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]).

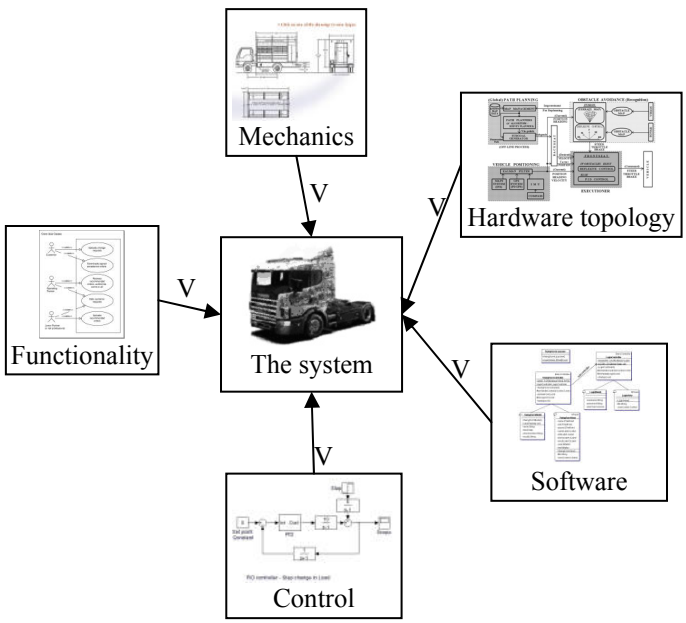

Figure 18. Some of the disciplines and views in system development.

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.

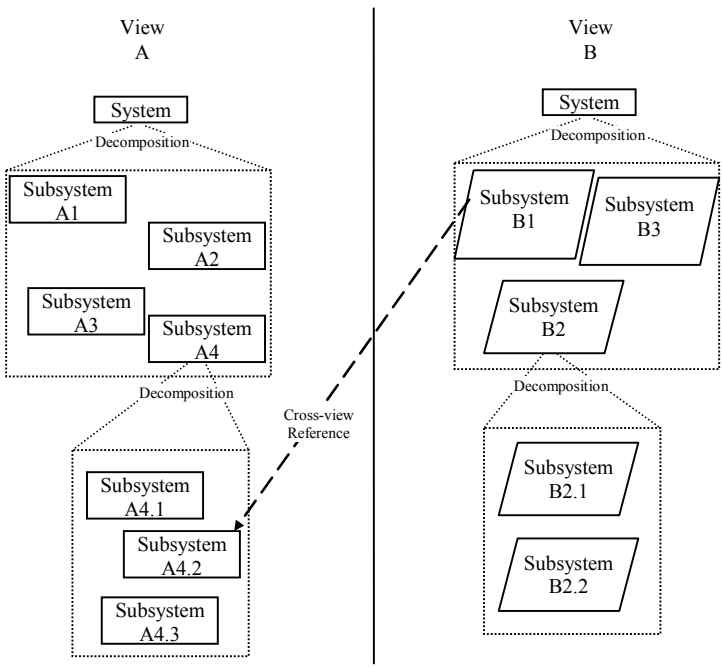

Figure 19. The integration of multi-view and hierarchical decomposition techniques. The broken arrow illustrates a relation between the separated views.

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, Letterspacing 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.

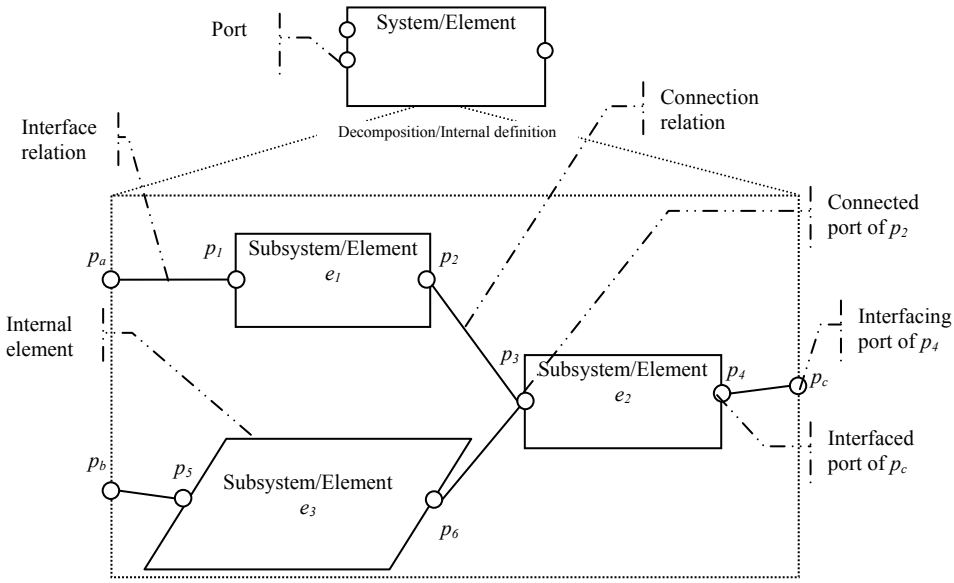

Figure 20. A graphical representation of the general modelling concepts.

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 $E_{dc}(e) = \varnothing$

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

- 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.

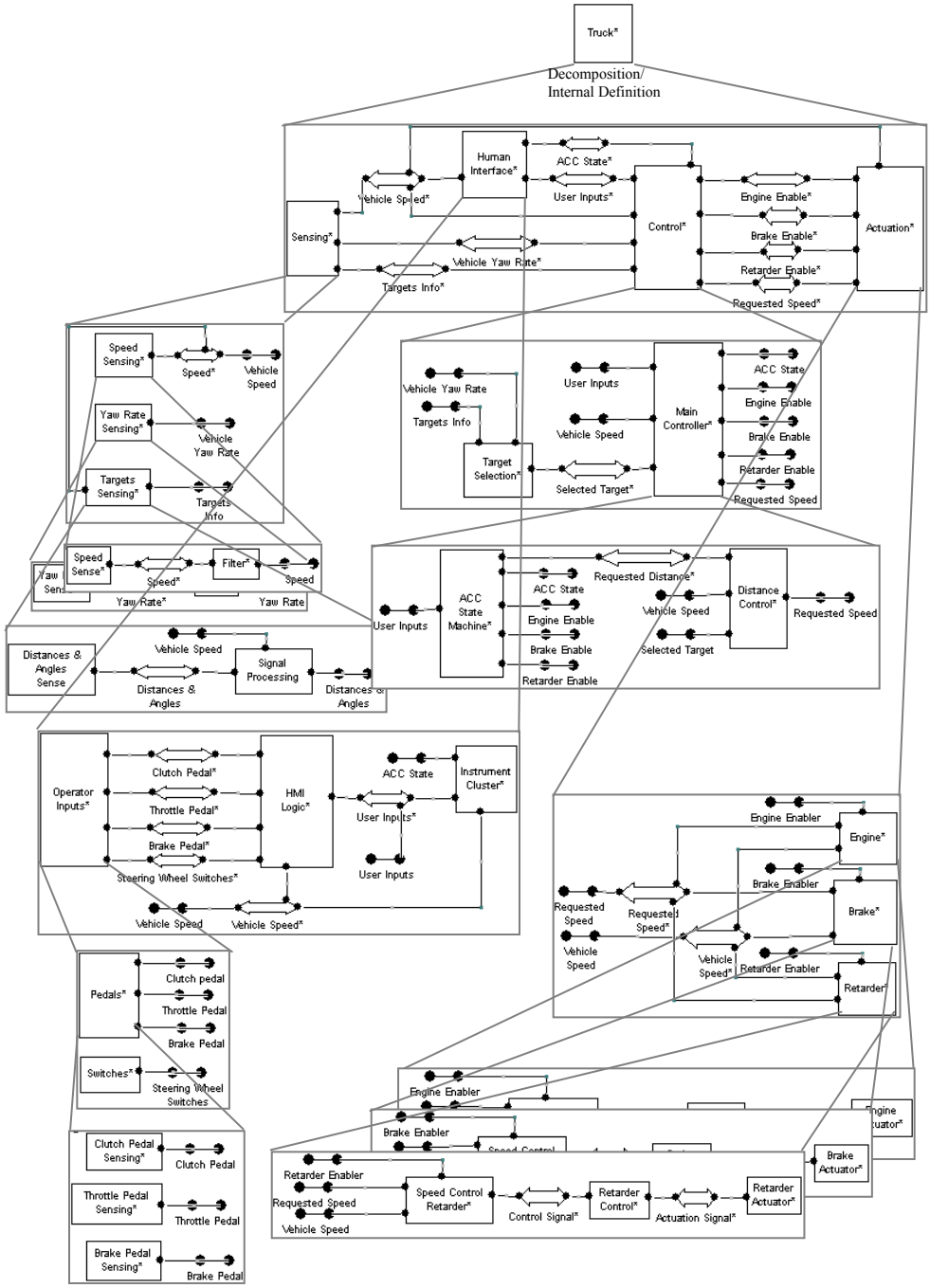

Figure 21. A Function Structure model of the truck ACC functionality.

- 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.

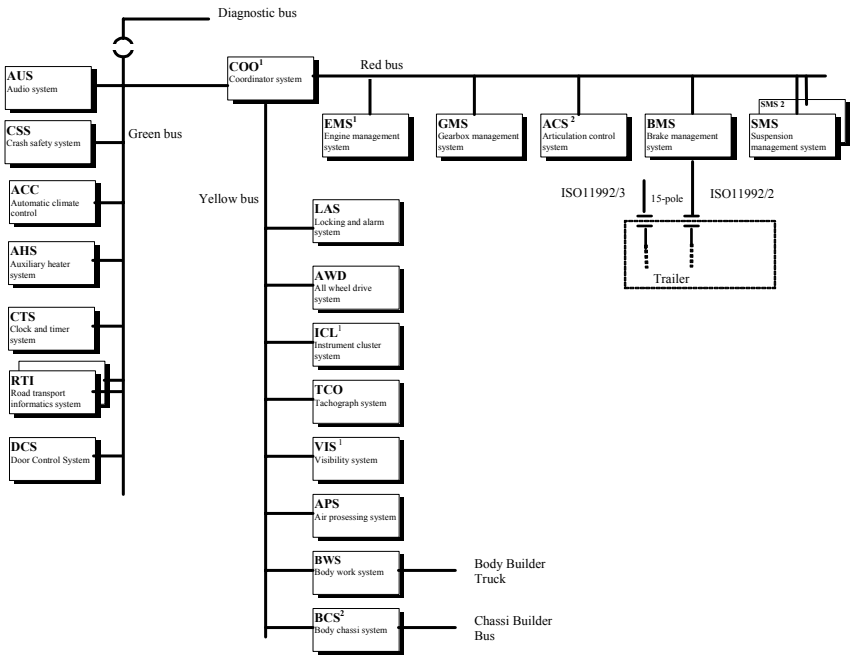

Figure 22. Scania EE architecture

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.

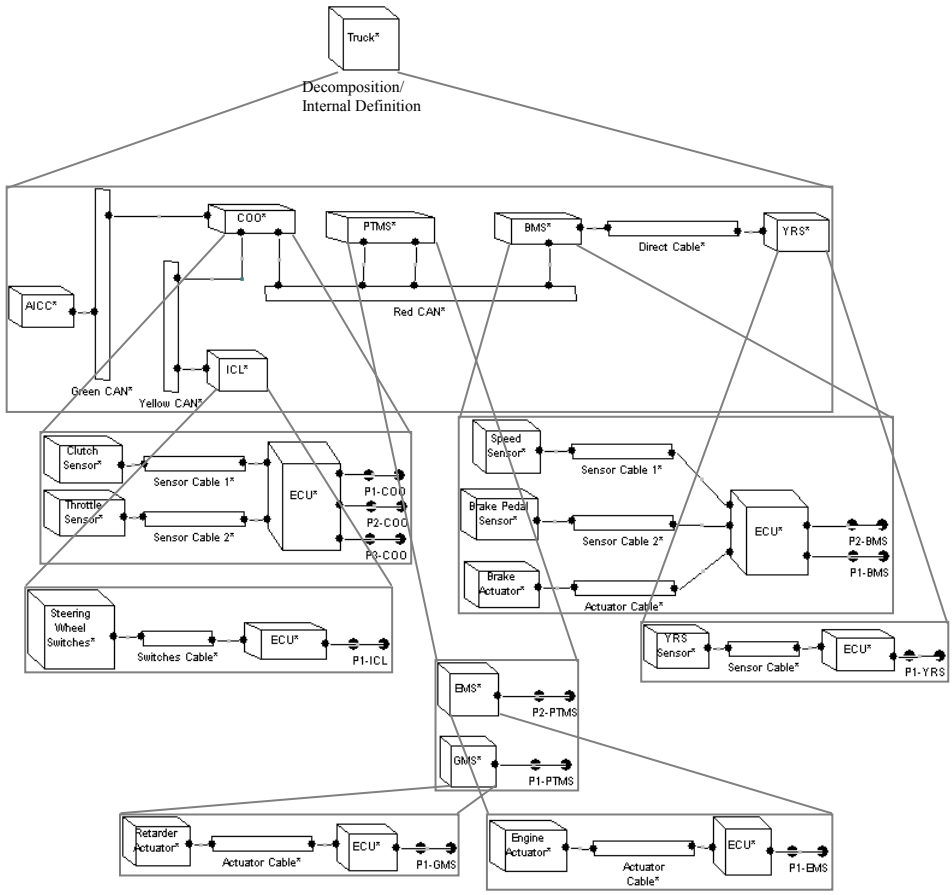

Figure 23. A Hardware Structure model of the truck ACC functionality.

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, $A_d(e_x, V_y)$.

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

- 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.

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

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

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.

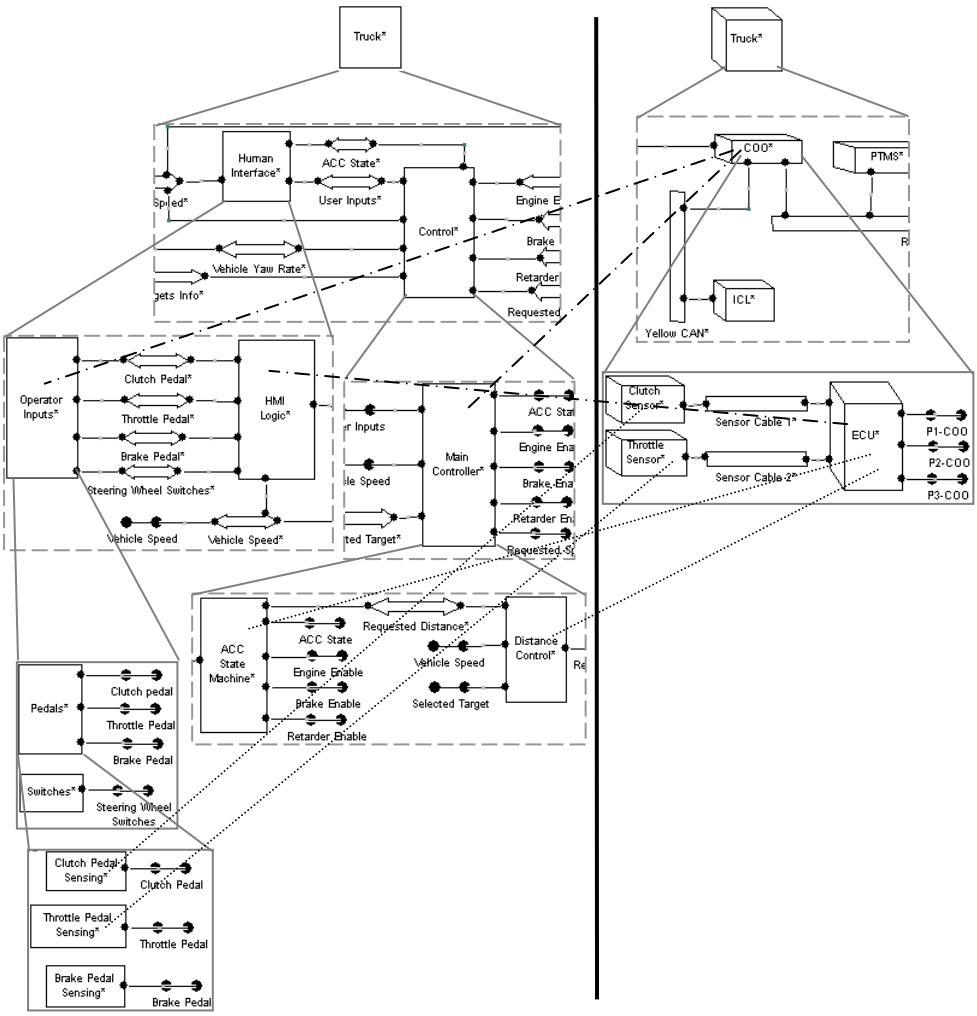

Figure 24. The direct associations of the hardware unit *COO*, as well as some of its child units *ECU*, *Clutch Sensor* and *Throttle Sensor*. The associations from *ECU* to *ACC State Machine* and *Distance Control* specialise that specified to *Main Controller*.

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:

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

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.

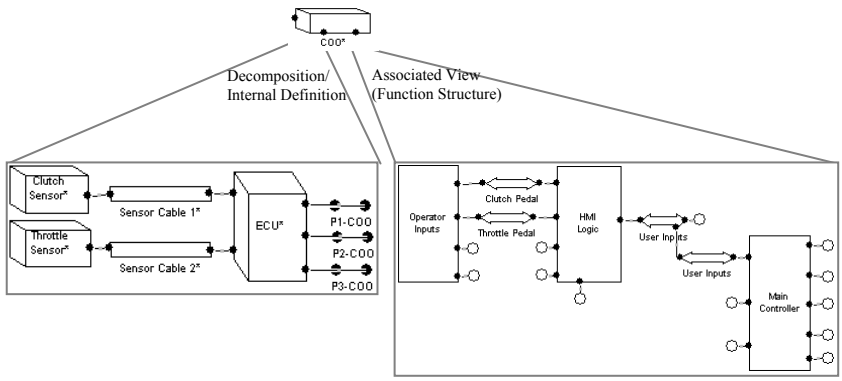

Figure 25. The views of the *COO* hardware unit, consisting of its internal (Hardware Structure) view, as well as the associated Function Structure view.

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:

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

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

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

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:

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

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:

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

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:

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

For example, in figure 24, the associating elements of *Clutch Sensor*, $A_{ai}\left(\textit{Clutch Sensor}, V_{FS}\right)$, 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.

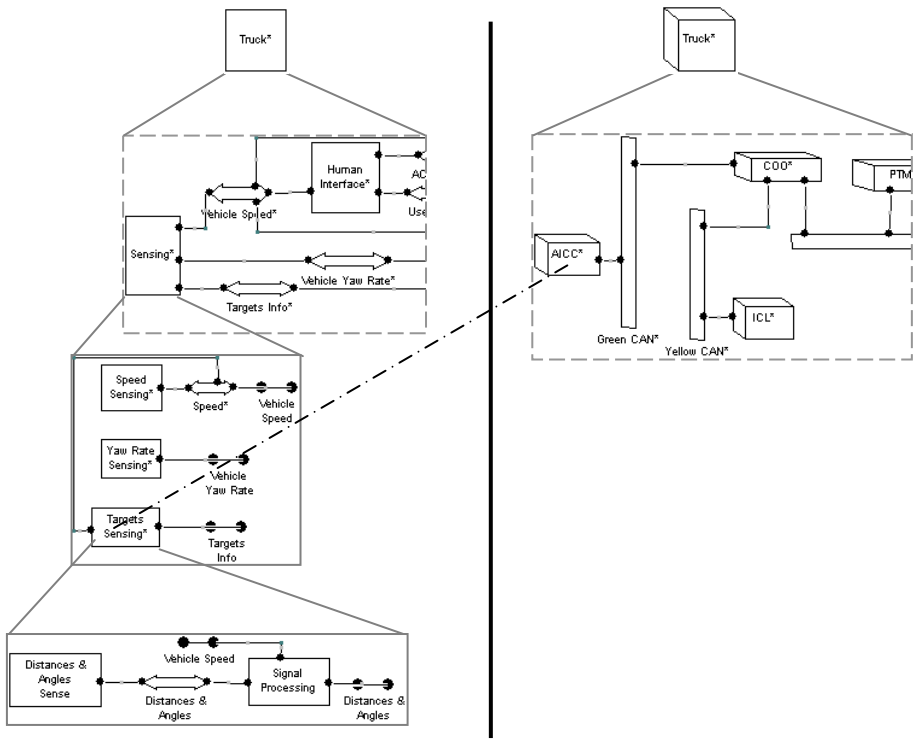

Figure 26. Element association between the *Target Sensing* element and the *AICC* hardware unit.

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.

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, $a_{lv}(\textit{Target Sensing}, V_{HS})$, 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, $\neg a_{lv}(Operator\ Inputs, V_{HS})$, 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:

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

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

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.

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

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.

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

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

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.

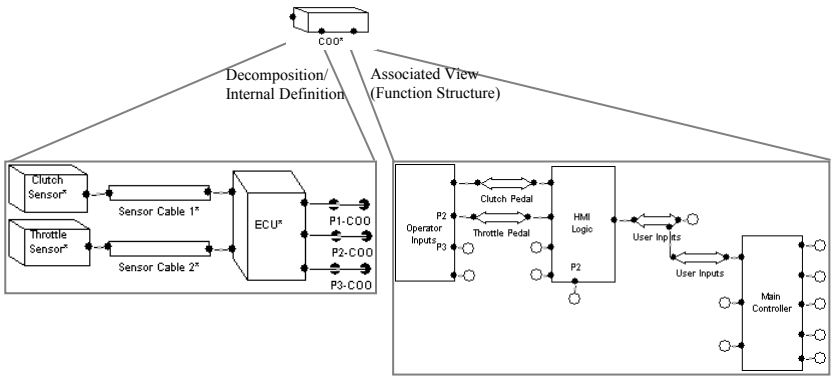

Figure 27. A reproduction of figure 25, highlighting certain port names in the associated view of *COO*, such as p_2 of the *HMI Logic* element.

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

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.

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

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 $p_{2,\text{DistanceControl}}$, 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:

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

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.

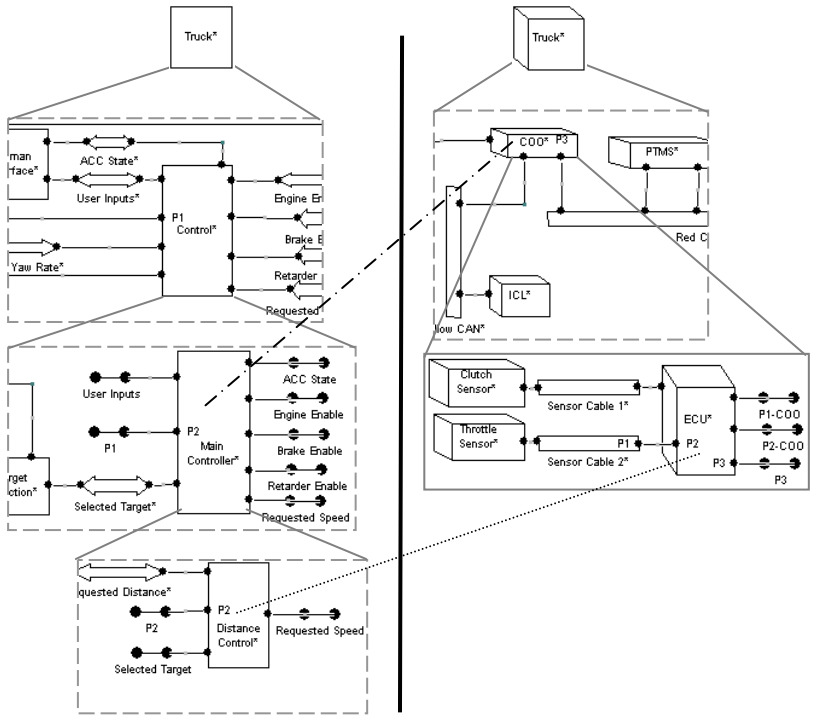

Figure 28. A reproduction of relevant parts from figure 24, focusing on specific direct associations of the hardware unit *COO*, and its child unit *ECU*.

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:

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

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 $Distance\ Control \in A_a(ECU, V_{FS}) \wedge \neg a_{cpa}(p_{2,Distance Control}, ECU)$.

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:

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

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:

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

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 all connected ports associated 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.

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

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:

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

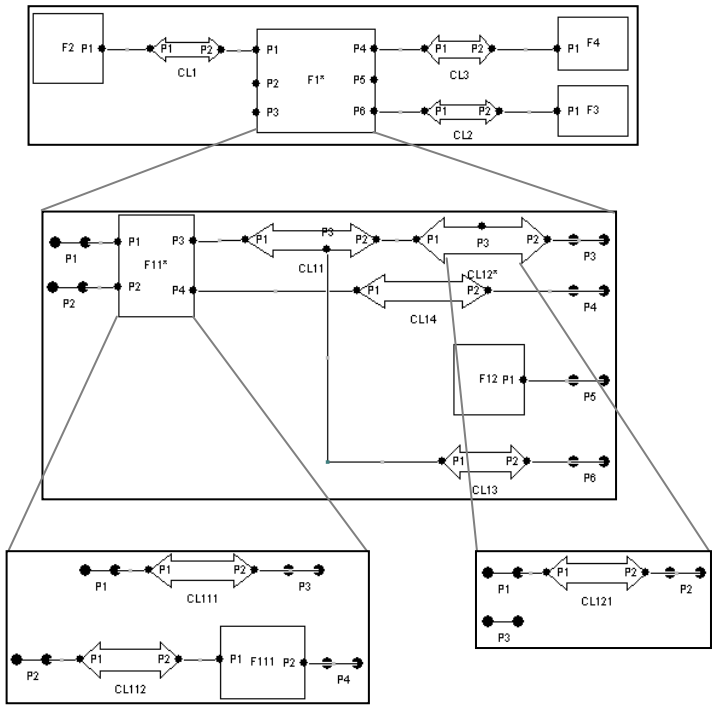

Figure 29. A hypothetical Function Structure model to illustrate internally linked ports.

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, $\left|A_p\left(p,V_{hs}\right)\right|=1$.

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.

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

- 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.

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

- 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.

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

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)*.

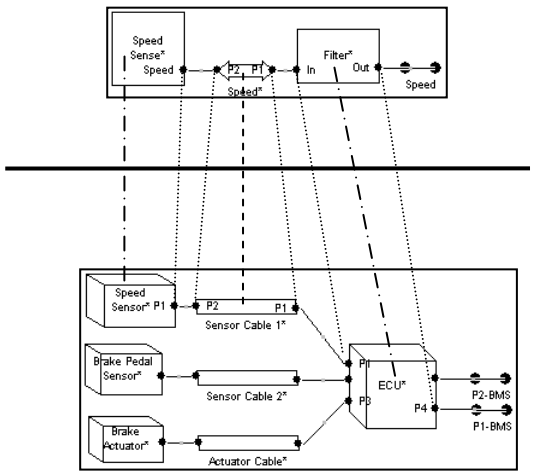

Figure 30. Element and port associations between the child elements of the *Speed Sensing* function element and the *BMS* hardware unit.

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.

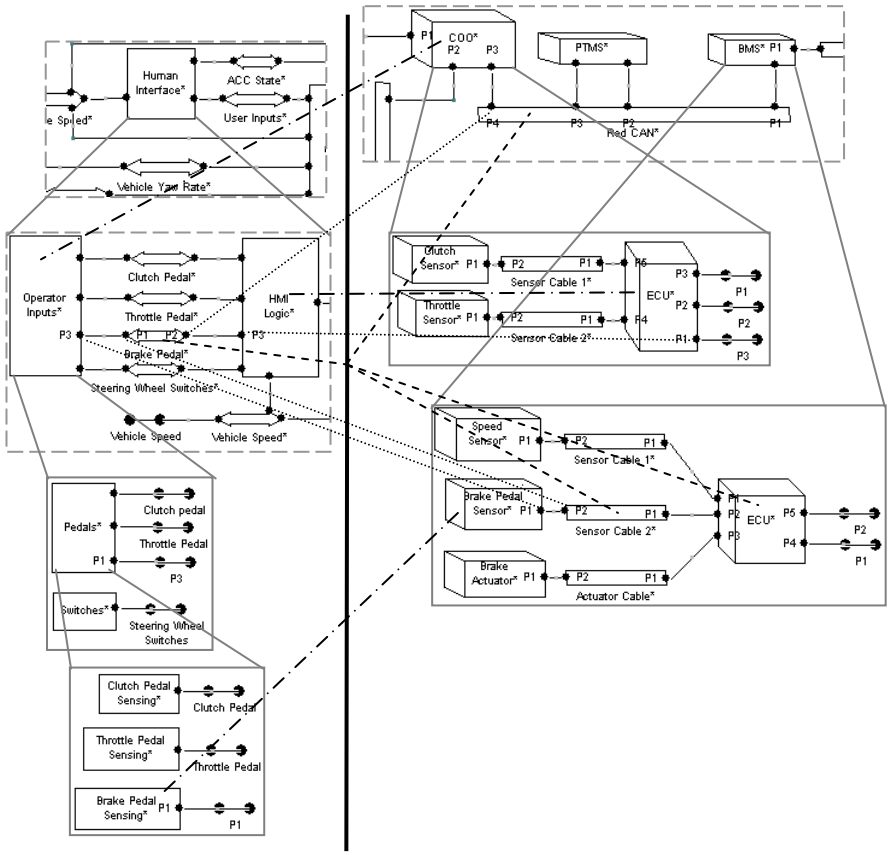

Figure 31. Element and port associations between the child elements of the *Human Interface* function element and the hardware elements onto which it is desired to implement them.

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 $p_{3,HMILogic}$, $p_{2,BrakePedal}$).

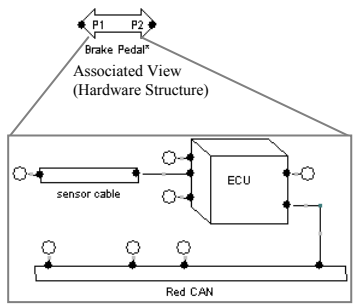

Figure 32. The Hardware Structure associated view of the *Brake Pedal* element.

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

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

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

And

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

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

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:

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

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

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

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

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

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

$$B_i\left(e_{x2},V_y\right)=\Bigg\{a\in\Bigg\{b\in\underset{m\in E_{oc}\left(e_{x2}\right)}{\cup}A_d\left(m,V_y\right):\left(\neg\exists p\in\underset{q\in E_c(x)}{\cup}A_d\left(q,V_y\right):p\in E_p(b)\right)\wedge C\Bigg\}\cup\underset{n\in E_{dc}\left(e_{x2}\right)}{\cup}A_d\left(n,V_y\right):$$
$$\left(\neg\exists m\in\Bigg\{b\in\underset{m\in E_{oc}\left(e_{x2}\right)}{\cup}A_d\left(m,V_y\right):\left(\neg\exists p\in\underset{q\in E_c(x)}{\cup}A_d\left(q,V_y\right):p\in E_p(b)\right)\wedge C\Bigg\}\right.$$
$$\left.\cup\underset{n\in E_{dc}\left(e_{x2}\right)}{\cup}A_d\left(n,V_y\right):m\in E_p(a)\right)\wedge C\Bigg\}$$

Where x is the parent of m that is also the direct child of e_{x2};

and $E_{oc}\left(e_{x2}\right)=E_c\left(e_{x2}\right)-E_{dc}\left(e_{x2}\right)$

Now, let

$$Y\left(e_{x2},V_y\right)=\left\{b\in\underset{m\in E_{oc}\left(e_{x2}\right)}{\cup}A_d\left(m,V_y\right):\left(\exists p\in\underset{q\in E_c(x)}{\cup}A_d\left(q,V_y\right):p\in E_p(b)\right)\wedge C\right\} \quad [9]$$

We have

$$Y\left(e_{x2},V_y\right)'=\underset{n\in E_{oc}\left(e_{z2}\right)}{\cup}A_d\left(n,V_y\right)-Y\left(e_{x2},V_y\right), \quad [10]$$

since $\underset{n\in E_{oc}\left(e_{x2}\right)}{\cup}A_d\left(n,V_y\right)\supset Y\left(e_{x2},V_y\right)$

$B_i\left(e_{x2},V_y\right)$ can be rewritten as:

$$B_i\left(e_{x2},V_y\right)=\Bigg\{a\in\left(Y\left(e_{x2},V_y\right)'\cup\underset{n\in E_{dc}\left(e_{x2}\right)}{\cup}A_d\left(n,V_y\right)\right):$$
$$\left(\neg\exists m\in\left(Y\left(e_{x2},V_y\right)'\cup\underset{n\in E_{dc}\left(e_{x2}\right)}{\cup}A_d\left(n,V_y\right)\right):m\in E_p(a)\right)\wedge C\Bigg\} \quad [11]$$

We first prove:

$$\forall a\in Y\left(e_{x2},V_y\right):\left(\exists m\in Y\left(e_{x2},V_y\right)':m\in E_p(a)\right) \quad [12]$$

Consider such an $a\in Y\left(e_{x2},V_y\right)$:

$$
\begin{aligned}
& a \in Y\left(e_{x2}, V_y\right) \\
\Rightarrow\ & \exists p \in \underset{q \in E_c(x)}{\cup} A_d\left(q, V_y\right) : p \in E_p(a) \quad \left[\text{Definition of } Y\left(e_{x2}, V_y\right)\right] \\
\Rightarrow\ & \exists p \in \underset{q \in E_{oc}\left(e_{x2}\right)}{\cup} A_d\left(q, V_y\right) : p \in E_p(a) \quad \left[E_c(x) \subset E_{oc}\left(e_{x2}\right)\right]
\end{aligned}
\qquad [13]
$$

$$
p \in Y\left(e_{x2}, V_y\right)' \vee p \in Y\left(e_{x2}, V_y\right),
$$

since $p \in \underset{q \in E_{oc}\left(e_{x2}\right)}{\cup} A_d\left(q, V_y\right)$, and $\underset{q \in E_{oc}\left(e_{x2}\right)}{\cup} A_d\left(q, V_y\right) \supset Y\left(e_{x2}, V_y\right)$.

If $p \in Y\left(e_{x2}, V_y\right)'$, then we found a $p \in Y\left(e_{x2}, V_y\right)'$, such that $p \in E_p(a)$, and hence proving expression [12].

If $p \in Y\left(e_{x2}, V_y\right)$, then

$$
\begin{aligned}
& p \in Y\left(e_{x2}, V_y\right) \\
\Rightarrow\ & \exists v \in \underset{q \in E_c(x)}{\cup} A_d\left(q, V_y\right) : v \in E_p(p) \qquad \left[\text{Definition of } Y\left(e_{x2}, V_y\right)\right] \\
\Rightarrow\ & \exists v \in \underset{q \in E_{oc}\left(e_{x2}\right)}{\cup} A_d\left(q, V_y\right) : v \in E_p(p) \qquad \left[E_c(x) \subset E_{oc}\left(e_{x2}\right)\right]
\end{aligned}
$$

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:

$$
\exists v \in \underset{q \in E_{oc}\left(e_{x2}\right)}{\cup} A_d\left(q, V_y\right) : v \in E_p(p), \exists u \in \underset{q \in E_{oc}\left(e_{x2}\right)}{\cup} A_d\left(q, V_y\right) : u \in E_p(v), \quad \ldots
$$

$$
\text{if } v \in Y\left(e_{x2}, V_y\right), u \in Y\left(e_{x2}, V_y\right), \text{ etc.}
$$

(Where $p \in E_p(a), v \in E_p(p),\ u \in E_p(v), \ldots$)

Or prove expression [12] if $v \in Y\left(e_{x2}, V_y\right)', u \in Y\left(e_{x2}, V_y\right)'$, since we would have found a $v/u \in Y\left(e_{x2}, V_y\right)'$, such that $v/u \in E_p(a)$.

(Note that $v \in E_p(a)$, since $v \in E_p(p) \in E_p\left(E_p(a)\right) \in E_p(a)$)

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]:

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

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:

$$
\begin{aligned}
B_i\left(e_{x2},V_y\right) &= \left\{ a \in \left(Y\left(e_{x2},V_y\right) \cup Y\left(e_{x2},V_y\right)' \cup \underset{n\in E_{dc}(e_{x2})}{\cup} A_d\left(n,V_y\right)\right): \right. \\
&\qquad \left. \left(\neg\exists m \in \left(Y\left(e_{x2},V_y\right)' \cup \underset{n\in E_{dc}(e_{x2})}{\cup} A_d\left(n,V_y\right)\right): m \in E_p(a)\right) \wedge C \right\} \\
&= \left\{ a \in \left(\underset{n\in E_{oc}(e_{x2})}{\cup} A_d\left(n,V_y\right) \cup \underset{n\in E_{dc}(e_{x2})}{\cup} A_d\left(n,V_y\right)\right): \right. \\
&\qquad \left. \left(\neg\exists m \in \left(Y\left(e_{x2},V_y\right)' \cup \underset{n\in E_{dc}(e_{x2})}{\cup} A_d\left(n,V_y\right)\right): m \in E_p(a)\right) \wedge C \right\} \\
&= \left\{ a \in \underset{n\in E_c(e_{x2})}{\cup} A_d\left(n,V_y\right): \right. \\
&\qquad \left. \left(\neg\exists m \in \left(Y\left(e_{x2},V_y\right)' \cup \underset{n\in E_{dc}(e_{x2})}{\cup} A_d\left(n,V_y\right)\right): m \in E_p(a)\right) \wedge C \right\}
\end{aligned}
\qquad [14]
$$

We now prove:

$$\forall a \in B_i\left(e_{x2},V_y\right): \left(\neg\exists m \in Y\left(e_{x2},V_y\right): m \in E_p(a)\right) \qquad [15]$$

Assume the inverse of [15]. That is:

$$\exists a \in B_i\left(e_{x2},V_y\right): \left(\exists m \in Y\left(e_{x2},V_y\right): m \in E_p(a)\right) \qquad [16]$$

For this *a*, we know that $\left(\exists m \in Y\left(e_{x2},V_y\right): m \in E_p(a)\right)$

$$
\begin{aligned}
& m \in Y\left(e_{x2},V_y\right) \\
\Rightarrow\ & \exists p \in \underset{q\in E_c(x)}{\cup} A_d\left(q,V_y\right): p \in E_p(m) \quad \left[\text{Definition of } Y\left(e_{x2},V_y\right)\right] \\
\Rightarrow\ & \exists p \in \underset{q\in E_{oc}(e_{x2})}{\cup} A_d\left(q,V_y\right): p \in E_p(m) \quad \left[E_c(x) \subset E_{oc}(e_{x2})\right]
\end{aligned}
\qquad [17]
$$

$p \in Y\left(e_{x2},V_y\right)' \vee p \in Y\left(e_{x2},V_y\right)$,

since $p \in \underset{q\in E_{oc}(e_{x2})}{\cup} A_d\left(q,V_y\right)$, and $\underset{q\in E_{oc}(e_{x2})}{\cup} A_d\left(q,V_y\right) \supset Y\left(e_{x2},V_y\right)$.

But, $p \notin Y\left(e_{x2},V_y\right)'$, Since

$$
\begin{aligned}
& p \in E_p(m) && \left[\text{from } [17]\right] \\
\Rightarrow\ & p \in E_p\left(E_p(a)\right) && \left[\text{m} \in \text{E}_\text{p}(a), \text{from } [16]\right] \\
\Rightarrow\ & p \in E_p(a)
\end{aligned}
$$

and

$$a \in B_i\left(e_{x2}, V_y\right)$$
$$\Rightarrow \left(\neg\exists m \in \left(Y\left(e_{x2}, V_y\right)' \cup \underset{n \in E_{dc}(e_{x2})}{\cup} A_d\left(n, V_y\right)\right) : m \in E_p(a)\right) \wedge C \quad [\text{From } [14]]$$
$$\Rightarrow \left(\neg\exists m \in Y\left(e_{x2}, V_y\right)' : m \in E_p(a)\right)$$

That is, if $\left(\neg\exists m \in Y\left(e_{x2}, V_y\right)' : m \in E_p(a)\right)$ and $p \in E_p(a)$, then $p \notin Y\left(e_{x2}, V_y\right)'$.

Therefore,

$$p \in Y\left(e_{x2}, V_y\right)$$

Now,

$$p \in Y\left(e_{x2}, V_y\right)$$
$$\Rightarrow \exists v \in \underset{q \in E_c(x)}{\cup} A_d\left(q, V_y\right) : v \in E_p(p) \quad \left[\text{Definition of } Y\left(e_{x2}, V_y\right)\right]$$
$$\Rightarrow \exists v \in \underset{q \in E_{oc}(e_{x2})}{\cup} A_d\left(q, V_y\right) : v \in E_p(p) \quad \left[E_c(x) \subset E_{oc}(e_{x2})\right]$$

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:

$$\exists v \in \underset{q \in E_{oc}(e_{x2})}{\cup} A_d\left(q, V_y\right) : v \in E_p(p),\ \exists u \in \underset{q \in E_{oc}(e_{x2})}{\cup} A_d\left(q, V_y\right) : u \in E_p(v),\ \ldots$$

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

$$\exists t \in \underset{q \in E_{oc}(e_{x2})}{\cup} A_d\left(q, V_y\right) : t \in E_p(r)$$

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 $B_i\left(e_{x2}, V_y\right)$ in [14]:

$$B_i\left(e_{x2},V_y\right)=\left\{a\in \underset{n\in E_c\left(e_{x2}\right)}{\cup} A_d\left(n,V_y\right):\right.$$
$$\left.\left(\neg\exists m\in\left(Y\left(e_{x2},V_y\right)'\cup \underset{n\in E_{dc}\left(e_{x2}\right)}{\cup} A_d\left(n,V_y\right)\right):m\in E_p\left(a\right)\right)\wedge C\right\}$$

We know from [15] that for $\forall a\in B_i\left(e_{x2},V_y\right)$, $\left(\neg\exists m\in Y\left(e_{x2},V_y\right):m\in E_p\left(a\right)\right)$ is true.

Therefore, [14] can be rewritten:

$$\begin{aligned}
&B_i\left(e_{x2},V_y\right)\\
&=\left\{a\in \underset{n\in E_c\left(e_{x2}\right)}{\cup} A_d\left(n,V_y\right):\right.\\
&\quad\left(\neg\exists m\in\left(Y\left(e_{x2},V_y\right)'\cup \underset{n\in E_{dc}\left(e_{x2}\right)}{\cup} A_d\left(n,V_y\right)\right):m\in E_p\left(a\right)\right)\wedge C\\
&\quad\left.\wedge\left(\neg\exists m\in Y\left(e_{x2},V_y\right):m\in E_p\left(a\right)\right)\right\}\\
&=\left\{a\in \underset{n\in E_c\left(e_{x2}\right)}{\cup} A_d\left(n,V_y\right):\right.\\
&\quad\left.\left(\neg\exists m\in\left(Y\left(e_{x2},V_y\right)\cup Y\left(e_{x2},V_y\right)'\cup \underset{n\in E_{dc}\left(e_{x2}\right)}{\cup} A_d\left(n,V_y\right)\right):m\in E_p\left(a\right)\right)\wedge C\right\}\\
&=\left\{a\in \underset{n\in E_c\left(e_{x2}\right)}{\cup} A_d\left(n,V_y\right):\right.\\
&\quad\left.\left(\neg\exists m\in\left(\underset{n\in E_{oc}\left(e_{x2}\right)}{\cup} A_d\left(n,V_y\right)\cup \underset{n\in E_{dc}\left(e_{x2}\right)}{\cup} A_d\left(n,V_y\right)\right):m\in E_p\left(a\right)\right)\wedge C\right\}\\
&=\left\{a\in \underset{n\in E_c\left(e_{x2}\right)}{\cup} A_d\left(n,V_y\right):\right.\\
&\quad\left.\left(\neg\exists m\in \underset{n\in E_c\left(e_{x2}\right)}{\cup} A_d\left(n,V_y\right):m\in E_p\left(a\right)\right)\wedge C\right\}\\
&=A_i\left(e_{x2},V_y\right)
\end{aligned}\qquad [18]$$

Now,

$$\begin{aligned} B_a(e_{x2},V_y) &= B_i\left(e_{x2},V_y\right)\cup A_d\left(e_{x2},V_y\right) \quad [\text{from } [4]] \\ &= A_i\left(e_{x2},V_y\right)\cup A_d\left(e_{x2},V_y\right) \quad [\text{from } [18]] \\ &= A_a\left(e_{x2},V_y\right) \end{aligned} \qquad [19]$$

Combining [18] and [19], we get

$$\left(A_i(e_{x2},V_y)=B_i\left(e_{x2},V_y\right)\right)\wedge\left(A_a(e_{x2},V_y)=B_a(e_{x2},V_y)\right)$$

We have now proved that [5] is true for e_{x2}, assuming [5] is true for $\forall e_{x1}\in E_c\left(e_{x2}\right)$ ([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

$$\left(\begin{array}{l} \left(E_p\left(e_y\right)\cap \underset{n\in E_c(e_x)}{\cup} A_d\left(n,V_y\right)=\varnothing\right) \\ \wedge\left(e_y\notin A_d\left(e_x,V_y\right)\right) \\ \wedge\left(E_p\left(e_y\right)\cap A_d\left(e_x,V_y\right)=\varnothing\right) \\ \wedge\left(E_c\left(e_y\right)\cap A_d\left(e_x,V_y\right)=\varnothing\right) \end{array}\right)$$

$$\Leftrightarrow$$

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

We first prove that

$$\left(\begin{array}{l}\left(E_p(e_y)\cap \underset{n\in E_c(e_x)}{\cup} A_d(n,V_y)=\varnothing\right)\\ \wedge\left(e_y\notin A_d(e_x,V_y)\right)\\ \wedge\left(E_p(e_y)\cap A_d(e_x,V_y)=\varnothing\right)\\ \wedge\left(E_c(e_y)\cap A_d(e_x,V_y)=\varnothing\right)\end{array}\right)$$

$$\Leftrightarrow$$

$$\left(\begin{array}{l}\left(E_p(e_y)\cap A_i(e_x,V_y)=\varnothing\right)\\ \wedge\left(e_y\notin A_d(e_x,V_y)\right)\\ \wedge\left(E_p(e_y)\cap A_d(e_x,V_y)=\varnothing\right)\\ \wedge\left(E_c(e_y)\cap A_d(e_x,V_y)=\varnothing\right)\end{array}\right) \qquad [1]$$

Now,

$$\left(E_p(e_y)\cap \underset{n\in E_c(e_x)}{\cup} A_d(n,V_y)=\varnothing\right)$$

$$\Rightarrow \neg\exists x\in E_p(e_y): x\in \underset{n\in E_c(e_x)}{\cup} A_d(n,V_y)$$

$$\Rightarrow \neg\exists x\in E_p(e_y): x\in A_i(e_x,V_y) \qquad \left[\text{Since } A_i(e_x,V_y)\subset \underset{n\in E_c(e_x)}{\cup} A_d(n,V_y)\right]$$

$$\Rightarrow \left(E_p(e_y)\cap A_i(e_x,V_y)=\varnothing\right)$$

Hence,

$$\left(\begin{array}{l}\left(E_p(e_y)\cap \underset{n\in E_c(e_x)}{\cup} A_d(n,V_y)=\varnothing\right)\\ \wedge\left(e_y\notin A_d(e_x,V_y)\right)\\ \wedge\left(E_p(e_y)\cap A_d(e_x,V_y)=\varnothing\right)\\ \wedge\left(E_c(e_y)\cap A_d(e_x,V_y)=\varnothing\right)\end{array}\right)$$

$$\Rightarrow$$

$$\left(\begin{array}{l}\left(E_p(e_y)\cap A_i(e_x,V_y)=\varnothing\right)\\ \wedge\left(e_y\notin A_d(e_x,V_y)\right)\\ \wedge\left(E_p(e_y)\cap A_d(e_x,V_y)=\varnothing\right)\\ \wedge\left(E_c(e_y)\cap A_d(e_x,V_y)=\varnothing\right)\end{array}\right) \qquad [2]$$

Considering the RHS of (1),

$$\left(\begin{array}{l}\left(E_p\left(e_y\right)\cap A_i\left(e_x,V_y\right)=\varnothing\right)\\ \wedge\left(e_y\notin A_d\left(e_x,V_y\right)\right)\\ \wedge\left(E_p\left(e_y\right)\cap A_d\left(e_x,V_y\right)=\varnothing\right)\\ \wedge\left(E_c\left(e_y\right)\cap A_d\left(e_x,V_y\right)=\varnothing\right)\end{array}\right)$$

$$\Rightarrow$$

$$\left(\begin{array}{l}\left(\neg\exists x\in A_i\left(e_x,V_y\right):x\in E_p\left(e_y\right)\right)\\ \wedge\left(e_y\notin A_d\left(e_x,V_y\right)\right)\\ \wedge\left(E_p\left(e_y\right)\cap A_d\left(e_x,V_y\right)=\varnothing\right)\\ \wedge\left(E_c\left(e_y\right)\cap A_d\left(e_x,V_y\right)=\varnothing\right)\end{array}\right) \quad [3]$$

Now, assume that

$$\exists a\in\underset{n\in E_c\left(e_x\right)}{\cup}A_d\left(n,V_y\right):a\in E_p\left(e_y\right) \quad [4]$$

$a\in A_i\left(e_x,V_y\right)\vee a\in A_i\left(e_x,V_y\right)'$, since $a\in\underset{n\in E_c\left(e_x\right)}{\cup}A_d\left(n,V_y\right)$,
and $\underset{n\in E_c\left(e_x\right)}{\cup}A_d\left(n,V_y\right)\supset A_i\left(e_x,V_y\right)$.

But $a\notin A_i\left(e_x,V_y\right)$, since from [3], we have $\neg\exists x\in A_i\left(e_x,V_y\right):x\in E_p\left(e_y\right)$, and from [4] we have $a\in E_p\left(e_y\right)$.

Therefore,

$$a\in A_i\left(e_x,V_y\right)'$$

From the definition of $A_i\left(e_x,V_y\right)$ (section B.5.1.1), we get that for $a\in A_i\left(e_x,V_y\right)'$

$$\neg\left(\begin{array}{l}\left(\neg\exists m\in\underset{n\in E_c\left(e_x\right)}{\cup}A_d\left(n,V_y\right):m\in E_p\left(a\right)\right)\\ \wedge\left(\neg\exists m\in A_d\left(e_x,V_y\right):m\in E_p\left(a\right)\vee m=a\right)\end{array}\right)$$

That is,

$$\left(\begin{array}{l}\left(\exists m\in\underset{n\in E_c\left(e_x\right)}{\cup}A_d\left(n,V_y\right):m\in E_p\left(a\right)\right)\\ \vee\left(\exists m\in A_d\left(e_x,V_y\right):m\in E_p\left(a\right)\wedge m\neq a\right)\end{array}\right) \quad [5]$$

Considering the second predicate of [5]:

$$\begin{aligned} &\exists m \in A_d\left(e_x, V_y\right): m \in E_p(a) \wedge m \neq a \\ \Rightarrow\ &\exists m \in A_d\left(e_x, V_y\right): m \in E_p\left(E_p\left(e_y\right)\right) \wedge m \neq a \qquad \left[\text{from } [4], \text{ we have } a \in E_p\left(e_y\right)\right] \\ \Rightarrow\ &\exists m \in A_d\left(e_x, V_y\right): m \in E_p\left(e_y\right) \wedge m \neq a \end{aligned}$$

But, this is false since it is given in [3] that $E_p\left(e_y\right) \bigcap A_d\left(e_x, V_y\right) = \varnothing$

Hence [5] becomes:

$$\left(\exists m \in \underset{n \in E_c\left(e_x\right)}{\cup} A_d\left(n, V_y\right): m \in E_p(a)\right)$$

This is similar to assumption [4], where m replaces a with, $a \in E_p\left(e_y\right)$ and $m \in E_p(a)$.

So, by repeating the argument above, the following statements can be deduced:

$$\left(\exists p \in \underset{n \in E_c\left(e_x\right)}{\cup} A_d\left(n, V_y\right): p \in E_p(m)\right), \left(\exists q \in \underset{n \in E_c\left(e_x\right)}{\cup} A_d\left(n, V_y\right): q \in E_p(p)\right), \ldots,$$

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

$$\left(\exists v \in \underset{n \in E_c\left(e_x\right)}{\cup} A_d\left(n, V_y\right): v \in E_p(r)\right)$$

But, since no such v can exist since r is the root of the tree, we conclude that assumption [4] is false.

That is

$$\neg \exists a \in \underset{n \in E_c\left(e_x\right)}{\cup} A_d\left(n, V_y\right): a \in E_p\left(e_y\right)$$

or

$$E_p\left(e_y\right) \bigcap \underset{n \in E_c\left(e_x\right)}{\cup} A_d\left(n, V_y\right) = \varnothing \qquad [6]$$

Now, [6] is proven true based on [3], and we hence can write:

$$\left(\begin{array}{l}\left(\neg\exists x \in A_i\left(e_x, V_y\right): x \in E_p\left(e_y\right)\right) \\ \wedge \left(e_y \notin A_d\left(e_x, V_y\right)\right) \\ \wedge \left(E_p\left(e_y\right) \cap A_d\left(e_x, V_y\right) = \varnothing\right) \\ \wedge \left(E_c\left(e_y\right) \cap A_d\left(e_x, V_y\right) = \varnothing\right)\end{array}\right)$$

$$\Rightarrow$$

$$\left(E_p\left(e_y\right) \cap \bigcup_{n \in E_c\left(e_x\right)} A_d\left(n, V_y\right) = \varnothing\right)$$

$\therefore$

$$\left(\begin{array}{l}\left(E_p\left(e_y\right) \cap A_i\left(e_x, V_y\right) = \varnothing\right) \\ \wedge \left(e_y \notin A_d\left(e_x, V_y\right)\right) \\ \wedge \left(E_p\left(e_y\right) \cap A_d\left(e_x, V_y\right) = \varnothing\right) \\ \wedge \left(E_c\left(e_y\right) \cap A_d\left(e_x, V_y\right) = \varnothing\right)\end{array}\right)$$

$$\Rightarrow$$

$$\left(E_p\left(e_y\right) \cap \bigcup_{n \in E_c\left(e_x\right)} A_d\left(n, V_y\right) = \varnothing\right)$$

$\therefore$

$$\left(\begin{array}{l}\left(E_p\left(e_y\right) \cap A_i\left(e_x, V_y\right) = \varnothing\right) \\ \wedge \left(e_y \notin A_d\left(e_x, V_y\right)\right) \\ \wedge \left(E_p\left(e_y\right) \cap A_d\left(e_x, V_y\right) = \varnothing\right) \\ \wedge \left(E_c\left(e_y\right) \cap A_d\left(e_x, V_y\right) = \varnothing\right)\end{array}\right)$$

$$\Rightarrow$$

$$\left(\begin{array}{l}\left(E_p\left(e_y\right) \cap \bigcup_{n \in E_c\left(e_x\right)} A_d\left(n, V_y\right) = \varnothing\right) \\ \wedge \left(e_y \notin A_d\left(e_x, V_y\right)\right) \\ \wedge \left(E_p\left(e_y\right) \cap A_d\left(e_x, V_y\right) = \varnothing\right) \\ \wedge \left(E_c\left(e_y\right) \cap A_d\left(e_x, V_y\right) = \varnothing\right)\end{array}\right) \quad [7]$$

Combining [2] and [7], we get:

$$\begin{pmatrix} \left(E_p\left(e_y\right)\bigcap A_i\left(e_x,V_y\right)=\varnothing\right) \\ \wedge\left(e_y\notin A_d\left(e_x,V_y\right)\right) \\ \wedge\left(E_p\left(e_y\right)\bigcap A_d\left(e_x,V_y\right)=\varnothing\right) \\ \wedge\left(E_c\left(e_y\right)\bigcap A_d\left(e_x,V_y\right)=\varnothing\right) \end{pmatrix}$$

$$\Leftrightarrow$$

$$\begin{pmatrix} \left(E_p\left(e_y\right)\bigcap \underset{n\in E_c\left(e_x\right)}{\cup} A_d\left(n,V_y\right)=\varnothing\right) \\ \wedge\left(e_y\notin A_d\left(e_x,V_y\right)\right) \\ \wedge\left(E_p\left(e_y\right)\bigcap A_d\left(e_x,V_y\right)=\varnothing\right) \\ \wedge\left(E_c\left(e_y\right)\bigcap A_d\left(e_x,V_y\right)=\varnothing\right) \end{pmatrix}$$

Hence, we prove [1].

Now,

$$\left(\left(E_p\left(e_y\right)\bigcap A_i\left(e_x,V_y\right)=\varnothing\right)\wedge\left(E_p\left(e_y\right)\bigcap A_d\left(e_x,V_y\right)=\varnothing\right)\right)$$

$$\Leftrightarrow$$

$$\left(E_p\left(e_y\right)\bigcap A_a\left(e_x,V_y\right)=\varnothing\right)$$

Since $A_a\left(e_x,V_y\right)=A_i\left(e_x,V_y\right)\bigcup A_d\left(e_x,V_y\right)$

Hence,

$$\begin{pmatrix} \left(E_p\left(e_y\right)\bigcap \underset{n\in E_c\left(e_x\right)}{\cup} A_d\left(n,V_y\right)=\varnothing\right) \\ \wedge\left(e_y\notin A_d\left(e_x,V_y\right)\right) \\ \wedge\left(E_p\left(e_y\right)\bigcap A_d\left(e_x,V_y\right)=\varnothing\right) \\ \wedge\left(E_c\left(e_y\right)\bigcap A_d\left(e_x,V_y\right)=\varnothing\right) \end{pmatrix}$$

$$\Leftrightarrow$$

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

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,

$$\begin{aligned}
&\left(A_d\left(e_x, V_y\right) \neq \varnothing\right) \vee \left(\exists n \in E_c\left(e_x\right) : A_d\left(n, V_y\right) \neq \varnothing\right) \\
&\equiv \neg\left(\left(A_d\left(e_x, V_y\right) = \varnothing\right) \wedge \neg\left(\exists n \in E_c\left(e_x\right) : A_d\left(n, V_y\right) \neq \varnothing\right)\right) \\
&\equiv \neg\left(\left(A_d\left(e_x, V_y\right) = \varnothing\right) \wedge \left(\forall n \in E_c\left(e_x\right) : A_d\left(n, V_y\right) = \varnothing\right)\right) \\
&\equiv \neg\left(\left(A_d\left(e_x, V_y\right) = \varnothing\right) \wedge \left(\bigcup_{n \in E_c\left(e_x\right)} A_d\left(n, V_y\right) = \varnothing\right)\right)
\end{aligned} \quad [1]$$

We now prove that

$$\left(\left(A_d\left(e_x, V_y\right) = \varnothing\right) \wedge \left(\bigcup_{n \in E_c\left(e_x\right)} A_d\left(n, V_y\right) = \varnothing\right)\right) \Leftrightarrow \left(\left(A_d\left(e_x, V_y\right) = \varnothing\right) \wedge \left(A_i\left(e_x, V_y\right) = \varnothing\right)\right) \quad [2]$$

First, given the definition of A_i in section B.5.1.1:

$$\begin{aligned}
&\bigcup_{n \in E_c\left(e_x\right)} A_d\left(n, V_y\right) = \varnothing \\
&\Rightarrow A_i(e_x, V_y) = \left\{a \in \varnothing : \left(\neg\exists m \in \bigcup_{n \in E_c\left(e_x\right)} A_d\left(n, V_y\right) : m \in E_p(a)\right)\right. \\
&\qquad\qquad \left. \wedge \left(\neg\exists m \in A_d\left(e_x, V_y\right) : m \in E_p(a) \vee m = a\right)\right\} \\
&\Rightarrow A_i(e_x, V_y) = \varnothing
\end{aligned}$$

Hence,

$$\left(\left(A_d\left(e_x, V_y\right) = \varnothing\right) \wedge \left(\bigcup_{n \in E_c\left(e_x\right)} A_d\left(n, V_y\right) = \varnothing\right)\right) \Rightarrow \left(\left(A_d\left(e_x, V_y\right) = \varnothing\right) \wedge \left(A_i\left(e_x, V_y\right) = \varnothing\right)\right) \quad [3]$$

Second,

$$
\begin{aligned}
&\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\left(A_i\left(e_x,V_y\right)=\varnothing\right)\right)\\
\Rightarrow&\left\{a\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):\left(\neg\exists m\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):m\in E_p\left(a\right)\right)\right.\\
&\qquad\left.\wedge\left(\neg\exists m\in\varnothing:m\in E_p\left(a\right)\vee m=a\right)\right\}\\
&=\varnothing\\
\Rightarrow&\left\{a\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):\left(\neg\exists m\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):m\in E_p\left(a\right)\right)\right\}=\varnothing\\
\Rightarrow&\left(\forall a\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):\left(\exists m\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):m\in E_p\left(a\right)\right)\right)\\
&\vee\left(\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right)=\varnothing\right)
\end{aligned}
\qquad [4]
$$

Now, assume that

$$
\forall a\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):\left(\exists m\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):m\in E_p\left(a\right)\right) \qquad [5]
$$

And consider an a such that $a\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right)$.

$$
\begin{aligned}
&a\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right)\\
\Rightarrow&\ \exists m\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):m\in E_p\left(a\right)\qquad[\text{from }[5]]\\
\Rightarrow&\ \exists p\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):p\in E_p\left(m\right)\qquad\left[\text{from }[5],\text{ since }m\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right)\right]\\
\Rightarrow&\ \exists q\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):q\in E_p\left(p\right)\qquad\left[\text{from }[5],\text{ since }p\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right)\right]\\
&\ldots
\end{aligned}
$$

Note that $m\in E_p\left(a\right)$, $p\in E_p\left(m\right)$, $q\in E_p\left(p\right)$, 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

$$
\exists v\in\bigcup_{n\in E_c(e_x)}A_d\left(n,V_y\right):v\in E_p\left(r\right)
$$

But since no such *v* can exist, we can conclude that [5] is not valid.

Therefore, [4] becomes

$$\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\left(A_i\left(e_x,V_y\right)=\varnothing\right)\right)$$
$$\Rightarrow\left(\underset{n\in E_c(e_x)}{\cup}A_d\left(n,V_y\right)=\varnothing\right)$$

Hence,

$$\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\left(A_i\left(e_x,V_y\right)=\varnothing\right)\right)\Rightarrow\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\left(\underset{n\in E_c(e_x)}{\cup}A_d\left(n,V_y\right)=\varnothing\right)\right) \quad [6]$$

Combining [3] and [6], we get

$$\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\left(\underset{n\in E_c(e_x)}{\cup}A_d\left(n,V_y\right)=\varnothing\right)\right)$$
$$\Leftrightarrow$$
$$\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\left(A_i\left(e_x,V_y\right)=\varnothing\right)\right)$$

and thus proving [2].

Combining [1] and [2], we get:

$$\left(A_d\left(e_x,V_y\right)\neq\varnothing\right)\vee\left(\exists n\in E_c\left(e_x\right):A_d\left(n,V_y\right)\neq\varnothing\right)$$
$$\equiv\neg\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\left(\underset{n\in E_c(e_x)}{\cup}A_d\left(n,V_y\right)=\varnothing\right)\right)\quad[\text{from }[1]]$$
$$\equiv\neg\left(\left(A_d\left(e_x,V_y\right)=\varnothing\right)\wedge\left(A_i\left(e_x,V_y\right)=\varnothing\right)\right)\quad[\text{from }[2]]$$
$$\equiv\left(A_d\left(e_x,V_y\right)\cup A_i\left(e_x,V_y\right)\right)\neq\varnothing$$
$$\equiv A_a\left(e_x,V_y\right)\neq\varnothing$$

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,

$$\left(\forall n\in E_c\left(e_x\right):\neg a_{xv}\left(n,V_y\right)\right)\Rightarrow\left(\forall n\in E_{dc}\left(e_x\right):\neg a_{xv}\left(n,V_y\right)\right)$$
$$[\text{Since } E_{dc}\left(e_x\right)\subset E_c\left(e_x\right)] \quad [1]$$

Second,

$$\left(\forall n \in E_{dc}(e_x) : \neg a_{xv}(n, V_y)\right)$$
$$\Rightarrow \left(\forall n \in E_{dc}(e_x) : \left(\forall m \in E_c(n) : \neg a_{xv}(m, V_y)\right)\right)$$
$$\left[\text{From section 1.6}, \neg a_{xv}(e_{x1}) \Rightarrow \forall n \in E_c(e_{x1}) : \neg a_{xv}(n, V_y)\right]$$
$$\Rightarrow \left(\forall n \in E_{oc}(e_x) : \neg a_{xv}(n, V_y)\right)$$
$$\Rightarrow \left(\forall n \in E_{oc}(e_x) : \neg a_{xv}(n, V_y)\right) \wedge \left(\forall n \in E_{dc}(e_x) : \neg a_{xv}(n, V_y)\right)$$
$$\left[\begin{array}{l}(A \Rightarrow B) \equiv (A \Rightarrow (B \wedge A)) \\ \text{and} \quad E_{oc}(e_x) = E_c(e_x) - E_{dc}(e_x)\end{array}\right]$$
$$\Rightarrow \left(\forall n \in E_c(e_x) : \neg a_{xv}(n, V_y)\right)$$

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

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