for a tighter integration of the design information between different models. In addition, functionalities are generically applicable for many kinds of models, simplifying the process of adding new tools into the toolset. However, an initial effort is required to integrate new models in the development of the adaption layer. The fine-grained management of models is bound to require more computational effort that the coarse-grained approach.

The development process of software and hardware products will always differ due to the nature of the products themselves. However, in a unified approach the same mechanisms ought to be used to support these differing processes. Moreover, by providing different strategies for different kinds of models, the development needs of both disciplines can be satisfied, using variants of the same basic mechanisms in a unified management system. It is essential however to base the strategies on the same basic mechanisms and user interface, allowing the reuse of basic components and preventing confusion in terminologies.

In the case where development is not (completely) model-based, MDM facilities may still be used. Any product data inputted into the platform is restructured and interpreted to form model data. For example, a MDM system can manage the files of a Java project by reinterpreting each file as a class model, extracting and managing the attributes and methods contained within each file as fine-grained structured data.

The approach is currently implemented using a PDM system. It is our ideal vision that with the acceptance of model-based development, one no longer needs to discuss the integration of PDM and SCM systems. Instead, a truly unified approach to model data management can be used by both disciplines.

C.6. Acknowledgements

This work has been supported by the Swedish Strategic Research Foundation, through the SAVE project.

C.7. References

[1] Dahlqvist, A.P., Crnkovic, I. and Asklund, U., Quality Improvements by Integrating Development Processes, 11th Asia-Pacific Software Engineering Conference, 2004.

[2] OMG, Model Driven Architecture Specification, MDA Guide Version 1.0.1, Document Number: omg/2003-06-01, June 2003.

[3] OMG, Unified Modeling Language (UML) Specification, V1.5, March 2003.

[4] Crnkovic I., Asklund U. and Persson Dahlqvist A., Implementing and integrating product data management and software configuration management, Artech House Publishers, 2003.

[5] Ohst D. and Kelter U., A fine-grained version and configuration model in analysis and design, Proceedings of the International Conference on Software Maintenance, 2002.

[6] Westfechtel B. and Conradi R., “Software Configuration Management and Engineering Data Management: Differences and Similarities” Proceedings 8th International Workshop on System Configuration Management, Springer-Verlag, pages 95-106, 1998.

[7] Kemmerer S. J. (editor), “STEP, the grand experience”, National Institute of Standards and Technology, special publication 939, 1999.

[8] Estublier J., Favre J. M. and Morat P., Toward SCM / PDM integration?, International Workshop on Software Configuration Management, (SCM8), Springer Verlag, 1998.

[9] Kruchten, P., Casting Software Design in the Function-Behavior-Structure Framework, IEEE Software, Volume 22, Issue 2, 2005.

[10] Ohst D., Welle M. and Kelter U., “Differences between Versions of UML Diagrams”, Proceedings of the joint European software engineering conference (ESEC) and SIGSOFT symposium on the foundations of software engineering (FSE-11), 2003.

[11] Simulink, Mathworks, http://www.mathworks.com/products/simulink/, accessed March 2005.

[12] OMG, Meta Object Facility (MOF) Specification, V1.4, April 2002.

[13] Dome, “Dome Guide” Version 5.2.2, http://www.htc.honeywell.com/dome/index.htm, 1999.

[14] GME, A Generic Modeling Environment, GME 4 User’s Manual, Version 4.0, Institute for Software Integrated Systems, Vanderbilt University, 2004.

[15] El-khoury J., Chen D. and Törngren M., “A survey of modelling approaches for embedded computer control systems (Version 2.0)” Technical report, ISRN/KTH/MMK/R-03/11-SE, TRITA-MMK 2003:36, ISSN 1400-1179, Department of Machine Design, KTH, 2003.

[16] Kemmerer S. J. (editor), STEP, the grand experience, National Institute of Standards and Technology, special publication 939, 1999.

[17] El-khoury J., Redell O. and Törngren M., A Tool Integration Platform for Multi-Disciplinary Development, to be published, 31st Euromicro Conference on Software Engineering and Advanced Applications, 2005.

[18] El-khoury J and Redell O., A Model Data Management Architecture for Multidisciplinary Development, Internal Technical Report, Mechatronics Lab. Royal Institute of Technology, Stockholm. 2005.

[19] MatrixOne, Matrix10, http://www.matrixone.com/, accessed April 2005.

[20] Nguyen T. N., Munson E.V., Boyland J.T. and Thao C., Flexible Fine-grained Version Control for Software Documents, 11th Asia-Pacific Software Engineering Conference, 2004.

[21] Cooling J., Software Engineering for Real-time Systems. Pearson Education Limited, ISBN 0201596202, 2003.

[22] Redell O., El-khoury J. and Törngren M., The AIDA toolset for design and implementation analysis of distributed real-time control systems, Microprocessors and Microsystems, Volume 28, Issue 4, 2004.

[23] Chien S. Y., Tsotras V. J., Zaniolo C., Version Management of XML Documents, Third International Workshop WebDB 2000 on The World Wide Web and Databases, 2000.

[24] Becker S. M., Haase T. and Westfechtel B., Model-based a-posteriori integration of engineering tools for incremental development processes, Journal of Software and Systems Modeling, Volume 4, Number 2, Springer, 2005.

Paper-D

The Version Control Algorithm Implementation in the Model Data Management (MDM) Platform

El-khoury Jad

Published as
Technical report, ISRN/KTH/MMK/R-05/27-SE, TRITA-MMK 2005:27, ISSN 1400-1179, Department of Machine Design, KTH, 2005.

Abstract

An architecture for a Model Data Management (MDM) system that supports model-based development is being developed. The system aims to generically handle the models produced by the different tools during the development of software-intensive, yet multidisciplinary, products. The proposed architecture builds on existing technologies from the mature discipline of mechanical engineering, while borrowing new ideas from the software domain. To illustrate the architecture, an initial implementation of a Model Version Control (MVC) system, which allows the fine-grained version management of models, is developed.

D.1. Introduction - Model Data Management Functionality

This paper presents the model-based version control algorithm implemented in the Model Data Management (MDM) platform, presented in [1].

The MDM platform aims to generically store and handle models from the various tools used in the development of mechatronics products. Such a platform is viewed as a unification of the management functionalities typically provided by the discipline-specific PDM and SCM systems traditionally used in the hardware and software disciplines respectively. The model-based approach to data management unifies the software and hardware disciplines by unifying the kinds of objects it manages – models.

The functionalities of the union of typical SCM and PDM tools would include: Version management, product structure management, build management, change management, release management, workflow and process management, document management, concurrent development, configuration management and workspace management [2].

In order to test and exemplify the proposed architecture, we investigate in details the version management functionality of models. This functionality is termed Model Version Control (MVC). While version control is needed in both domains, the functionality differs between SCM and PDM systems. This allows us to investigate how far such mechanisms can be aligned between the disciplines.

Naturally, a full validation of the approach needs to investigate the feasibility of the remaining management functionalities using the model-based approach. However, version control is most fundamental and best validates our MDM approach. It will put to the test the other crucial factors discussed in [1], such as the possibility of having a common product structure and data representation.

The aim of the current implementation is to investigate the potential of implementing model-based management functionalities using the technology offered by a typical PDM system. Specifically, the implementation of a fine-grained version control algorithm is investigated. As a result, the extensions necessary for a PDM system in order to support such functionality will be highlighted.

The MDM architecture is shown in figure 35. The platform consists of two main parts: A set of tool-specific *adaption layers* and a *data repository* with mechanisms to handle the stored data. The data repository stores the data for each of the tools. To perform this role in a generic way, the data from the different tools is expected to be presented in a neutral form, and this functionality is provided by the adaption layer. Triggered either by a tool or the repository, the corresponding adaption layer permits the data flow between a tool and the repository, in a predefined format.

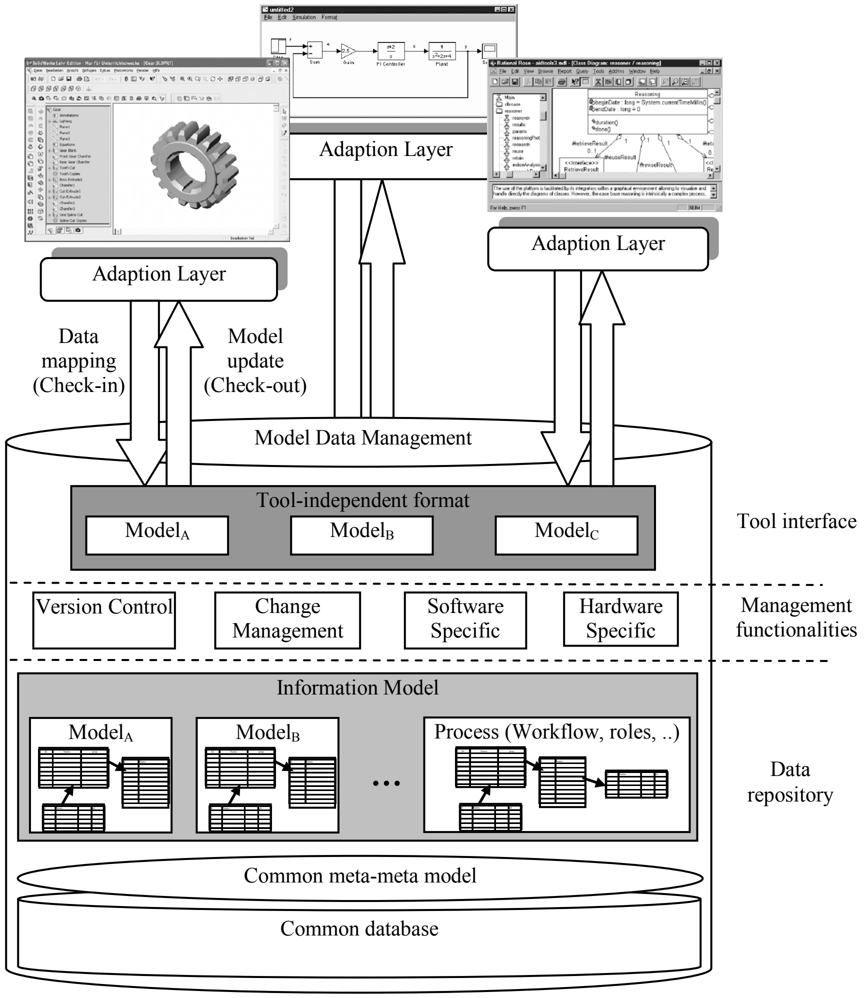


Figure 35. The major components of the MDM architecture.

D.2. Model Version Control

A simple MVC algorithm is implemented. This should be seen as an exploration of the integration possibilities of the model-based approach to data management.

The algorithm supports the versioning of any model that can be mapped to the meta-meta-model assumed in the platform.

Even though each tool’s models contain different kinds of modelling objects, with different set of properties, MVC operates generically on many model types. This owes to the adaption layer which presents the model instances using a common format and structure.

D.2.1. Meta-model

The content of a model is generally defined using a specific meta-model that reflects its internal structure and constrains how modelling elements can be combined to form a valid model. In many tools such as Simulink [3], a meta-model is implicitly assumed, while others, such as any UML tool [4], are strongly based on a given meta-modelling framework. The meta-model acts as a basis for the data schema used by a tool to internally manage and store the model contents.

Similarly, the MDM system managing an integrated model needs to map the corresponding meta-model onto the data schema of the repository. Since different types of models assume a different meta-model, each model type would occupy a separate space in the repository with a different data structure. However, in order to simplify the specification of a schema for each integrated model, a meta-meta-model is adopted as a basis for the repository. This meta-meta-model is instantiated to reflect a given meta-model, which is then further instantiated when mapping the internal data of its tool to the information model of the repository.

In the current implementation, the instantiation to a specific meta-model is not explicitly specified in the system, and is only implicitly assumed by the corresponding adaption layer of a tool when performing the mapping between the tool-specific meta-model and that assumed by MDM.

The MVC functionality (or any other MDM functionality for that matter) is expected to work generically on any instance of the meta-meta-model. This is made possible by providing the instantiation information for a specific meta-model (such as the specific element type names) as input to the generic algorithm through an adaption layer. Section D.2.5 further discusses the process of mapping a specific meta-model to this meta-meta-model.

We adopt a simple meta-meta-model which generalises among established meta-meta-modelling languages such as MoF [5], Dome [6] and GME [7], and based on a broad survey of modelling languages for embedded computer systems [8]. In view of this meta-meta-model, a model can be considered as consisting of a hierarchical structuring of elements that may possess properties; ports defining interfaces to these elements; and relations (such as associations, inheritance and refinement) between ports. Modelling languages differ in the kinds of elements that can be specified, their relationships and the kind of properties they possess. When integrating a particular model, the meta-meta-model is instantiated to reflect a given meta-model by defining the kind of elements, ports and relations that will exist in that particular model. The meta-model is then further instantiated when defining a specific model for a specific system. Figure 36 shows a class diagram of the object types, attributes and relations defining the generic meta-meta-model. A more detailed and formalised definition of the meta-meta-model can be found in [9].

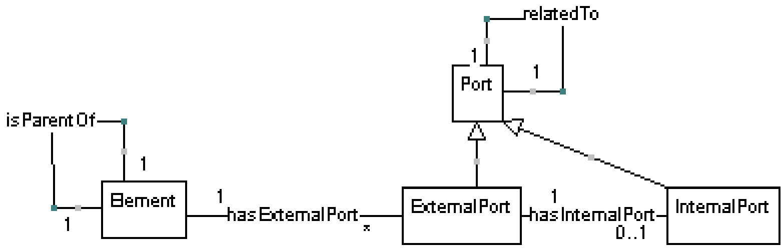


Figure 36. The generic meta-meta-model represented as a class diagram.

In this approach, the granularity at which the MDM system operates on a particular model is controlled by the definition of its respective meta-model, implemented in the adaption layer. MDM mechanisms will understand the model structure down to the level at which the elements, ports and properties are defined. Finer structures within these entities are not the concern of MDM. For example, if a property of an element is defined as a blob of text, an MDM functionality cannot be expected to interpret the detailed semantics of this property. This provides the necessary flexibility when integrating a particular model to decide on the level of granularity at which the MDM functionalities ought to operate.

D.2.2. Versioning Model

The MVC algorithm supports the following operations to be performed on a model:

- An element can be created as a child of a parent element.
- An element can be deleted.
- An element can be modified, when one of its properties are modified.
- An external port can be created for a containing element.
- An external port can be deleted.

- An external port can be modified, when one of its properties are created, deleted or modified.
- An internal port can be created for an external port.
- An internal port can be deleted.
- An internal port can be modified, when one of its properties are created, deleted or modified.
- A *relatedTo* relation can be created between a set of external/internal ports.
- A *relatedTo* relation can be deleted.

The following operations are not supported:

- Moving an element from one parent to another (Changing the end entities of an *isParentOf* relationship).
- Moving an external port from one containing element to another (Changing the end entities of a *hasExternalPort* relationship).
- Moving an internal port from one external port to another (Changing the end entities of a *hasInternalPort* relationship).
- Exchanging one or more of the end entities of a *relatedTo* relationship.
- Modifying a *relatedTo* relationship, by creating, deleting or modifying one of its properties.

We refer to *versionable objects* as those objects of the model to which new versions can be created when changes to them occur. In the current approach, only elements, ports (external ports and internal ports) and the *relatedTo* relations of a model are versionable. Currently, a *relatedTo* relation can be created and deleted between different versions of the system. However, changes to a relation's properties are ignored and hence change operations cannot be version controlled. The reason of this limitation is mainly due to the current PDM system being used, in which the built-in relations cannot have revisions. Allowing versions of relations would require the further extension of the provided facilities. While not technically impossible, this extension is left for future development. Given the current implementation, in the case where relations need to be versioned, then they ought to be defined as elements.

The remaining relationships (*isParentOf*, *hasExternalPort*, *hasInternalPort*) need not be versioned since such relationships do not contain properties that can change. The creation/deletion of such a relationship reflects a creation/deletion of one of its end entities. Such actions are version controlled through the versioning of the end entities. (The *relatedTo* relation differs from these relationships in that the creation/deletion of such a relation cannot be reflected in any one of its end entities, and hence need to be specified at the relation itself.)

A move operation, in which one end of the relationship is changed from one entity to another, is not supported. The version control of move operations can only be supported by version controlling the relationships. Such support would also allow changes to the end elements of a *relatedTo* relation to be supported. While it may be desired, the current implementation does not support such move operations. Instead, a move is interpreted as a deletion of the old relationship and recreation of a new one.

Finally, an attribute is not a versionable object. While changes to a versionable object's attributes are used to define whether a new version of the object is created or not, the attributes are not versioned themselves.

Figure 37 shows an extension to the class diagram of figure 36, highlighting the properties of versionable objects, to be discussed in this section.

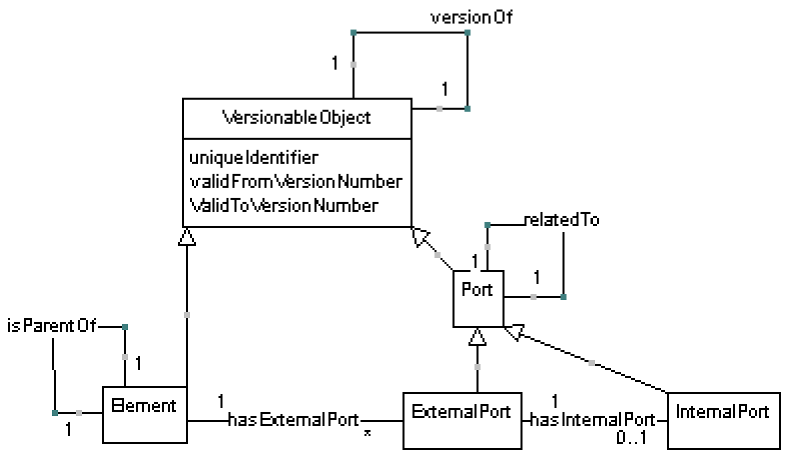


Figure 37. Extending the adopted meta-meta-model to support the MVC algorithm. The *relatedTo* relationship has the attributes *validFromVersionNumber* and *validToVersionNumber*, not illustrated in the model.

Each versionable object needs to have a *uniqueIdentifier* that is unique within its context. The *uniqueIdentifier* of an element is unique in the context of its parent element, meaning that no two sibling elements can have the same identifier. The context of an external port is its containing element. An internal port's context is its external port, and hence no *uniqueIdentifier* is needed since there is a maximum one-to-one relationship between them. Since changing the end entities of a *relatedTo* relation is not supported, a relation can be fully identified by its end elements. For this reason, relations need not have a *uniqueIdentifier*.

Compared to version control mechanisms in conventional SCM systems, the major difference with the model-based approach is that the entities to be versioned have relations between them that also need to be handled. Such relations do not exist between files in the file-based approach. In file-based version control, the versioning of an entity (file) is done independently, and does not affect the versioning of other entities in the system, since no relations exist between them. In contrast, the versionable objects of a model are interrelated and creating a new version of an object might influence the versions of others. For example, when a given element's properties are changed, not only has this element changed, but the parent element can be considered as changed, although none of its direct properties has. No new version of the parent needs to be created, but it becomes necessary to indicate that the parent's version is valid for both the child's old version as well as the new version. This change propagation is recursively performed up the hierarchy, which implies that every time a set of changes is performed on elements of a model (and new versions are created), a new version number (but no new object versions) of the entire model needs to be created, in order to encompass these changes.

We differentiate between a *direct change* to a versionable object in which the object's properties are changed, and an *indirect change* in which a change to one of its dependent versionable objects is changed (in turn either directly or indirectly changed).

- An element is indirectly changed if one of its direct children or one of its direct external ports is changed (either directly or indirectly).
- An external port is indirectly changed if its internal port is changed.
- A *relatedTo* relation cannot be changed in the current implementation. (Once supported, changing any of its end entities would be considered an indirect change.)

A consequence of these change rules is the propagation of indirect changes from any entity up in the model hierarchy. Since a model is defined as a single rooted tree, any direct change to any entity in the model implies an indirect change to the root of the model. This in turn implies that the complete model has changed.

In order to handle the propagation of changes in the hierarchy, two kinds of versions need to be differentiated:

- A *local version* of an object that is associated with direct changes to the object, ignoring its related objects. (Denoted by V_1, V_2, V_3, etc.). Each local version manages the corresponding set of valid properties.
- A *global version* of the entire model that is associated with the set of local versions of each object in the hierarchy that forms a consistent version of a complete model (denoted by the numbers 1, 2, 3, etc.).

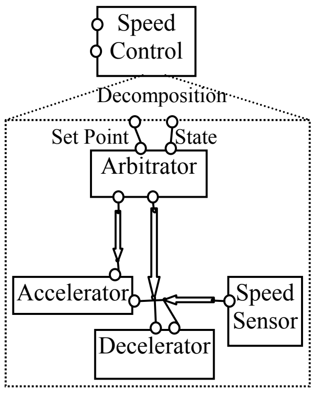
Note that one talks about the 'local version of an object' and the 'global version of the entire model'. In addition, since the number of local versions (V_1, V_2, …) is less relevant, the terms 'local object version' and 'global version number' can be shortened in this discussion to 'object version' and 'version number' respectively.

Now, whenever a direct change to an object occurs, a new local version is created for that object. This new local version records the new properties of the entity. Since any single direct change implies an indirect change - and hence a new version - of the complete model, a new global version is also created. The new global version is associated with the new local version of the directly changed entity, as well as the previous local versions of any other unchanged entity in the hierarchy. In the case when more than one entity is directly changed (within a single check-in operation), a single global version number is created, which is associated to these new local versions and the remaining unchanged objects.

Each local version of a versionable object has a *validFromVersionNumber* and *validToVersionNumber* property defining the range of global version numbers for which the local version is valid. Only one local version in the sequence of local versions of an object can be valid for a specific global version of the model. It is possible to represent the set of global versions valid for each local version as a range of numbers - [*validFromVersionNumber validToVersionNumber*] - assuming the global version number is incremented between different versions of the model. This is because once a local version, V_n, is created and made valid for the new *validFromVersionNumber* global version, V_n remains the valid version to use for any newer global version until a new local version, V_{n+1}, is created. Once the newer V_{n+1} is created, V_n's range is closed and it cannot become valid for any new global version.

For example, in figure 38, the local version V_1 of the element *Speed Control* is valid for more than one global version (1 to 3) since its children elements have changed in these versions without any changes being performed on the element itself. This also means that the number of local and global versions of an object is not necessarily the same.

In the implementation of the algorithm, if an object's version is valid for the latest global version number of the model, then the *validToVersionNumber* of that version is set to ∞. This feature is used simply to avoid updating the *validToVersionNumber* of all the objects in the model, whenever a new global version number is created, unless an object is deleted or a new local version is created. For example, in figure 38, assuming a new global version number 6 is created, the last version V_3 of *Speed Control* is made automatically valid for this new version number.



		V_1	V_2	V_3
Speed Control	validFromVersionNumber	1	4	5
	validToVersionNumber	3	4	∞
Arbitrator	validFromVersionNumber	1	3	
	validToVersionNumber	2	3	
Speed Sensor	validFromVersionNumber	3		
	validToVersionNumber	∞		

Figure 38. The relationships between the local and global versions of elements in the model hierarchy.

Note that the range of valid versions of a parent includes the range of any of its children since a child cannot exist without its containing parent. When an element is deleted, *validToVersionNumber* is set to the last version at which the element exists. For example, in figure 38, the *Arbitrator* element is deleted in the global version 4. This is deduced since there exists a version 5 of the parent *Speed Control* element, but the versions of *Arbitrator* terminate at 3. The above discussion also applies for the other relations between versionable objects (between an element and its external ports, and between an external port and its internal port).

A consequence of this versioning model is that whenever a user performs any small set of changes to any parts of the model, a new version number of the complete model is created. Elements from the other unchanged branches of the model hence inherit this new version number, yet no changes have been performed on them or any of their own children. This behaviour, while necessary, may be confusing, and need to be handled appropriately by good tool support. (For a given element, the tool can easily work out the set of relevant version numbers for which the element or any of its children has changed.)

The approach of assuming a global configuration for the entire model tree is similar to modern file-based SCM systems such as Subversion [10], in which any changes to any of the files, is associated with a global version number of the entire file tree. Each version number *N* represents the state of the repository/ file-system after the N^{th} commit.

Similar to file-based SCM systems, by only saving the changes between versions of a model, the versioning model maintains efficient storage in the repository and avoids the duplication of information. In addition, comparison between different versions of a model can be efficiently performed. In this way, additional functions such as the merging of models and the visual presentation of model changes can be readily implemented.

MVC provides mechanisms that allow a user to save and extract any part of the system model through *check-in* and *check-out* operations respectively. These operations will be further detailed in section D.2.3.

D.2.3. Versioning Algorithm

As discussed in section D.2.1, a model can be viewed as a tree structure of elements forming a parent-child relationship. In the versioning algorithm, the model tree is assumed to be single rooted. This assumption forms no limitation given that the platform aims to support multiple models and model types. However, each root contains its own global version number, and changes in one rooted tree are not directly reflected in another.

Since the focus of this work is not in developing new versioning algorithms, we attempted to borrow from existing efforts, while ensuring their broad usage for more general model types. The MVC algorithm is mainly inspired by the algorithm in [17], borrowing ideas such as assuming a unique identifier for each node in the model tree and versioning all changes within an editing session as a single change. However, instead of using conventional databases, the implementation is based on the MDM platform facilities (and indirectly on the facilities already available in a PDM system).

D.2.3.1. Check-out Operation

The general behaviour of the check-out operation is illustrated in figure 39. When performing a check-out operation, the user is first prompted for the element in the model hierarchy stored in the repository to be checked-out, as well as the particular global version number for that element.

The *saveWorkspaceInformation* operation saves relevant information about the checked-out element together with the model outputted. This information is later used in the check-in operation to identify the location of the element just checked-out in the repository. Different techniques can be used to save the workspace information. One solution is to save a special text file in the file system to control the list of the checked-out elements. For each element, the location in the file system where the checked-out model is placed, the element's identifier in the repository and the version number with which it is checked-out are stored. The specific details of managing the workspace information will not be discussed in this report.

```
void checkout() {
  Element element = promptForElement();
  int versionNumber = promptForVersionNumber(element);
  saveWorkspaceInformation(element, versionNumber);
  checkoutElement(element, versionNumber, outputFile);
}
```

Figure 39. Pseudocode for the *check-out* operation

Once the element and global version number are determined, the *checkoutElement* operation (illustrated in figure 40) is performed.

```
public void checkoutElement (Element element,
                 int versionNumber, FileWriter out) {
  checkoutProperties(element, versionNumber);

  loop ∀p ∈ externalPorts(element, versionNumber)
    checkoutExternalPort (p, versionNumber, out);

  loop ∀c ∈ directChildren(element, versionNumber)
    checkoutElement (c, versionNumber, out);

  loop ∀r ∈ internalRelations(element, versionNumber)
    checkoutInternalRelation (r, versionNumber, out);
}
```

Figure 40. Pseudocode for the *checkoutElement* operation

The first step in the *checkoutElement* operation is to check-out the element properties. This is followed by checking-out each of the element's external ports (illustrated in figure 41), as well as its direct children and internal relations. Note that *checkoutElement* operates recursively over the direct children of the element.

```
public void checkoutExternalPort (ExternalPort externalPort,
                                int versionNumber, FileWriter out) {
  checkoutProperties(externalPort, versionNumber);

  loop ∀p ∈ internalPorts(externalPort, versionNumber)
    checkoutInternalPort (p, versionNumber, out);
}
```

Figure 41. Pseudocode for the *checkoutExternalPort* operation

D.2.3.2. Check-in Operation

In a *check-in* operation, changes to the model since the last check-in operation are saved in the repository. The general behaviour of the check-in operation is illustrated in figure 42. The first step is to determine the position in the existing tree structure where the model sub-tree, with root *modelRoot*, needs to be checked-in. This can be determined based on information already saved in the user workspace. In the case where *modelRoot* is an update of an existing sub-tree in the repository that has been checked-out earlier, the workspace would contain the necessary information, produced during the last check-out operation (as detailed in section D.2.3.1). In the case where a new sub-tree needs to be added, the workspace would contain no such information and the user is prompted for the location in the tree structure of the repository (A new model is created in the repository in this way.).

```
void checkin (Node modelRoot) {
  Element parent = findParent (modelRoot, Workspace);
  if (parent == null)
    parent = promptUserForParent();
  boolean newVersionCreated = checkinChild (parent, modelRoot);
  if (newVersionCreated)
    incrementGlobalVersionNumber();
}
```

Figure 42. Pseudocode for the *check-in* operation

Once the parent of *modelRoot* is determined, the *checkinChild* operation (illustrated in figure 43) is performed. The operation returns a boolean value indicating whether the model was modified in any way since the last check-out. In this case, the complete model is considered modified, and the global version number of the complete model is incremented.

```
boolean checkinChild (Element parent, Node modelRoot) {
  incrementGlobalVersionNumber = false;

  Element child = locateChild (parent, modelRoot);
  if (child == null) {
    incrementGlobalVersionNumber = true;
    child = createNewChild(parent, modelRoot);
  }
  elseif (changed(child, modelRoot)) {
    incrementGlobalVersionNumber = true;
    createNewVersion (child, properties(modelRoot));
  }

  //checkinDirectChildren
  loop ∀c∈directChildren(modelRoot) {
    if (checkinChild (child, c))
      incrementGlobalVersionNumber = true;
  }
  loop ∀c ∈ directChildren(child)
            s.t. c ∉ directChildren(modelRoot) {
    incrementGlobalVersionNumber = true;
    delete(c);
  }

  //checkinExternalPorts
  loop ∀p ∈ externalPorts(modelRoot) {
    if (checkinExternalPort (child, p))
      incrementGlobalVersionNumber = true;
  }
  loop ∀p ∈ externalPorts(child)
            s.t. p ∉ externalPorts(modelRoot) {
    incrementGlobalVersionNumber = true;
    delete(p);
  }

  //checkinInternalRelations
  loop ∀r ∈ internalRelations(modelRoot) {
    if (checkinInternalRelation (child, r))
      incrementGlobalVersionNumber = true;
  }
  loop ∀r ∈ internalRelations(child)
            s.t. r ∉ internalRelations(modelRoot) {
    incrementGlobalVersionNumber = true;
    delete(r);
  }

  return incrementGlobalVersionNumber;
}
```

Figure 43. Pseudocode for the *checkinChild* operation

The first step in the *checkinChild* operation is to version control the *modelRoot* node itself. An attempt is made to locate the *modelRoot* node to one of the designated parent's direct children. The *locateChild* operation searches the direct children of *parent* for the child that equals the *uniqueIdentifier* signature of *modelRoot*.

If no such element is found, then *modelRoot* is considered to be a new child of *parent*. *Parent* is hence modified by creating a new direct child with the properties of *modelRoot*. The operation *createNewChild(parent, modelRoot)* creates a new child of *parent*. The relationship between the new child and its parent, together with the values of their *validFromVersionNumber* and *validToVersionNumber* before and after the *createNewChild* operation are illustrated in figure 44.

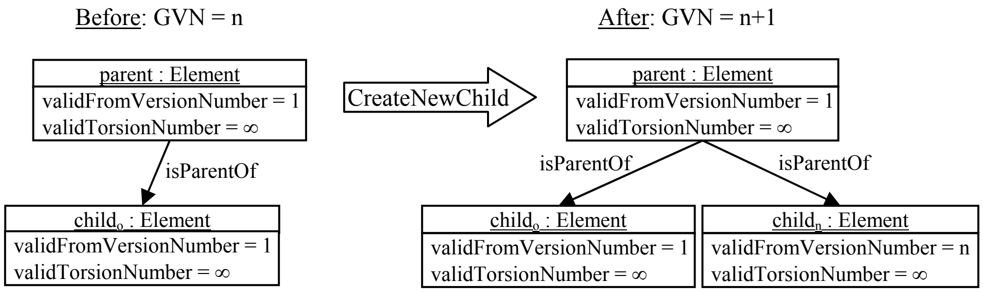


Figure 44. The effect of performing the *createNewChild(parent, child$_n$)* operation, illustrated as an Object Diagram.

If a corresponding child is found, then the child's properties are compared with those of *modelRoot*. A change in any of the properties since the last version of the element would indicate that a new version of the existing child needs to be created. The operation *createNewVersion(child, properties(modelRoot))* creates a new version of *child*. The relationship between *child* and the new version, together with the values of their *validFromVersionNumber* and *validToVersionNumber* before and after the createNewVersion operation are illustrated in figure 45.

Second, the *checkinChild* operation is operated recursively over the direct children of *modelRoot*, where each direct child is assumed to be a *modelRoot* to be checked-in as a child of the current *child* element. In addition, any direct child of *child* that is no longer a direct child of *modelRoot* is considered deleted. The operation *delete(element)* deletes *element* in the new version of the model. The values of *element*'s *validFromVersionNumber* and *validToVersionNumber* before and after the delete operation are illustrated in figure 46.

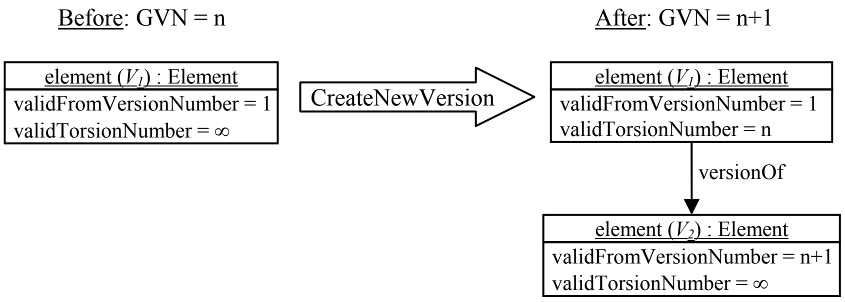

Before: GVN = n
After: GVN = n+1
element (V_1) : Element
validFromVersionNumber = 1
validTorsionNumber = ∞
CreateNewVersion
element (V_1) : Element
validFromVersionNumber = 1
validTorsionNumber = n
versionOf
element (V_2) : Element
validFromVersionNumber = n+1
validTorsionNumber = ∞

Figure 45. The effect of performing the *createNewVersion(element, newProperties)* operation, illustrated as an Object Diagram.

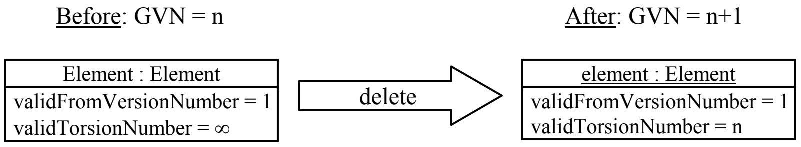


Figure 46. The effect of performing the *delete(element)* operation, illustrated as an Object Diagram.

These latter steps are analogously repeated for the external ports of *modelRoot*, where each external port of *modelRoot* is checked in as an external port of *child*, using the *checkinExternalPort* operation shown in figure 47. In addition, any external port of *child* that is no longer an external port of *modelRoot* is deleted.

Note the similarity between the *checkinExternalPort* and *checkinChild* operations. Specifically, each internal port of *externalPortNode* is checked in as an internal port of *externalPort*, using the *checkinInternalPort* operation shown in figure 48. In addition, any internal port of *externalPort* that is no longer an internal port of *externalPortNode* (The equivalent of *externalPort* in the repository) is deleted.

Once the direct children and external ports of *modelRoot* are synchronised with the *child* element in the repository, it becomes possible to also synchronise the internal *relatedTo* relations between ports of *modelRoot*'s direct children. An internal relation of an element is a *relatedTo* relation between two ports from the set of external ports of all its direct children, as well as the element's internal ports.

The internal relations synchronisation is analogous to that of the direct children (and external ports) described earlier. Each internal relation of *modelRoot* is checked in using the *checkinInternalRelation* operation shown in figure 49. In addition, any internal relation of *child* that is no longer an internal relation of *modelRoot* is considered deleted.

```
boolean checkinExternalPort (Element containingElement,
                                         Node externalPortNode) {
  incrementGlobalVersionNumber = false;
  ExternalPort externalPort = locateExternalPort(containingElement
                                      , externalPortNode);
  if (externalPort == null) {
    incrementGlobalVersionNumber = true;
    createNewExternalPort(containingElement, externalPortNode);
  }
  elseif (changed(externalPort, externalPortNode)) {
    incrementGlobalVersionNumber = true;
    createNewVersion (externalPort, properties(externalPortNode));
  }
  //checkinInternalPorts
  loop ∀p ∈ internalPorts(externalPortNode) {
    if (checkinInternalPort (externalPort, p))
      incrementGlobalVersionNumber = true;
  }
  loop ∀p ∈ internalPorts(externalPort)
              s.t. p ∉ internalPorts(externalPortNode) {
    incrementGlobalVersionNumber = true;
    delete(p);
  }
  return incrementGlobalVersionNumber;
}
```

Figure 47. Pseudocode for the *checkinExternalPort* operation

```
boolean checkinInternalPort (ExternalPort externalPort,
                                         Node internalPortNode) {
  incrementGlobalVersionNumber = false;
  InternalPort internalPort = locateInternalPort (externalPort,
                                            internalPortNode);
  if (internalPort == null) {
    incrementGlobalVersionNumber = true;
    createNewInternalPort(externalPort, internalPortNode);
  }
  elseif (changed(internalPort, internalPortNode)) {
    incrementGlobalVersionNumber = true;
    createNewVersion (internalPort, properties(internalPortNode));
  }
  return incrementGlobalVersionNumber;
}
```

Figure 48. Pseudocode for the *checkinInternalPort* operation

```
boolean checkinInternalRelation (Element parent,
                                 Node internalRelationNode) {
  incrementGlobalVersionNumber = false;

  PortRelatedTo internalRelation = locateInternalRelation (parent,
                                          internalRelationNode);
  if (internalRelation == null) {
    incrementGlobalVersionNumber = true;
    createNewInternalRelation(parent, internalRelationNode);
  }
  return incrementGlobalVersionNumber;
}
```

Figure 49. Pseudocode for the *checkinInternalRelation* operation

Note the similarity between the *checkinInternalRelation* and *checkinChild* operations. Specifically, since *relatedTo* relations cannot be versioned (as discussed in section D.2.2), the only check that can be performed is whether the internal relation is newly created.

D.2.4. Tool Implementation

The MDM platform is currently developed using the Matrix PDM system [11]. In the current implementation, Data Flow Diagram (DFD) [12] models from the Matlab/Simulink [3] tool and Hardware Structure Diagram models [13] in the Dome [6] tool are handled.

The Matrix system manages its data using a relational database. However, the database is made transparent by providing an interface allowing the modeller to indirectly define the database schema using an object-oriented approach, which is more appropriate in modelling the product data.

Two user-levels in the Matrix system are relevant for the purposes of this report:

- The Business Modeller usage level - where the user defines types of objects, along with the attributes, process rules, and persons associated with those objects. This information represents the company’s business model and is used to set up the database schema.
- The Matrix Navigator usage level - where the user creates specific instances of the object types that were defined in the Business Modeller.

At the Business Modeller usage level, the following kinds of objects can be defined:

- A *business object type* - defines the kind of *business objects* that can be instantiated at the Matrix Navigator level. The definition of a business object type includes a definition of a collection of attributes as well as an (optional) specification of the object type from which it is derived. Inheritance provides the specialising type with the attribute collection from the inherited parent as well as the relationship connections in which the parent type plays a role. Given that many types can inherit from the same parent type, an inheritance hierarchy is formed.
- A *relationship type* - defines a connection that can be made between business objects. The definition of a *relationship* determines the types of objects that can be specified at each end of the relationship as well as any attributes it may have.
- An *attribute* – defines a characteristic or property that can be assigned to an object or to its relationship with other objects.

The Business Modeller usage level is a way of extending and configuring the Matrix system for specific needs. For the implementation of the MDM platform, the information model is configured to reflect the desired meta-meta-model of the MDM platform (figure 50). The *VersionableObject* business type is defined to allow the generic definition of *Element*, *ExternalPort* and *InternalPort* with the attributes *uniqueIdentifier*, *validFromVersionNumber* and *validToVersionNumber*. The relationship types between these object types reflect the meta-meta-model definition of the platform. The *versionOf* relation is a built-in relation, used to relate the revisions history of business objects. This relation is used to relate the different local versions of objects in the platform.

Any integrated model is defined by specifying business object types that inherit from these objects. Section D.2.5 presents the necessary steps that need to be performed when integrating a specific model.

Besides the configuration capabilities of the Business Modeller, further customisation capabilities are available through an Application Development Kit (ADK), allowing programmers to write custom applications in either C++ or Java. These custom applications can be used for the integration of Matrix with other applications. The provided API provides applications with access to the product data in the repository, allowing typical operations such as creating and deleting business objects and relationships; modifying attribute values of business objects and relationships; and querying objects about attribute values, revisions history, ownership, etc.

The MVC algorithm is implemented as a Java application that takes advantage of the provided API to perform the necessary queries and modifications of the repository data. By basing the implementation on the MDM-generic object types and relationships, the algorithm is designed to be applicable to any kind of specialising business objects, and hence any kind of integrated model.

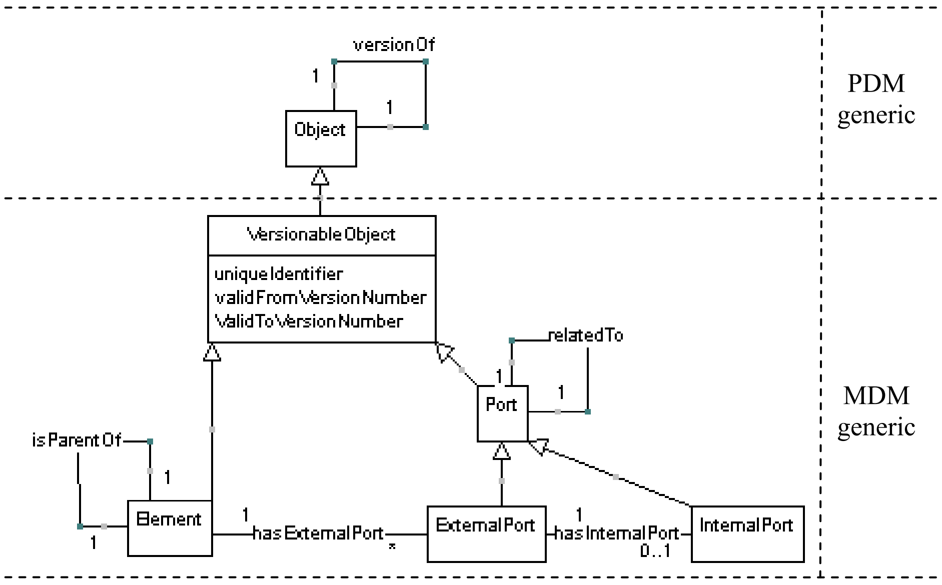


Figure 50. The PDM information model implementing the meta-meta-model and the version control data model.

The details of the check-in/out operations are performed transparently to the user, allowing him/her to interface with the modelling tool's interface and format. The input to the check-in operation is a model represented in an XML document, with the predetermined structure as presented in section D.2.5. Such a document is produced by the adaption layer of the corresponding tool which maps the tool-specific model format to that expected in the platform. The document is hierarchically structured, where the element is represented as an XML-element, with the properties, external ports, direct children and internal relations forming a list of child XML-elements. Each direct child of an element contains in its turn internal child elements of its constituting external ports, elements and internal relations. When performing a check-out operation of a specific element within a model, that element is reconstructed for a given version, together with its children, producing a model of the information in the repository as an XML document. This document is then further transformed by the adaption layer of the corresponding tool to produce a tool-specific format and structure. The output document of the check-out operation for a given element is equivalent to the input to the check-in algorithm for that same element, with the exception that the ordering of the XML-elements may not be reserved. Given the meta-meta-model definition adopted, the ordering of sibling elements is not considered and hence the order difference has no semantic relevance.

Limited testing has so far been performed on the MVC algorithm. However, given its application to two different tools, confidence in the generality of the algorithm is gained.

In addition, performance issues need to be evaluated in the future. The current implementation's performance is not satisfactory. However, no final judgement can be made, since having focused on the functionality, no attempts has been made to optimise the algorithm. In addition, neither measurement nor control of the hardware performance onto which the PDM system resides, were made.

The current implementation aimed to evaluate whether the functionalities offered by a PDM system are sufficient to implement a fine-grained version control system. It remains to see if the expected performance can be provided by PDM, given that such a system is not normally designed to deal with a large number of fine-grained data items. Such an evaluation will provide valuable feedback on to the expected performance of new MDM solutions.

D.2.5. Integrating a Specific Model/Tool

We advocate for the decoupling of the modelling tools from the MDM platform, permitting an open architecture where various tools can be integrated as desired. For this to be possible, the tool-specific format and meta-model, used internally to manage the model data, are mapped to the generic format and meta-model of the corresponding adaption layer. This mapping is performed by the adaption layer. In this way, the management functionalities can operate uniformly using a single format. The data neutral XML format is adopted to interface the adaption layer to the platform. It is these XML files that are communicated between the modelling tool and the platform through the adaption layer. This format, as well as the whole adaption layer, is transparent to the user. The user is expected to interact with the modelling tool only and perform the check-in/out activities based on modelling items.

The process of integrating a specific model of a specific tool is exemplified by integrating the traditional data flow diagram (DFD) model used in the Matlab/Simulink tool.

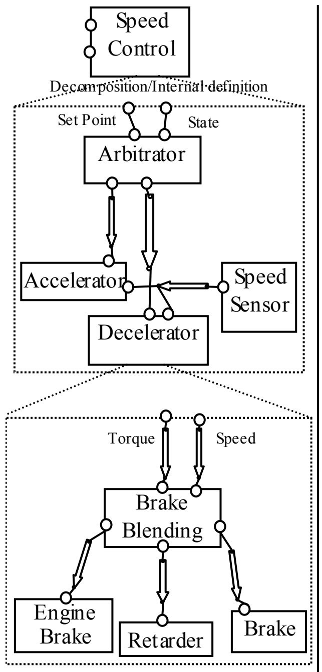
The model of computation is a simple variant of the data-flow model making the definition of the information model quite easy. However, the number and variety of properties of each object that needs to be handled provided an interesting complexity to deal with. There exist more than 100 different types of functions blocks, with at least 100 parameters in each. These properties deal with both the model properties as well as graphical properties. This provided a challenge to define a generic mechanism to handle these properties. In this case, the properties of an object are bundled as an XML node, with a tool-specific structure which the adaption layer can parse and reparse when saving and extracting the properties respectively.

Figure 51a illustrates an example DFD model. In a DFD model, two types of elements are defined: *Functions* and *Communication Links*. A function element designates certain functionality that given a certain input, produces a certain output. A communication link element designates a link that transports data between functions. We here choose to model the links between functions as first-class elements of their own, and not simply as connection relations between functions. Such an approach is also advocated in [14] and [15]. The interface of an element is a set of ports, specifying the data items that are externally accessible to other elements. Furthermore, a DFD is generally hierarchically decomposed, where an element designates an aggregation of other elements. In this sense, the terms 'element' and 'model' can most often be used interchangeably, where an element is internally described (modelled) through its children elements and their relations.

In integrating the Simulink tool, a meta-model of Simulink's DFD model is defined to conform to the meta-meta-model defined in section D.2.1 and implemented according to section D.2.4. This is illustrated as a class diagram in Figure 52. In this figure, the generalisation association between the meta-model and its meta-meta-model items is used to illustrate this relationship from meta-meta-model to meta-model level. While it may not be appropriate to represent the two modelling (meta-meta-model and meta-model) domains using the same class model, this illustration is however closer to the realisation mechanism used in the Matrix system.

In the Matrix platform, the elements Function and Communication Link are defined in the Business Modeller as subtypes of the abstract object type *Element*. Similarly, *SExternalPort* and *SInternalPort* are defined as subtypes of *ExternalPort* and *InternalPort* respectively. It is not necessary to define the relations between the DFD specific object types since such relationships are inherited automatically.

For each of the specialising subtypes, the tool-specific attributes necessary are also defined. In figure 52, a small subset of these properties is illustrated. In reality, given the large number of attributes needed for each object types, it was chosen for this particular tool to bundle the properties into a single attribute (properties). Since the MDM platform does not handle the semantics and internal structure of the attributes, the adaption layer can freely determine the bundling and later unbundling of these properties. Different tools can hence be handled differently. Unless the properties need to be individually defined in the MDM information model, a single XML structure of the properties is sufficient in most cases.



```
<Element type="SimulinkFunction" name="Speed Control"
              uniqueIdentifier="Speed Control">
  <Properties type="struct">
    <Name type="char">Speed Control</Name>
    <Position type="double">260 74 360 116</Position>
    ...
  </Properties>
  <ExternalPort type="SimulinkExternalPort" name=""
                    uniqueIdentifier="inport 1">
    <Properties type="struct">
      <PortNumber type="double">1</PortNumber>
      ...
    </Properties>
  </ExternalPort>
  <ExternalPort type="SimulinkExternalPort" name=""
                    uniqueIdentifier="inport 2">
    ...
  </ExternalPort>
  ...
  <Element type="SimulinkFunction" name="Decelerator"
                uniqueIdentifier="Decelerator">
    ...
    <Element type="SimulinkLine" name=""
                  uniqueIdentifier="inport 1 outport 2">
      <Properties type="struct">
        <Name type="char"/>
        <Description type="char"/>
        ...
      </Properties>
    </Element>
    ...
  </Element>
  ...
</Element>
```

(a) (b)

Figure 51. (a) An example DFD model of parts of a hypothetical truck electrical architecture. (b) The model representation using XML as defined by the DFD adaption layer.

The adaption layer is expected to produce an XML document of any Simulink element (Function or Communication Link) as illustrated in figure 51b. The document reflects the hierarchical structuring of the model as defined in the meta-meta-model. Each element contains within its definition the definition of its child elements (recursively), its contained external ports (each containing the definition of its internal ports), and the internal relations.

In addition, each XML-element is accompanied with a type attribute that reflects the specific MDM object type that the element refers to. This type information is used by the generic MDM functionalities for their configuration and adaptability to any tool-specific meta-model.

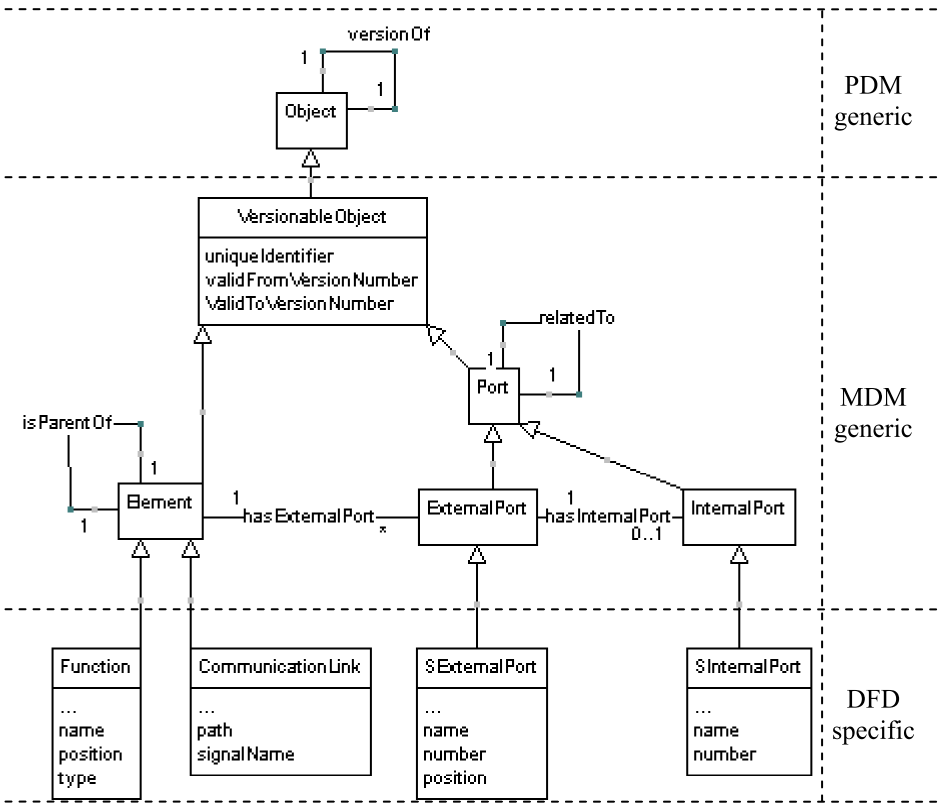


Figure 52. Extending the generic meta-meta model to define a tool-specific meta-model (DFD).

The XML document also defines a *uniqueIdentifier* attribute for each XML-element representing an element or port as required by the MVC algorithm (section D.2.2). The *uniqueIdentifier* is expected to be unique within a limited context (the parent element for an element, the containing element for an external port, and the external port for an internal port). This *uniqueIdentifier* is determined by the adaption layer independent of the MDM platform, and can be determined in different ways depending on the integrated tool. Two main solutions can be envisaged:

- In the case where the tool already allocates an identifier for its entities, this identifier can be adopted. This is generally the case when a database is used by the tool to manage its model content.
- A combination of properties is used to form a *uniqueIdentifier*. This approach is adopted for the Simulink tool for determining the *uniqueIdentifier* for function elements, since the tool allows no two functions within a subsystem to have the same *name* property. For external ports, the combination of the *port type* (inport/outport) with the *port number* attribute are guaranteed to be unique within each function element.

The adaption layer of the Simulink tool uses the tool's available API to query the model when creating the XML document as well as when recreating the model given an XML document. Such an approach is more suitable than performing a transformation between the XML document and the specific file format expected by the tool. This is based on the assumption that over different versions of the tool, the published API is expected to be more stable than the internal file format and structure. In this way, modifications to the internal workings of the tool have less impact on the integration platform.

D.3. Future work

A simple MVC algorithm has been developed, with many future extension possibilities available. A list of potential extensions follows:

- The visualisation of the differences between two versions of a model needs to be developed. Naturally, differences in graphical models would need to be represented graphically.
- As discussed in section D.2.2, the current implementation does not identify the move of one branch of the model tree across the hierarchy. Instead, such a move is considered as a deletion and recreation of each element in the branch. Future implementation may need to handle this feature. It would also be desired to support the version control of changes to a *relatedTo* relation's properties as well as changes to its end ports.
- A limitation ought to be placed so that changes to the interface of the root element checked-out cannot be performed. This is necessary to ensure the consistency of the complete model. Any changes to the interface need to be synchronised with changes to the interfaces and relations of its connected

elements. To perform such a change, the user should check out the parent element of the element of interest.

- It would be interesting to develop a number of version control algorithms based on the same MDM platform. The system can then be configured so that different algorithms can be applied depending on the object types being managed. Such a feature can be advantageous allowing different disciplines to apply different versioning strategies. For example, software development might require the complex version control mechanisms and concurrent development normally provided by SCM systems, while hardware development is satisfied with sequential revision control. There should be no problem providing different solutions in MDM, depending on the kind of data items the functionality operates on. It would however be advantageous to base the different solutions on generic mechanisms for reusability purposes. The different solutions ought to be also based on the same user interface and terminology.

- Concurrent development support needs to be developed. This can be divided into two categories: Inter-model concurrent development and intra-model concurrent development. These two kinds are further detailed in the following subsections.

D.3.1. Intra-model Concurrent Development

In the current implementation, no conflicts occur when users check-out and modify elements from different branches of the hierarchy. However, the synchronisation of concurrent changes between users is necessary when one user attempts to modify an element of the model that is a child of, or the same as, an element being modified by another user. Changes performed would need to be safely merged into a consistent one. There exist two strategies to handle this issue in the future:

- *Lock-unlock* - A user locks the elements to be checked out (and recursively all children elements), and hence prevents other users from simultaneously modifying these elements, avoiding the need for future merging effort.

- *Copy-merge* - Users are allowed to check-out and modify elements along the same hierarchy concurrently. When checking-in changes, MVC merges the changes performed by each user into a consistent model, dealing with conflicts in parallel changes in the process. Analogous to the merging of source code files in conventional SCM systems, model branching/merging mechanisms need to be developed.

The choice of which strategy to adopt is left to the development team. It is interesting to note that traditionally, mechanical engineering have adopted the former strategy while software engineering tools generally allow the latter. By providing different strategies for different kinds of models, the development needs of both disciplines can be satisfied, using variants of the same basic mechanisms in a unified management system.

D.3.2. Inter-model Concurrent Development

As mentioned earlier, MVC operates on a single rooted model hierarchy. However, a system description generally consists of a set different models or views covering its different aspects [16]. This multi-view need is supported in the MDM platform by handling each view as an independent single-root model hierarchy.

But, since these views are not necessarily independent, changes in one view need to be synchronised with changes in another view. For example, a class diagram and the source code further implementing the classes in the diagram share much information that needs to be synchronised in order to maintain consistency in the system specification. For example, changes to attribute names in the former need to be reflected in the attributes in the source code, which could themselves have been changed concurrently.

This kind of synchronisation between different types of views is here termed *inter-model merging/branching* and would need to be developed in the future. It differs from the intra-model merging/branching discussed in the previous subsection since the merging mechanisms depend on inter-view relationships defining the dependencies between the views.

Compared to intra-model merging, merging of seemingly independent models may be a source of confusion for the users that are generally not aware of the dependencies between the system views.

D.4. Related Work

Version control systems targeting models, instead of file objects, are increasingly appearing in the literature ([17], [18] and [19]). In these approaches, an information model of the documents to be handled is assumed, allowing the management of the internal information stored in the documents. While focused on software models, these approaches are helpful since the mechanisms can be applied to any kind of models throughout the development lifecycle.

Version control algorithms can be broadly divided into two categories:

- Algorithms based on the tree-to-tree correction problem [20].
- Algorithms assuming a unique identifier for each node in the hierarchy.

Any algorithm from any of these two groups can be adopted for our purposes. The first group of algorithms does not assume a persistent unique identifier, making it more generally applicable. However, this comes at the cost of reduced performance due to the need to match nodes between the model trees to compare.

For our purposes, the MVC algorithm is implemented based on the second category. The algorithm is applicable as long as a unique identifier can be produced either by the modelling tool or the adaption layer as detailed in section D.2.5. Such a requirement can be satisfied with many modern modelling tools.

D.5. Conclusion

In order to illustrate the MDM solution, an initial implementation of a Model Version Control (MVC) functionality was performed, allowing for the fine-grained version management of two types of models from two different tools – Simulink DFD and Dome Hardware Structure models. MVC permits stakeholders to perform design activities in terms of models, where they can organise, share and modify their models, transparent to the underlying file structure. A simplified version control functionality has been realised. Less focus is however currently placed on advanced capabilities of the version control algorithm such as branching and merging of models.

MVC provides mechanisms that allow a user to save and extract any part of the system model. In a check-in operation, changes to the model since the last check-in operation are saved in the repository. When performing a check-out operation, the specified element is reconstructed for a given version, together with its subparts, forming an XML document of the information in the repository. This document is then further transformed by the adaption layer to create a tool-specific format that can be used by the tool. The details of these operations are performed transparently to the user, allowing him/her to interface with the modelling tool’s interface and format.

D.6. Acknowledgement

This work has been supported by the Swedish Strategic Research Foundation, through the SAVE project. Special thanks go to Martin Törngren and Ola Redell for their reviews of this report.

D.7. References

[1] El-khoury J., Model Data Management – Towards a common solution for PDM/SCM systems, Twelfth International Software Configuration Management Workshop (SCM-12), 2005.

[2] Crnkovic I., Asklund U. and Persson Dahlqvist A., Implementing and integrating product data management and software configuration management, Artech House Publishers, 2003.

[3] Mathworks, Simulink, http://www.mathworks.com/products/simulink/, accessed October 2005.

[4] OMG, Unified Modelling Language (UML) Specification, V1.5, March 2003.

[5] OMG, Meta Object Facility (MOF) Specification, V1.4, April 2002.

[6] Dome, Dome guide, Version 5.2.2, http://www.htc.honeywell.com/dome/index.htm, 1999.

[7] GME, A generic modelling environment, GME 4 User's Manual, Version 4.0, Institute for Software Integrated Systems, Vanderbilt University, 2004.

[8] El-khoury J., Chen D. and Törngren M., A survey of modelling approaches for embedded computer control systems (Version 2.0), Technical report, ISRN/KTH/MMK/R-03/11-SE, TRITA-MMK 2003:36, ISSN 1400-1179, Department of Machine Design, KTH, 2003.

[9] El-khoury J., Towards a multi-view modelling environment for mechatronics systems, Technical report, ISRN/KTH/MMK/R-05/24-SE, TRITA-MMK 2005:24, ISSN 1400-1179, Department of Machine Design, KTH, 2005.

[10] Collins-Sussman B., Fitzpatrick B. W., and Pilato C. M., Version control with Subversion: For Subversion 1.0, 2004.

[11] MatrixOne, Matrix10, http://www.matrixone.com/matrixonesolutions/index.html, accessed October 2005.

[12] Cooling J., Software engineering for real-time systems. Pearson Education Limited, ISBN 0201596202, 2003.

[13] Redell O., El-khoury J. and Törngren M., The AIDA toolset for design and implementation analysis of distributed real-time control systems, Microprocessors and Microsystems, Volume 28, Issue 4, 2004.

[14] Garlan D., Monroe R., and Wile D., ACME: An architecture description interchange language, Proceedings of CASCON'97, 1997.

[15] Clements P., Bachman F., Bass L., Garlan D., Ivers J., Little R., Nord R. and Stafford J., Documenting software architectures: Views and beyond, Addison Wesley, Reading, MA, 2002.

[16] El-khoury J., Redell O. and Törngren M., Integrating views in a multi-view modelling environment, proceedings of the 15th international symposium of the systems engineering conference (INCOSE), 2005.

[17] Ohst D. and Kelter U., A fine-grained version and configuration model in analysis and design, Proceedings of the International Conference on Software Maintenance, 2002.
[18] Nguyen T. N., Munson E.V., Boyland J.T. and Thao C., Flexible fine-grained version control for software documents, 11th Asia-Pacific Software Engineering Conference, 2004.
[19] Chien S. Y., Tsotras V. J., Zaniolo C., Version Management of XML Documents, Third International Workshop WebDB 2000 on The World Wide Web and Databases, 2000.
[20] Tai K. C., The tree-to-tree correction problem, Journal of the ACM, 1979.