COMPUTERIZED INTERFACE CONTROL DOCUMENTS

Keyvan Rahmani

Department of Mechanical Engineering

McGill University

Montreal, Quebec

April 2012

A dissertation submitted to McGill University in partial fulfillment of the requirements for the degree of Doctor of Philosophy

ABSTRACT

Interfaces between subsystems in collaborative product development projects are presently defined by interface control documents. These documents are created after agreements are made between designers on how to design subsystem interfaces. The designers must commit to the definitions given by these documents in order to ensure their subsystems remain compatible as the design process continues. An important consequence of using interface control documents during product development processes is that they make the interface control task manual and document based, which can impede design processes.

This thesis presents the ingredients of a computer aided methodology for defining and controlling subsystem interfaces. In this thesis, interfaces are port to port interactions between subsystems. Ports are specified attributively. The two main sources of attributes that specify a port are its form and function. Two ports are called compatible if the values of their attributes satisfy the compatibility constraints that have been defined for them. An interface can be established between two ports if they are compatible.

Compatibility constraints are defined by different subsystem designers. They are transformed into interface control rules in order to be used to control the status of interfaces during a product development project. The rules altogether constitute a knowledge base that can be used for compatibility checking. The left hand sides of the rules in the knowledge base correspond to the compatibility constraints that have been defined for ports. The right hand sides of the rules specify detection messages that alert designers about violations of compatibility constraints as well as their exact location. The interface control knowledge base is the computer manageable representation of interface control documents.

The thesis also proposes a mechanism that ensures interface definitions are created consistently by different teams. An ontology is used for this purpose. The

ontology explicitly provides a vocabulary from which port attributes can be chosen. By committing to the ontology, interface definitions are defined consistently.

Finally, the thesis proposes a software architecture that can operate on the ontology and the interface control knowledge base to control the consistency and compatibility of interfaces during collaboration. A piece of software that corresponds to the proposed architecture is implemented to demonstrate its key functionalities. The functionalities are illustrated by means of two examples that show how interface information in a design project can be captured by the software and how the consistency and compatibility of interfaces can be checked.

RÉSUMÉ

Les interfaces entre les sous-systèmes dans les projets collaboratifs de développement de produits sont actuellement définies par les documents de contrôle d'interface. Ces documents sont créés après que des accords soient conclus entre les concepteurs sur la façon de concevoir les interfaces des sous-systèmes. Les concepteurs doivent s'engager à respecter les définitions données par ces documents afin de s'assurer que leurs sous-systèmes restent compatibles courant le processus de conception. Une conséquence importante de l'utilisation de documents de contrôle d'interface au cours des processus de développement de produits, c'est qu'ils rendent la tâche de contrôle d'interface manuelle et liée à des documents, ce qui peut entraver les processus de conception.

Cette thèse présente les ingrédients d'une méthodologie assistée par ordinateur pour définir et contrôler les interfaces des sous-systèmes. Dans cette thèse, des interfaces sont des interactions port à port entre des sous-systèmes. Les ports sont précisés au moyen de leurs attributs. Les deux principales sources d'attributs qui spécifient un port sont sa forme et sa fonction. Deux ports sont appelés compatibles si les valeurs de leurs attributs satisfont des contraintes de compatibilité qui ont été définies pour eux. Une interface peut être établie entre deux ports s'ils sont compatibles.

Des contraintes de compatibilité sont définies par les concepteurs qui développent des sous-systèmes différents. Elles sont transformées en règles de contrôle d'interface afin d'être utilisées pour contrôler l'état des interfaces au cours d'un projet de développement de produits. Tout en tout, les règles constituent une base de connaissances qui peut être utilisée pour la vérification de compatibilité. Les côtés gauches des règles dans la base de connaissances correspondent aux contraintes de compatibilité qui ont été définies pour les ports. Les côtés droits des règles spécifient les messages de détection qui alertent les concepteurs sur des contraintes de compatibilité violées avec leurs emplacements exacts. La base de

connaissances de contrôle d'interface est la représentation de documents de contrôle d'interface qui est gérable par ordinateur.

La thèse propose également un mécanisme qui assure que les définitions des interfaces sont créées de manière cohérente. Une ontologie est utilisée à cette fin. L'ontologie fournit explicitement un vocabulaire à partir de laquelle les attributs des ports peuvent être choisis. En s'engageant à l'ontologie, la définition des interfaces est définie de façon cohérente.

Enfin, la thèse propose une architecture logicielle qui peut fonctionner sur l'ontologie et la base de connaissances de contrôle d'interface pour contrôler la cohérence et la compatibilité des interfaces au cours d’une collaboration. Un logiciel qui correspond à l'architecture proposée est implémenté afin de démontrer ses fonctionnalités clés. Les fonctionnalités sont illustrées au moyen de deux exemples qui montrent comment les informations d'interface dans un projet de conception peuvent être capturées par le logiciel et comment la cohérence et la compatibilité des interfaces peuvent être vérifiées.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deep gratitude to my advisor, Professor Vince Thomson. I would like to sincerely thank him for all his guidance, patience, and encouragement during the development of this thesis and my graduate studies.

Next, I am indebted to the people who provided the input information for the application examples discussed in this thesis. I would like to thank Mr. Keith Hoege of CAE Inc. (Canadian Aviation Electronics) for providing the input information for the flight simulator example. I also thank the students associated with the 2011 CAMAQ project at École Polytechnique de Montréal, particularly Mr. Boris Toche, for providing the input information for the pylon example.

I am also indebted to all my colleagues in the PLM group at McGill for useful advice during the course of this project. My special thanks go to Dr. Onur Hisarciklilar, a post-doctoral fellow in our lab.

Finally, I would like to thank professor Alain Desrochers of the University of Sherbrooke for reading and giving useful comments about the thesis that helped me to improve its content.

TABLE OF CONTENTS

LIST OF FIGURES

1 INTRODUCTION

1.1 Problem statement

An *interface* refers to any logical or physical relationship required to join the boundary of a system to another system, or the boundary of a system to its environment. Here, the word *system* refers to a set of interoperable elements compatible with each other in *form*, *fit* and *function* to achieve a specific outcome (Wasson, 2006).

Interface management has been defined as "the management of communication, coordination and responsibility across a common boundary between two organizations, phases, or physical entities which are interdependent" (Wideman, 2002). The main idea behind interface management is to improve communication, and therefore, to prevent inconsistencies and errors in information exchange between organizations.

The above definition of interface management is broad; it includes interfaces between all entities that can exist in a complex system: humans, machines, procedures, missions, policies, environments, media, etc. This thesis is only concerned with systems that are the result of engineering work. These are commonly referred to as *products*[1]. Interface management plays an important role in the development of complex products. To develop such products, fast and consistent information sharing among design teams is critical to prevent design errors (Blyler, 2004).

Product interface management consists of a variety of activities from identifying interfaces during conceptual design to ensuring interoperability of subsystems during detailed design. Nowadays, to develop a complex product, the design task is often distributed among collaborating teams that are located at different places.

[1] In this thesis, the words system and product are used interchangeably.

In such a setup, design teams must first agree on the specification of interfaces before they proceed with their own part of the design. These agreed specifications are written down in *interface control*[2] *documents* (ICD). As the design process continues, any subsystem that is being developed must adhere to these specifications so that it can be integrated into the rest of the system.

An important consequence of using ICDs during product development is that they make projects document driven. This has some major drawbacks since documents differ substantially from one organization to another. There are standards for the *organization* of ICDs in certain domains, such as aircraft stores (SAE, 2004), but there is no standard for the content of ICDs. The common practice of using natural languages, homemade drawings, graphs, etc. to create ICDs by diverse organizations leads to ambiguities when design information is shared among organizations. Moreover, using documents makes design processes manual and time consuming. Finally, in the lack of a standard language for interface representation, ICDs have not been included in the set of data that can be managed by computer aided design (CAD) and product data management (PDM) systems.

The above shortcomings of a document based interface control methodology can be eliminated by using a computer readable language to define ICDs. If one can represent any interface in a structured and computer readable form, the interface control process and ICDs can be automatically managed by computers; hence, a significant amount of error and misunderstanding due to poor interface design can be prevented in product development processes. This thesis presents a language and software architecture to *define* and *control* interfaces in a product development process.

[2] Interface control as used in this thesis means the management of interface information.

1.2 Why study interfaces in product development?

There are two main reasons to study product interfaces. One is to analyze product *architecture* and the other is to ensure *compatibility* of subsystems during product development. The first one deals with interfaces at an abstract level, whereas the second one deals with them at a concrete level.

Much research on product architectural analysis has been sparked since some key studies were done in the early 1990s that revealed the impact of product architecture on product development activities. The influential paper published by Karl Ulrich (1995) is one of the most notable ones. He argued that product architecture plays a key role in many performance aspects of a manufacturing firm. He identified product interfaces as one of the defining elements of product architecture.

Ever since, studies that use mathematical approaches to analyze product architecture and its effects on product development processes have received a lot of attention. The design structure matrix (DSM) has been one of the most popular tools in conducting such studies (Browning, 2001; Danilovic and Browning, 2007; MacCormack et al., 2006; Pimmler and Eppinger, 1994; Sosa et al., 2003).

DSM shows dependencies among components. The main idea behind using DSM in architectural analysis is to see if components can be regrouped into modules such that each module contains only highly dependent components that are otherwise less dependent on the rest of the system. This is done by using *clustering* algorithms[3]. A clustered DSM has a reduced number of dependencies among modules. Dividing a product into modules in this way makes the whole product development process more efficient. Modules set the tune for the organization of product development teams and the design process.

[3] DSM is also used to find an optimal sequence of design tasks. The algorithms used for this purpose are called partitioning algorithms.

Evidently, dependencies among components can be represented by graphs or matrices. As such, studies of product architecture have received much attention in academia, whereas ICDs have not, because ICDs are information intensive. This thesis is intended to bring some academic insights into the issue of interface control.

1.3 Interface definition and control

ICDs have been traditionally used in highly complex projects, such as the lunar module in the Apollo program (Blair-Smith, 2010; Eyles, 2004). In non-complex products, much of the effort is put into the design of individual parts, and ICDs are not usually needed. In complex products, ICDs cannot be ignored because such products are composed of *subsystems* rather than just simple parts. These products often have a distributed architecture. For example, a control system that sits in one location can be used to derive a hydro-mechanical system that sits in another location. Such products are also increasingly designed and built by geographically distributed teams. Therefore, they require consistent definition and careful control of ICDs.

Perhaps, one of the most well known approaches to create ICDs is described in NASA's training manual for interface definition and control (Lalli et al., 1997). According to this manual, the main steps of interface control are as follows: identify interfaces, categorize interfaces, document interfaces, and analyze for interface compatibility. These steps are briefly explained here.

1.3.1 Identification of interfaces

The first task of any interface management process is to identify where interfaces are going to occur. Interfaces are identified during conceptual design when subsystem boundaries are drawn within a system. Drawing system boundaries can be done based on an organization's experience, or by using reasoning methods such as functional decomposition. The result of functional decomposition is a description of a product in terms of its primitive functions. This description is

called the functional architecture of the product. By grouping highly relevant functions together, a functional area is obtained that can be implemented by a physical subsystem. This process is called *synthesis*, and the resulting description of a product in terms of its subsystems is called its physical architecture (Figure 1.1). The physical architecture of a system should be established before interface control documents are created.

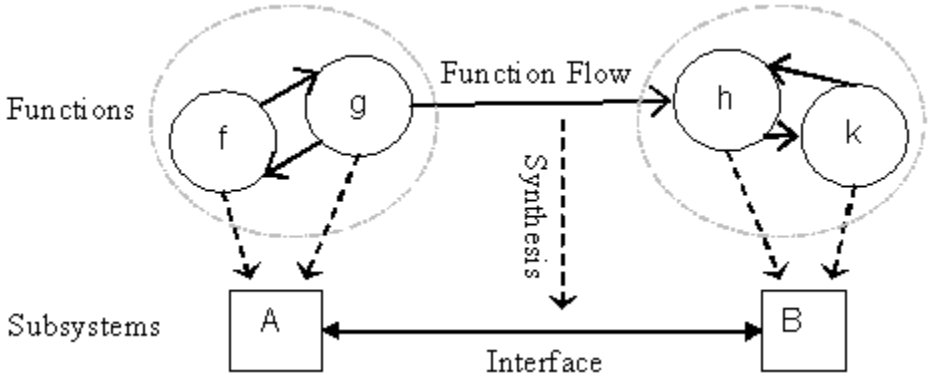


Figure 1.1: Relationship between functional and physical architecture.

1.3.2 Categorization of interfaces

The most widely used interface categorization comes from design theory. From the design theoretic point of view, interactions among systems are divided into the following four classes: spatial, energy, signal, and material (Pahl and Beitz, 2005; Pimmler and Eppinger, 1994). These interactions are briefly defined as:

Spatial: identifies adjacency or orientation between two entities.
Energy: identifies energy transfer between two entities.
Signal : identifies signal exchange between two entities.
Material: identifies material exchange between two entities.

Categorization of interfaces can help to better organize ICDs. Interfaces of the same type can be compiled into separate ICDs, for example, mechanical ICD, electrical ICD, etc.

1.3.3 Documentation of interfaces

Documentation of interfaces can be started right after they are identified. Interfaces are usually identified during conceptual design and ICDs are created after this phase. ICDs are then used during the detailed design and the subsequent stages. They are used as reference documents during the subsequent stages, but they may also evolve. This is quite expected because all the details of interfaces may not be known right after conceptual design. As the design process continues, more details can be added to ICDs. The main objectives of ICDs can be summarized as follows (Lalli et al., 1997):

1. control the design of subsystem interfaces by preventing any changes to a subsystem's boundary that would affect *compatibly* of its interfaces with other subsystems,
2. *communicate* design decisions that affect a subsystem's boundary to all collaborating parties,
3. identify missing interface data (*voids*) and control the submission of these data,
4. identify the subsystems that are associated with an interface.

The most important criterion in writing these documents is to avoid unnecessary details. The documentation should only highlight how the compatibility of interfaces can be demonstrated during subsystem design. Interface documentation should not assume any information about the internal structure of any subsystem. The focus should remain only at subsystem boundaries. This is why the metaphor *interface plane* is used as illustrated in Figure 1.2.

The documents created during the interface documentation phase can take a variety of forms and names. In most cases, the term ICD has been used in place of all such documents. These documents provide either a one sided view or a two sided view of the interfaces between subsystems (Figure 1.2).

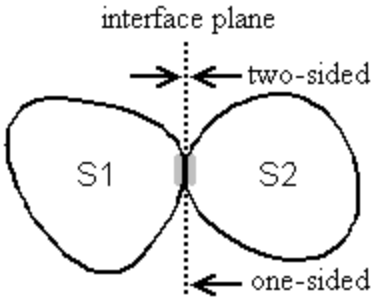


Figure 1.2: Imaginary interface plane between boundaries of subsystems *S*1 and *S*2.

One sided view: These documents are created by subsystem developers. They describe the requirements for a subsystem's boundary. Satisfying these requirements is necessary to ensure proper functionality of the subsystem.

Two sided view: These documents are created when two subsystems from separate organizations must adhere to a common interface. They detail the parameters of interfaces between interacting elements of two subsystems.

The lifetime of a specific ICD depends on whether interface definitions change during a project. Some interface definitions remain fixed throughout the lifecycle of the program while others change frequently. In a document based interface management system, it is worth treating each group separately to reduce the document management workload.

1.3.4 Analysis of interface compatibility

Interface information compiled into ICDs must be analyzed for compatibility. This means that any proposed changes to the interface definitions must be evaluated to ensure the interfaces are still compatible. This task is essential in any interface control process. Each time a change to interface definitions occurs, the compatibility analysis must be able to demonstrate the *completeness* and *correctness* of interface information. Demonstrating information completeness means revealing whether any part of interface data is missing. Demonstrating

correctness means providing a record that shows that interfaces have been *examined* to have the right form, fit, and function for the interacting subsystems. In document based interface control, these tasks are very time consuming, particularly if interfaces change frequently. Each time a change in a definition is requested, the designers themselves need to elicit the pertinent data from ICDs and conduct the analysis.

1.4 Thesis scope

Although the term *product* in this thesis is used interchangeably with the term *system*, it should be noted that in general the term product has a much more limited scope than the term system. The term product usually refers to an engineered good. Cars, airplanes, bridges, and refrigerators are products, whereas banks, universities and fire departments are systems. By *subsystem*, we refer to any interoperable group of components in a product.

The domain of all possible interfaces that can occur among all possible products is a vast universe. It is too ambitious to think that a single tool can be designed to solve all the problems that can arise in such a huge domain. The purpose of this section is to draw a clearer boundary around the set of problems this thesis intends to solve. As mentioned before, this thesis is concerned with the issue of interface control during product development processes. Some of the assumptions used in the thesis have already been put forward in the previous sections; they are restated again here more precisely and followed by some additional assumptions.

1.4.1 Internal and external interfaces

We need to distinguish between internal and external interfaces of a system. *Internal* interfaces are the ones that occur among the components within a system's boundary. *External* interfaces are the ones that occur between a system's boundary and its environment, such as people, physical environment, etc. It might have already been noticed from frequent use of the word *subsystem* that this thesis focuses only on internal system interfaces.

1.4.2 Abstraction level

It has also been set forth that this thesis is not intended for product architectural analysis or conceptual design; it is intended for interface control. Here, the product architecture/concept is considered an input to the process of interface control. This is true in a *top-down* design process.

Functions that describe a product's architecture have highly abstract representations that only describe the intent of product's subsystems. This means that the actual implementation of functions is disregarded in the product's architecture. ICDs on the other hand are concerned with the concrete implementations of the functions that occur at subsystem boundaries; hence, it is the content of interfaces that receives attention in ICDs, not the intent. The following paragraphs are intended to clarify what is meant by the content of interfaces.

Product functions are usually represented in the form of a verb-object pair of symbols (f , o) where f is chosen from a set of appropriate action verbs and o is chosen from a set of objects (Pahl and Beitz, 2005). For example, consider the function "light the lamp". Here the verb is 'to light', and the object is 'the lamp'. This function can be decomposed into two simpler subfunctions: "connect to the electrical outlet" and "press the electrical switch". These functions only show the intent of product's components/subsystems.

In some function representations, objects of the verb-object pair are regarded as flows (Hirtz et al., 2002). In such representations the scope of action verbs is more limited, but the approach becomes more informative. The flows are usually assumed to belong to one of the following classes: energy, material, and signal. Note that this is in accordance with the common functional categorization of interfaces introduced in §1.3.2.

The verb-flow representation of functions can be used to describe the functionality of interfaces, given that the difference between an interface function and that of a subsystem is properly understood. The former is a two sided phenomenon that describes what happens at an interface, but the latter is a one sided phenomenon that describes what a subsystem does. The full specification of flows in a verb-flow representation of an interface function is a part of its content.

It can be seen later in chapter 3 that the full specification of the content of an interface requires the specification of the flows between subsystem boundaries as well as the relationships between the forms of these boundaries. In such a representation, forms and flows are defined by their physical properties.

In this thesis, we are more interested in the functionality of interfaces than their forms. This is the characteristic of a system with a distributed architecture, for example, a flight simulator. A flight simulator is a composition of hardware, software and mechanical subsystems that are placed in different locations and interact with each other through a collection of wires, data buses, tubes, etc. Purely mechanical interfaces, such as the geometric interfaces in a mechanical structure or a mechanism can be represented by mechanical assemblies. However, we do consider the information pertinent to the forms of interfaces since they cannot be completely ignored in interface definitions even in a distributed system.

1.4.3 Physical interfaces

A final note to establish the scope of the thesis is to state that this thesis only focuses on *physical interfaces*. Physical interfaces are generally defined as the interfaces that are physically quantifiable (DAU, 2001; USAF, 2005; FAA, 2006; Lalli et al., 1997). Physical interfaces themselves constitute a vast domain, and they are the ones that are highly relevant to engineering design. The interface definition language that is proposed in this thesis applies to all physical interfaces. The language is not limited to a single engineering domain. An example of a

system whose interfaces are describable by the language is a mechatronic system where there are spatial, electrical and signal interactions among its subsystems.

1.5 Thesis objectives

As explained earlier, interface definition and control processes are currently document based; so, they are highly manual. There is a lack of a language and software architecture for definition and control of interfaces. This problem has received no attention by academia in spite of the large amount of attention that has been given to component dependencies in product architectural analysis. This thesis is intended to reduce this gap.

This thesis proposes a language, software architecture and a methodology to define and manage the information contained in ICDs. The interface definition language can be used to define all sorts of physical interfaces regardless of their domain (mechanical, electrical, etc.). It is shown later in Chapter 3 that a physical interface can be represented by a set of binary constraints over peripheral attributes of two interacting subsystems. The software architecture proposed in this thesis is intended for collaborative product development. The interface definition and control methodology proposed in this thesis provides a framework to:

1. consistently *define* interfaces,
2. identify *missing* interface information,
3. automatically check the *compatibility* of interfaces,
4. *communicate* violations of interface compatibilities to all stakeholders and precisely track the violation.

To fulfill the above objectives, the thesis proposes the following ingredients to create the interface control knowledge that can be managed by computers and be included in CAD/PDM systems:

- An ontology that explicitly specifies the semantics of interfaces. The ontology provides a common vocabulary for interface definitions; so, it helps to overcome the lack of commonality in interface terminologies and improves information sharing among organizations. When all collaborating agents commit to a formal ontology, the interface definitions become meaningful and consistent.
- An interface control knowledge base that semantically specifies interfaces. The interface control knowledge is represented as a collection of interface control rules. A checking mechanism is proposed that operates on the interface control knowledge base, and reports compatibility violations of interfaces. A violation message identifies the erroneous subsystem, the place where the violation happened on the subsystem's boundary, and the name of the subsystem property that has been violated.
- Software architecture that shows how interface control software operates with the ontology and the interface knowledge base to communicate the status of interfaces to designers.

1.6 Thesis organization

The rest of this thesis is organized as follows:

Chapter 2 presents a background for the thesis and surveys some of the areas that are relevant to the subject of this thesis. In this chapter, first, different interface conceptualization practices are discussed. The interface conceptualizations are discussed from the perspective of the engineering domain in which they are used. Next, the chapter discusses the current technology and the languages that are used for semantic information definition. Finally, the use of ontologies as knowledge sharing artifacts, particularly in engineering design, is described.

Chapter 3 presents a formal model for interface definition, which can be used as a language to create computer readable ICDs. This chapter also discusses some of the main sources of information that should be considered when creating

computerized ICDs. The three sources of information that are considered in this thesis are form, function and fit of an interface. The chapter also presents an ontology that can be used for consistent and semantic interface definition.

Chapter 4 presents the control mechanism that can be used by a piece of software that operates on semantic interface definitions. This is necessary because when an interface is violated, the precise meaning of the violation and the exact place where it occurred must be understood. The chapter concludes with a possible architecture for the interface control software.

Chapter 5 presents a prototype implementation of the interface control software. The interface control software is supposed to operate on the interface data stored in shared repositories that are accessible via a network. The prototype interface control software is implemented in Java.

Chapter 6 gives a demonstration of the interface control software that can check the consistency and correctness of computerized ICDs. The chapter includes examples that demonstrate the functionality of the implemented software.

Chapter 7 presents a summary of the contributions of this thesis and outlines a number of research issues that can be studied in the future.

2 BACKGROUND AND RELATED WORK

This chapter first gives an account of the prevalent interface conceptualizations in different engineering domains. Interface conceptualizations are discussed from three different engineering perspectives: software, electronic, and mechanical engineering. This chapter then discusses and compares some of the existing IT languages that are capable of making documents computer readable. Finally, since making a document computer readable is not enough to unequivocally share it among agents, a brief discussion of the current information sharing technology, i.e., ontologies, is also given. Ontologies resolve lexical and semantic problems in information sharing; so, they can be used for the same purpose in ICDs.

2.1 Conceptualization of interfaces

The concept of interface has different meanings in different engineering domains. This section explores different interface conceptualization methodologies in software, hardware and mechanical engineering.

2.1.1 Interfaces in software design

The methodologies that are used to design software interfaces have many benefits that make it worth to explore whether these methodologies can be applied to the design of physical interfaces. As such, the intention of this section is mainly to describe and illustrate a methodology that is known as *the principle of encapsulation* in software design. Later in Chapter 3, it is shown how the principle of encapsulation can be used to obtain a simple formulation for physical interfaces.

Software development puts much emphasis on the development of interfaces for software reuse. Software reuse is one of the main promises of object oriented (OO) programming. An OO program is a collection of interacting pieces of code

that are called objects. An object wraps both data structures and the functions that operate on the data structures into a single code unit. Objects are created by reusing a piece of code that is called a class code. It is the class code that defines the data structures and functions of its constituent objects once and for all. The class code can then be instantiated as many times as wished to create new objects. The data structures that are defined by a class are called attributes, and the functions that operate on these data structures are called methods. Any change to the attributes of an object, i.e., changing its state, must be done through relevant methods. The methods that change the state of an object are virtually its interfaces.

The area in software engineering where interfaces have essential significance is component based software engineering (CBSE) (Sommerville, 2007). A component can be an individual class or a group of semantically related classes. Some of the methods of the classes that constitute a component are responsible for interactions with other components in the system. These methods are called component interfaces. A component may provide or require an interface. An interface that offers services to other components is called a provided interface, whereas an interface that uses services from other components is called a required interface.

To effectively reuse components in different systems, they should be defined with regard to the principle of *encapsulation*. The intent of encapsulation is to isolate the internal structure of a piece of code from its users (Armstrong, 2006). The principle of encapsulation is closely tied to a more general principle called separation of concerns. Separation of concerns means system elements should have exclusive and singular purposes (Dijkstra, 1982). For example, explicitly defined component interfaces should be the only places through which it interacts with the rest of the system.

CBSE is an emerging software engineering paradigm with the hope of enabling black box reuse of software components (Szyperski et al., 2002). This means that

software components should be reused disregarding their source code, that is, *plugged and played* in any system (Clements, 1995). This is of course possible if components exclusively interact with each other through their interfaces. Well encapsulated software components can be interchanged if they have the *same* interfaces (Parnas, 1972).

Figure 2.1 illustrates how encapsulation and separation of concerns can be used in practice to hide a component implementation. Suppose that the goal here is to build an online shopping system. Figure 2.1 (a) shows some of the classes that are designed for this system in a Unified Modeling Language diagram (UML) (Rumbaugh et al., 2004). UML is a general purpose and standard modeling language that is used to create visual models of software systems.

Each class in Figure 2.1 (a) is visually modeled by a rectangle that is divided into three areas. The top area identifies the name of the class, the middle area identifies its attributes, and the bottom area identifies its methods. A solid arrow is drawn when a class depends on another class, that is, when it uses another class. A dashed arrow with triangular arrowhead is drawn when a class explicitly implements an interface, that is, when it provides an interface. A diamond on an association line means the class owns another class. The asterisk on an association line means many instances of the associated class are involved in the association. An interface is represented in the same way as a concrete class with the addition of <<*Interface*>> keyword to its heading.

Explicit separation of interfaces from classes is illustrated in Figure 2.1 (a). A *ShoppingCart* in this example does not care about the internal implementation of the *CardPayment* as long as the latter provides the *Payment* interface, that is, as long as it has *getPaymentAmount( )* method as specified by the interface. This class diagram can now be put into the form of a component diagram as shown in Figure 2.1 (b). In this example, let us assume that *CardPayment*, *CustomerDirectory*, and *ShoppingCart* are the three components that constitute

the online shopping system. Components can be as granular as individual classes, like this example, or as a collection of classes in more complex cases.

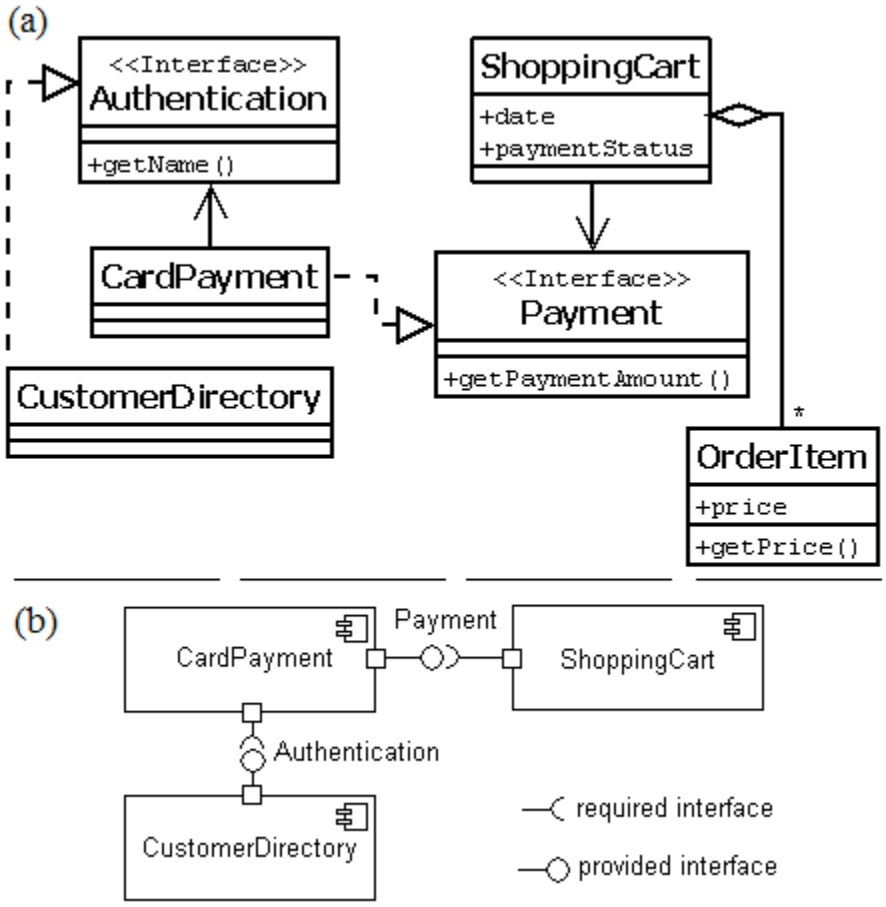


Figure 2.1: (a) The UML class diagram for an online shopping system. (b) The corresponding component diagram for the online shopping system.

In Figure 2.1 (b), one can replace *CardPayment* with another class, for example PayPal, if the replacement class provides *Payment* interface and requires *Authentication* interface. The internal implementation of replacement class, e.g., PayPal, could be anything; it only needs to satisfy the interfaces. This is called component interchangeability.

Figure 2.1 (b) also shows another important concept: The interfaces to a software component can be delegated to some of its constituting objects that are called

ports. Ports in UML are represented by small boxes at the boundaries of components. A port is an instance of an inner class that is exclusively in charge of a component's interface. Note that these inner classes are not shown in Figure 2.1 (a). The ports of a software component are the places where interactions with other components are supposed to happen.

Some of the concepts used in OO software development have been introduced to the systems engineering applications too. For example, the System Modeling Language, SysML (Friedenthal et al., 2008), is a language that has been recently proposed for visual modeling of both hardware and software systems. SysML is a language that is heavily founded on UML concepts, but it is smaller, semantically more complete, and less software centric. It has a range of enhancements over UML that allows it to be applied to modeling and analysis of a wider range of systems.

The atomic elements of most SysML's structural diagrams are *blocks*. Blocks in block definition diagrams of SysML play the same role as classes in UML class diagrams. The SysML counterpart of a UML component diagram is the internal block diagram, *ibd*. The interactions among blocks in *ibd* are represented by using a port notation. Ports are still represented by small boxes at the boundaries of blocks; however, SysML allows two types of ports to be defined. The first is a standard port. These are the places that exchange *services* with other components. The second type of port is an object *flow port*. A flow port identifies the kind of object that can flow in or out of an interaction point. A simple example is shown in Figure 2.2. The colon before the *Fuel / Air* in this figure indicates an unnamed object of the type *Fuel / Air*.

2.1.2 Interfaces in hardware design

The most prolific elements in conceptualization of interfaces in the electronic industry are *ports*. Ports are the locations where the inputs or outputs to a hardware component are defined. The inputs and outputs to electronics systems

are usually the flow of signals or electrical energy. In the specification of interfaces, both the physical properties of the signals and the data they carry are important, although more emphasis may be put on one than the other, depending on the situation.

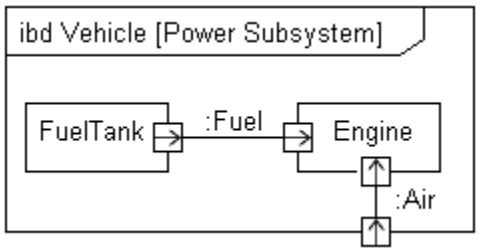


Figure 2.2: Object flow ports in SysML internal block diagram.

Electronics is an area of engineering in which many of the building blocks of its systems have been standardized from logical gates to integrated circuits and circuit boards. The existence of standard components has made it possible in some cases to automatically synthesize physical models from functional models. This is doable because the standard physical components that implement standard functions are widespread in electronics. For example, a logical circuit can be configured on a Field Programmable Gate Array (FPGA) just by specifying its functionality in a hardware description language (HDL).

Verilog HDL and VHDL[4] are two competing hardware description languages used to design logical circuits. Once the functionality of the circuit is defined by these languages, a physical circuit can be configured on an FPGA board automatically by synthesis tools. HDLs also use the concept of *ports* to define input/output interfaces among entities. For example, Verilog uses *in*, *out* or *inout* ports to specify input/output binary signals to an entity.

[4] VHDL stands for VHSIC Hardware Description Language, in which VHSIC itself stands for Very High Speed Integrated Circuits, a program that was launched by the Department of Defense.

2.1.3 Interfaces in mechanical design

The use of interfaces in the mechanical design of systems is less pronounced than that of hardware/software (HW/SW) engineering. This might have happened because it is probably easier to encapsulate and standardize functions than geometric forms and spatial relationships. HW/SW design is primarily concerned with the design of *functions* whereas mechanical design puts more emphasis on the design of *forms*. The more encapsulated the components are, the more pronounced their interfaces are.

In mechanical design, the most apparent place to look for interfaces is the mating constraints in assembly models. Mating constraints are binary relationships that are defined between low level geometric entities that exist in parts, such as faces, edges, axes and vertices (Lee and Gossard, 1985; ISO, 2004). Such low level and concrete mating constraints fit very well with the common *bottom up* practice of assembly design; that is, building an assembly model from parts. ICDs are, however, artifacts of top-down design.

One way to build a top-down assembly model is to use form features to define mating relationships. Shah and Rogers (1993) introduced the idea of feature based assembly modeling. An assembly feature is a group of mating constraints that are defined between the *form features* of two parts. Van Holland and Bronsvoort (2000) built a prototype assembly modeler that could be used for top down assembly design. It allowed users to instantiate compatible form features to define assembly connections.

This thesis also prescribes feature based representation of assembly connections for the purpose of defining mechanical interfaces. The reason is that form features better fit the idea of ports in mechanical components. A connection between two ports can be regarded as an assembly feature between their form features. In this way a unified model can be used to represent interfaces irrespective of the physical domain to which they belong. We can generally consider an interface as

an interconnection between two ports, whether the port is mechanical, electrical, etc.

There has been some recent interest in including port information in assembly models. One of the attempts to define ports in assembly models has been made by Singh and Bettig (2003). They suggested that port information be added to part models in order to automate the process of applying mating constraints in assembly models. They compared three different schemes to represent an assembly port; namely, as a single low level geometric entity, as a single form feature, or as a collection of all geometric entities that are intended for mating. The three schemes were quantitatively compared based on the efforts they impose on the automatic process of applying mating constraints.

For the benefit of this thesis, it is not possible to draw a definitive conclusion from Singh and Bettig's work on what the best scheme to represent assembly ports would be. Numerical criteria to define ports for the optimal process of assembly as proposed by Singh and Bettig cannot sufficiently address the issues that may arise in interface control. As such, this thesis relies on some conceptual criteria to define ports and interfaces in ICDs. The main criterion that is used in this thesis to define interfaces is that subsystems should be encapsulated to separate their internal structure from their interfaces.

Bettig and Gershenson (2010) also proposed some criteria to define interfaces for modular products. A module is not necessarily an assembly; it is a collection of highly dependent parts. They suggested that interfaces of modular products be classified into the following four groups to reduce the effort and the space required for their representation: attachment, control, transfer and field. However, the concept of connectivity between modules was absent in Bettig and Gershenson's work. The modules were just stored individually without defining how the interfaces of a module may interact with another in a system. In this thesis we are mainly concerned with proposing a methodology that defines the connectivity of subsystems in addition to their interface specifications.

2.1.4 Interfaces in simulation of mechatronic systems

Port based modeling has been a major paradigm used to simulate component behavior in mechatronic systems. In the work done by Paredis et al. (2001) and Sinha (2001), components were composed together via their ports to make a composite simulation model. Ports were considered as the places where energy and signals were exchanged among components. The interactions among components in the work by Paredis et al. and Sinha were defined based on the relationships among ports' effort and flow variables. The effort and flow variables are rooted in bond graph modeling (Rosenberg and Karnopp, 1983). For example, effort and flow variables map to voltage and current in the electrical domain, and velocity and force in the mechanical domain.

The work by Paredis et al. and Sinha described what could be regarded as a component behavioral model. Component behavior is defined by ordinary differential equations that show how effort and flow variables are related in a component, and by algebraic constraints that show how these variables are related to that of another component.

This thesis proposes to use a port based representation to define subsystem interfaces, but for the purpose of interface control. The interface specifications that appear in ICDs are very diverse; they are not just limited to power conjugate variables. As such, interface control issues should be dealt with by means of knowledge management tools; they cannot be dealt with by simulation models.

2.2 Markup and schema languages

One of the objectives of this thesis is to make ICDs machine readable. A markup language can be used for this purpose. The most widespread markup language that is in use today is HyperText Markup Language, HTML (W3C, 1999). HTML is a popular language used to construct a web page from text and media elements. The most important entities in an HTML document that markup the content are tags. Each tag is a keyword enclosed by angle brackets, e.g., <*p*> for paragraph. Tags

normally come in pairs, one of which marks the beginning and the other marks the end of a textual content, e.g., <*p*>*content*</*p*>. It is also possible to link a web page to another one by using the *anchor* tag. To do this, one needs to specify the URL (uniform resource locator)[5] of the target web page as an attribute of the anchor tag.

The above brief description of HTML exemplifies some of the main ingredients of *syntactic* web, i.e., a collection of documents (web pages), a unique scheme to identify these documents (URLs), and the linkage among them (hyperlinks)[6]. The role of HTML in a syntactic web is just to describe the presentation of web pages. It describes what part of a document is the header, what part is a paragraph, etc.; it does not know anything about the *content* of the document.

In syntactic web, it is the people, not computers, who take care of linkage and interpretation of contents. In order to get computers to do the work, a paradigm shift has already started to occur from *syntactic* web to *semantic* web. The current syntactic web is a web of unstructured documents. The semantic web, however, is a web of structured and semantic documents. The term semantic web was introduced by Tim Berners-Lee (2001), the inventor of World Wide Web and the director of World Wide Web Consortium (W3C). The movement to create semantic web is currently led by the W3C, where the aim of the movement is to develop the technology required to create semantic content for web pages.

The most basic task in creating semantic content is to encode textual data in a way that can be interpreted by computers. The eXtensible Markup Language (XML) (W3C, 2006) is a very popular tool used for such a purpose. It is designed to represent data in a way that is both human and machine readable. XML structures a document by wrapping data in tags. Like HTML, tags are defined by angle brackets. The main difference between HTML and XML tags, however, is that the tags in the former are predefined and mark up the presentation of the document,

[5] URL is a character string that indicates the location of a resource on the internet.
[6] A hyperlink is a direct reference to a document or data that the reader can follow.

whereas the tags in the latter are user defined and mark up the data within the document.

XML lies at the first level of the hierarchy of languages being developed for semantic web. XML can be used to define the semantics of the data that is embedded in an HTML webpage. However, only the creator of an XML marked data is aware of the meanings of the tags; XML by itself does not have any means to share the semantics of tags. A higher level language is required to actually define the semantics of tags in XML so that they can be shared among different agents. This goal can be partially achieved by using the XML Schema language, XSD (W3C, 2012); it is a language that describes the structure of XML documents.

It is possible to enforce consistency among a set of XML documents by committing to an XSD schema. This works well for agents in a stable and specialized community who want to share XML documents for some special purposes, for example, as in the case of ICDs. However, using XML/XSD technology alone falls short of meeting the ultimate goal of semantic web, which is to allow sharing of information across *all* applications and all community boundaries. Such a goal requires a more expressive language than XSD. W3C's Web Ontology Language, OWL (Horrocks et al., 2003) is the most recent attempt in designing such a general purpose and expressive language.

2.3 Ontologies

Ontology as a philosophical discipline is the systematic study of being, whereas *an ontology* as an engineering artifact is a vocabulary that describes the concepts in a certain domain as well as the explicit assumptions about the intended meaning of the concepts.

In engineering, ontologies act as a specification mechanism. "an ontology is an explicit specification of a conceptualization." (Gruber, 1995). The set of objects

that can be formally represented in a domain are called its universe of discourse. These objects and the relationships among them constitute an ontology, i.e., a vocabulary, which can be used by a knowledge representation program. The definition of the terms in an ontology specifies the meanings of the objects in a universe of discourse, and the specification of the relationships among the terms guarantees precise use of these terms.

2.3.1 The purpose of ontologies

The main thrust for the development of ontologies has been knowledge sharing and reuse. Similar to software engineering, knowledge components can be reused instead of building a new knowledge base from scratch (Neches et al., 1991). The two main challenges to be met in doing so are the lexical mismatches and the semantic problems among knowledge bases. Lexical problems emerge when different knowledge bases use different terminologies to represent the same concept. Semantic problems occur when the same term implies different meanings in different contexts. Ontologies constitute a dictionary to solve lexical and semantic problems in knowledge sharing (Gomez-Perez, 1997).

An ontology should be expressed by domain independent and machine readable languages. When agents, e.g., computer systems, share knowledge with others, they should commit to the terms and their meanings specified by the ontology. Ontological commitment means observable actions of an agent must be consistent with the definitions in the ontology (Gruber, 1995).

Ontologies can be classified based on the level of formality with which they are defined. An informal ontology is described by natural languages, whereas a formal ontology is codified by a formal and machine readable language. The most notable ontology representation language is OWL, but less expressive languages such as XSD can also be used to define simpler ontologies (Klein et al., 2000).

Depending on how they are intended to be used, ontologies can also be classified into different types (Gomez-Perez, 1997). There are several ontology categories that are defined in the literature, but the one that is related to this thesis is the domain ontology. Domain ontologies establish a vocabulary that describes the terms related to a specific domain, for example, medical domain, etc.

Domain ontologies can be used in different ways. Uschold and Gruninger (1996) identified three possible uses for domain ontologies: communication among organizations, interoperability among software systems, and systems engineering benefits. The usage of ontologies in systems engineering is more intended for design time issues. They provide a shared understanding of a system's domain. In this thesis we propose a domain ontology that captures the semantics of interfaces and improves the sharing of ICD information.

2.3.2 Ontologies in engineering design

The use of ontologies to solve knowledge sharing problems in engineering has been diverse. A complete account of all applications of ontologies to solve different engineering problems does not concern this thesis as many of such applications are not highly relevant to the subject of interface control. For a review of recent applications of ontologies in mechanical engineering, the reader can refer to Liu and Lim (2011).

2.3.3 Port ontologies for conceptual design

In engineering design, ontologies have been used to formalize knowledge during conceptual design of components (Horvath et al., 1998; Kitamura and Mizoguchi, 2003), to partition components into modules based on the semantic similarities of port functions (Cao and Fu, 2011; Cao et al., 2009), and to share the information of CAD assembly models (Kim et al., 2006; Patil et al., 2005).

Ontologies have also been proposed to support incremental refinement of design decisions made during conceptual design as proposed by Liang and Paredis

(2004). Their proposed ontology contained classes to define ports and their attributes. They suggested that port attributes should be defined by taking into account different design perspectives: form, function and behavior. In this thesis, a similar, but broadened ontology is proposed that is useful for interface control. Liang and Paredis' work was only aimed at the issues that arise in conceptual design, whereas this thesis uses an ontology, firstly, to semantically describe the interactions of subsystems in ICDs, and secondly, to semantically define the actions that designers must take to fix violated interfaces.

There are many technical issues in the use of an ontology for interface control that need to be addressed before it can be successfully used for such a purpose. One is how to ensure the interface definitions that come from different sources are actually consistent with the ontology. This thesis addresses these issues by proposing a software architecture that is responsible for checking completeness and correctness of interface definitions.

2.4 Summary

Port based representation is one of the most widely used methods to represent interfaces in HW/SW engineering. Port based representation has recently come to more prominence even in mechanical engineering. Ports are now included as features in the core product model that is proposed by the National Institute of Standards and Technology (NIST) (Fenves et al., 2004).

Port based representation of component interactions is more useful for a top down design process than a bottom-up process. ICDs are artifacts of a top-down design process. As such, this thesis prescribes a generic port based representation for interface definitions. It is assumed in this thesis that ICDs are written for subsystems that are well encapsulated. Such subsystems exclusively interact with each other through ports.

Encapsulation is a good design practice that improves reusability and maintainability of components, even though it is not as widespread in mechanical engineering as in software engineering. This may be because the emphasis in mechanical engineering is more on the design of forms than functions, which are probably harder to encapsulate.

ICDs can be made machine readable by using available technology. The XML language is the current standard for information transfer on the web. XML is used in this thesis to define machine readable interface definitions. The structures and definitions of the tags in XML documents can be defined by XSD language. XSD is used to actually get different agents to agree and commit to a specific group of tags used to create an XML document; hence, XSD is used in this thesis to define the interface ontology. The expressive power of XSD is enough to represent the ontology that is defined within the boundaries of this thesis.

The next chapter proposes a formal interface representation model for ICDs. The model can uniformly be applied to different physical domains such as mechanical and electrical interfaces. After defining the formal model, an ontology that defines the semantics of interfaces is presented. The use of the ontology is very important in enabling information sharing and in tracking the incompatibilities of interfaces.

3 INTERFACE REPRESENTATION MODEL

A formal model for interface definition is the first step to have ICDs managed by computers. This chapter presents an interface representation model that can be used to create computerized ICDs as it is expressible by a computer readable language, e.g., XML.

Before proposing the interface representation model, it is necessary to make a distinction between the concept of component[7] and the concept of interface. Unlike a component, an interface is not an *independent* entity; it occurs as a result of the interaction of the *boundaries* of *two* components; hence, a component always exists independently of how and where it is used, but an interface exists only if *two* components interact.

Both components and interfaces in a system can be described by mathematical models. A component model usually uses a set of differential and algebraic equations that describe the component's behavior. An interface model, on the other hand, exclusively describes the *compatibility* of component boundaries at any point of interaction. Interface models consider components as black boxes whereas component models consider them as white boxes. The internal structure of the components in an interface model should entirely be hidden; only the interacting boundaries of components should be visible in the interface model. This differentiation between component models and interface models contrasts the essence of interface control as opposed to component design.

An interface between two components *can be* established if their peripheral properties are *compatible* at the virtual plane at which their boundaries come together. For this to happen, the peripheral properties of the two components must satisfy a given compatibility relationship. Figure 3.1 illustrates this concept. The x

[7] The terms component and subsystem are used interchangeably in this thesis.

and y symbols in the figure represent two peripheral properties of components $c1$ and $c2$. The symbol $\preccurlyeq \in \{<, >, \leq, \geq, =\}$ is a relational operator that defines the compatibility relationship $x \preccurlyeq y+\delta$. The semantics of the relational operators depends on the properties they relate. The semantics of these operators is clear for simple real valued properties.

The areas of a component's boundary through which it interacts with other components are called ports. The component's peripheral properties that take part in interactions are *delegated* to ports and called *port attributes*. In each interface, the attributes of one port are related to the attributes of another port. For example, x and y in Figure 3.1 are port attributes of $c1$ and $c2$, respectively.

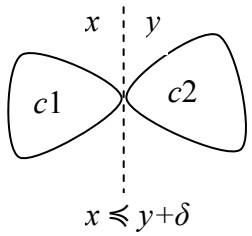


Figure 3.1: Interaction of two component boundaries.

As can be seen in Figure 3.1, a component in the interface model is analogous to an area; its boundary is analogous to a curve that surrounds this area; and a point on the curve is analogous to a port. A point cannot come into contact with more than one point; the same is true for ports. This means that in the interface model, all port to port interactions between components are independent.

In this thesis, an object oriented notation is used to define ports and their attributes. In this notation, different *types* of ports are represented by different *classes*. A class in object oriented terminology is an abstract representation of a group of individuals, called objects, which share common attributes.

Objects of a class are defined by assigning different *values* to the set of attributes that are defined by the class. For example, the concept of *pin_port* defines a class. All objects that belong to this class have a cylindrical shape that can be minimally described by a diameter and a length attribute. This class can be instantiated to different objects by assigning different values to these attributes; that is, different objects of the *pin_port* class have different diameters and lengths.

In general, attributes can be quantitative or qualitative. In the domain of physical interfaces, we are mainly interested in quantitative attributes. Quantitative attributes can be measured by numbers; hence, they have a magnitude and a unit of measure. The diameter of a pin is an example of a quantitative attribute.

The semantics of the attributes can be defined by using fully qualified names. The unqualified names of the attributes need not be different. For example, we can have the *hole_port* class whose attributes have the following unqualified names: diameter and length. We can also have the *pin_port* class whose attributes have the same unqualified names, i.e., diameter and length. The difference between the attributes of two classes becomes clear when the fully qualified names of the attributes are considered. For example, one talks about *the diameter of a pin port* and *the diameter of a hole port*, not just *the diameter*. In this regard, the diameter of a pin port and the diameter of a hole port are two different attributes.

Qualified names also distinguish different classes since two different classes may have the same unqualified name. For example, it is possible to have the *pin_port* class and the *pin_componet* class, which have the same unqualified name, i.e., *pin*. However, the former defines a class of ports whereas the latter defines a class of components. These two classes are related though; the pin port class is a *part of* the pin component class. Since this thesis is mostly concerned with port classes, the *_port* postfix is dropped in the majority of cases. In the discussions that follow, fully qualified names are avoided as much as possible to simplify the text.

3.1 Interface formalism

Definition 1. Let P and Q be two port classes and $p \in P$ and $q \in Q$ be two objects of these classes. An *object compatibility* relationship $M_{p,q}$ between p and q is the set of *binary constraints* over a subset of the attributes of p and a subset of the attributes of q. Each constraint in $M_{p,q}$ relates an attribute of p to exactly one commensurable attribute of q.

Let A_P and A_Q be the set of attribute symbols of P and Q, respectively. The attribute symbols of p and q are also chosen from A_P and A_Q. Each attribute $x \in A_P \cup A_Q$ has a domain D_x from which its value can be chosen. Further, let $B = \{x_1, \dots, x_n\} \subseteq A_P$ and $C = \{y_1, \dots, y_n\} \subseteq A_Q$. Note that $|B| = |C| = n$. An object compatibility relationship between p and q can be formally defined as follows:

$$M_{p,q} = \{\langle \{x_i, y_i\},\ R\rangle \mid x_i \in B \wedge\ y_i \in C\ \wedge\ R \subseteq D_{x_i} \times D_{y_i}\} \tag{3.1}$$

in which $\{x_i, y_i\}$ is a pair of commensurable attribute symbols of p and q, and R is a relation that is called a constraint over $\{x_i, y_i\}$, which can be denoted as R_{x_i,y_i} or $x_i R y_i$. This view of compatibility relationships is analogous to the concept of a binary constraint network (Dechter, 2003). The binary network here is composed of the relationships between the attributes of p and q. The domain of attributes in the case of physical interfaces is usually the set of real numbers $\mathbb{R}$; hence, $R \subseteq \mathbb{R}^2$.

When the *same* compatibility relationship $M_{p,q}$ is defined *for all* $p \in P$ and $q \in Q$, it is called a *class compatibility relationship* $\mathcal{M}_{P,Q}$.

In physical interfaces, R_{x_i,y_i} usually relates two real valued attributes x_i and y_i that can be defined by using a relational operator $\preccurlyeq \in \{<, >, \leq, \geq, =\}$ and positive constants δ_l and δ_r, e.g., $y_i - \delta_l \preccurlyeq x_i \preccurlyeq y_i + \delta_r$.

Port objects are specified by assigning values to their attributes. This is called an instantiation. Let $v_p(x) \in D_x$ be a variable assignment that assigns a value to an attribute x of p from its domain D_x. Port objects p and q are instantiated from their class P and Q by assigning values to their attributes.

Definition 2. Two port objects p and q are compatible with regard to an object compatibility relationship $M_{p,q}$ if $\forall\langle\{x, y\}, R\rangle \in M_{p,q} \Longrightarrow \langle v_p(x), v_q(y)\rangle \in R$. Likewise, two port objects p and q are compatible with regard to a class compatibility relationship $\mathcal{M}_{P,Q}$ if $\forall\langle\{x, y\}, R\rangle \in \mathcal{M}_{P,Q} \Longrightarrow \langle v_p(x), v_q(y)\rangle \in R$.

Note that deciding whether two ports are compatible or not requires two sources of information: a compatibility relationship (for classes or objects) and a value assignment. Due to the latter, it does not make sense to say two port classes are compatible because attributes in classes are unassigned; so, it only makes sense to say two port objects are compatible with regard to either an object or a class (universal) compatibility relationship.

The above definitions correspond to the two-sided view of interfaces that is mentioned in §1.3.3. Recall that there can be a one-sided view of interfaces too. A one-sided interface view can be regarded as a set of constraints on the attributes of a single port.

Definition 3. An object *requirement* L_p for a port object p is the set of *unary* constraints over its attributes.

$$L_p = \{\langle x, R\rangle \mid x \in A_P \ \wedge \ R \subseteq D_x\} \tag{3.2}$$

In the above definition, the relation R simply specifies a subset of the domain of x. For real valued attributes, each unary constraint in L_p specifies an interval of real numbers.

When the *same* requirement L_p is defined *for all* $p \in P$, it is called a *class requirement* $\mathcal{L}_P$.

Definition 4. A port object p is *eligible* with regard to an object requirement L_p if $\forall \langle x , R \rangle \in L_p \Longrightarrow v_p(x) \in R$. Likewise, A port object p is *eligible* with regard to a class requirement $\mathcal{L}_P$ if $\forall \langle x , R \rangle \in \mathcal{L}_P \Longrightarrow v_p(x) \in R$.

The interface formalism presented so far defines compatibility relationships between ports as a set of *binary* and *independent* constraints between their attributes. The interface model that is based on this formalism considers subsystems as black boxes with entirely invisible internal design. In this model, interfaces are regarded as port to port interactions as shown in Figure 3.2. Note that interfaces between a set of subsystems are defined by considering two of them at a time. This is expected because ICDs themselves define interfaces between every pair of subsystems. The purpose of ICDs, and therefore the interface model proposed here, is to control the design of interfaces, not to provide a system wide model that describes system behavior. In this regard, ICDs should not be confused with assembly models, system wide block diagrams, etc., that show the overall system view or behavior.

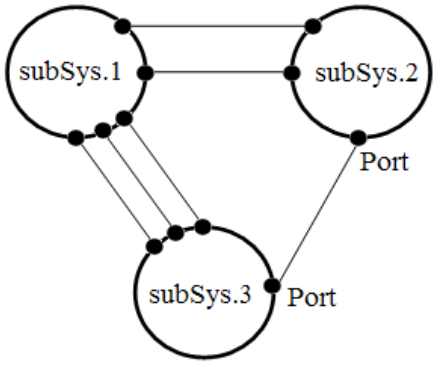


Figure 3.2: Interfaces as port to port interactions (binary) between two subsystems at a time.

The aforementioned interface model can be used to define all types of physical interfaces between subsystems. We *postulate* that all physical interfaces can be defined based on the binary constraints among port attributes if subsystems are well encapsulated (black boxes).

The example shown in Figure 3.3 is intended to make the idea of black box encapsulation clear even though it is a purely mechanical example that may not need an ICD. The figure shows three components: a rectangular bar and two discs. Components *c*1 and *c*2 interact through ports *p*1 and *q*. Components *c*1 and *c*3 interact through ports *p*2 and *r*. The interaction between p2 and r may seem unrealistic, but this is not the point of this example. The point is to show that in black box encapsulation, an interface can be defined by a set of independent binary constraints.

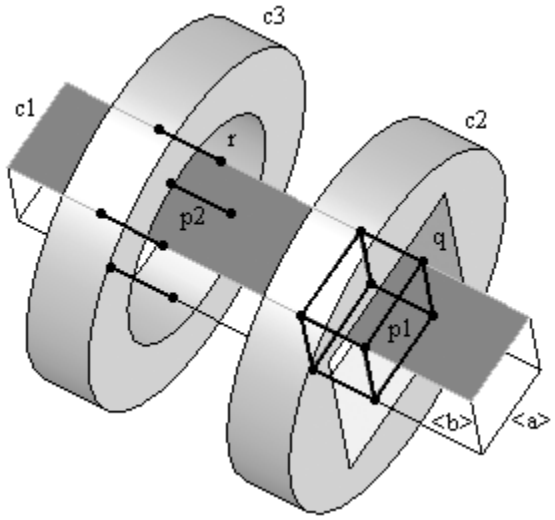


Figure 3.3: Black box interface formulation.

The interface model that is proposed in this chapter strictly considers interfaces as the places where the boundaries of two components come together; hence, the interface between *p*1 and *q* is different than the interface between *p*2 and *r*. *P*1 and *q* are collections of four rectangular surface segments, whereas *p*2 and *r* are collections of four line segments.

Assume that the side lengths of the rectangular cross sections at $p1$, $p2$, and q are all denoted by unqualified attribute names *a* and *b*. Further assume the diagonal of the rectangular and circular cross sections at $p2$ and r are both denoted by the unqualified name *dg*. Let us use the dotted notation *p.t* to refer to the qualified name of the attribute *t* of a port *p*. With this notation, the compatibility relationships in this example can be written as:

$$p1.a \leq q.a + \delta_1$$
$$p1.b \leq q.b + \delta_2$$
$$p2.dg \leq r.dg + \delta_3$$

which represents a set of linear, binary, and independent inequalities[8] as each pair of attributes appear in only one inequality. Constants δ_1, δ_2, and δ_3 in the above inequalities represent the *fit tolerances* between the ports. In the above example, it is tempting to relate *a, b* and *dg* together because by looking at Figure 3.3 one can see that $p1$ and $p2$ actually have the same dimensions, that is,

$$p2.dg = \left|\sqrt{(p1.a)^2 + (p1.b)^2}\right|$$

However, the above equation should be avoided in a well encapsulated interface model because we can only be aware of the above equation by seeing the overall shape of $c1$, which we could not see if it was a black box. The interface model needs to know that $p2$'s attribute is $p2.dg$, but it should not know how *dg* is actually calculated from component $c1$. The latter is the responsibility of the component designer, not the ICD. Therefore, the compatibility relationships in a well encapsulated system can be represented by a set of binary, independent, and linear inequalities.

Object requirements in Figure 3.3 can be formulated by specifying the convex real intervals from which the values of the port attributes can be chosen, for example

[8] If necessary, the above binary constraints can also be defined as $q.y - \delta_1 \leq p.x \leq q.y + \delta_2$, which is still a binary constraint between *p.x* and *q.y*. Using one relational operator is for brevity.

$$p1.a_{min} \leq p1.a \leq p1.a_{max}$$

in which $[p1.a_{min}, p1.a_{max}]$ represents a *dimensional tolerance* for $p1.a$.

In general, by using black box encapsulation, the compatibility constraints between port attributes can be formulated as

$$q.y - \delta^{(1)}_{p.x,q.y} \preccurlyeq p.x \preccurlyeq q.y + \delta^{(2)}_{p.x,q.y} \tag{3.3}$$

in which δ is a positive number. Port requirements can be formulated as

$$p.x \in [p.x_{min}, p.x_{max}] \tag{3.4}$$

The inequalities defined by (3.3) and (3.4) only tell us whether the ports of each pair of components are compatible. They do not tell us whether the entire assembly model shown in Figure 3.3 can actually work. The latter question can only be answered if the entire design of components, e.g., their shapes, is known. When put together, the components' boundaries may unintentionally interact, i.e., *interfere*. This is not, however, a question that can be answered by an interface model. Interface definitions are analogous to assembly mating constraints. Mating constraints only show how each pair of components are connected. They cannot guarantee that there would be no interferences between components after they are mated.

3.2 Function attributes

The example given in the previous section only illustrates compatibility relationships between form features. In cases where only form features are concerned, an assembly model is usually enough to show the interactions; there is no need to write ICDs for such cases. There is a lot of information, however, that cannot be captured by assembly models for which writing ICDs is justified.

In some cases, for example, in a large number of electrical and electronic connections, form attributes have less significance compared to the attributes that define the function of a port. In fact electrical interfaces are sometimes called functional interfaces in contrast with mechanical interfaces (Lalli et al., 1997).

Function attributes can be represented by *verb-flow* pairs, as defined in "*functional basis*" (Hirtz et al., 2002). The verbs that most relevantly describe the functions of components at interconnections are *support*, *connect* and *channel*. Support function means the component is intended to secure or position a flow into a defined location. Connect function means the component is intended to bring some flows together. The most frequently used subfunctions of 'connect' are *join* and *link*. Channel function means the component causes a flow to move from one place to another. The most frequently used subfunctions of 'channel' are *transport* and *transmit*. All flows are subclasses of *Signal*, *Material*, and *Energy* classes.

The primary purpose of "*functional basis*" was to classify the functionality of *components* in product design. The same scheme can also be used classify the functionality of *interfaces* if the subtle difference between the functionality of components and interfaces is understood. The functionality of an interface is always a two sided phenomenon; two ports with the same function constitute the functionality of an interface.

Like forms, flows can also be described by flow attributes. To specify an energy flow, the attributes that describe the transmission of power should be given. For example, voltage and power are the two most general attributes of an electrical energy flow. To specify a signal flow, the attributes that describe the waveform should be given. To specify a material flow, the attributes that describe the transport of material should be given. The three generic flow classes, i.e., material, energy, signal, can be hierarchically divided into more specific subclasses as shown in Figure 3.4 to semantically define different types of flow.

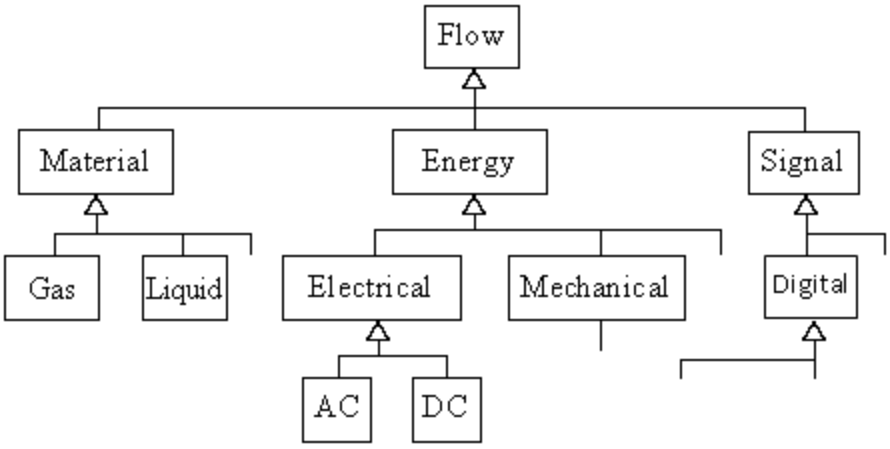


Figure 3.4: Flow hierarchy.

3.3 Aggregate ports

In some instances, multiple ports are grouped together into a single package that provides an interaction point on a subsystem boundary. These ports are called aggregate ports. Note that in addition to ports, form and functions can also be aggregates.

To correctly connect two aggregate ports, the entire collection of subports in one port must correctly match with that of the other. For this to happen, the subports in the two connecting aggregate ports must have the correct order. As such, a generic method to formulate aggregation constraints between aggregate ports is needed. An example of an aggregate port is shown in Figure 3.5. This thesis only considers 2D aggregate ports since it is rare to see a 3D arrangement of ports.

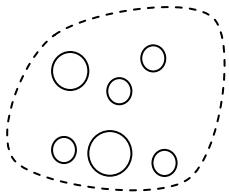


Figure 3.5: An aggregate port that is composed of six circular hole subports.

Following is a description of an aggregate port matching method that can be applied to any 2D arrangement of subports, if the *centers* and *orientations* of subports' in the 2D plane can be defined unambiguously. One group of shapes that satisfies this condition and most frequently happens in 2D arrangements is a regular polygon. Note that a circle is also a regular polygon with an infinite number of sides.

To check whether two 2D arrangements of subports match with each other, three tests are needed. The first test is to check whether the sets of center points of the polygons are congruent. The second test is to check whether at each coincidence between the centers, the shapes of the corresponding subports have the correct orientations and dimensions. The last test is to check whether at each coincidence between the centers, the functions of the corresponding subports are the same.

To consistently do the above tests, the subports in aggregate ports must be ordered. Two aggregate ports match if there is one-to-one mapping between their subports according to the given order. Defining a consistent method for subport ordering and checking the congruency between the center points of the elements of two aggregate ports are closely related issues.

The problem of finding the congruency of two sets of points can be regarded as a simplified version of the problem of finding congruency between two planar figures (Atallah, 1984). Two planar sets of points are congruent if one can be made coincident with another by a translation or a rotation. If two sets of points are coincident, their centroids[9] must be coincident too. Therefore, to align one set of points to another, we can first do a translation to make the centroids of the two sets coincident, and then rotate them around the centroid to make all individual points coincident. This means that the congruency of two sets of points can be checked by placing their coordinate frames at their centroids and finding out if one set is a rotated version of the other.

[9] The centroid *C* of a set of points D_i is defined by $\sum_{i=1}^{n} \overrightarrow{CD_i} = \vec{0}$.

Let A and B be two planar sets of points that represent the center points that belong to the subports of two aggregate ports p and q, respectively. The congruency between A and B can be decided if the points in A and B are defined in polar coordinate systems with the poles Cp and Cq placed at the centroids of A and B, respectively. The position of the polar axes can be arbitrarily chosen to go through a point in A and B. We also assume that the sets of points in A and B are viewed from the outside of their corresponding subsystems, that is, the z axes of the aggregate ports point away from the subsystems.

If p and q are compatible, the set of points defined by A and B must be oppositely congruent. Opposite congruence means a mirrored version of the points in A, denoted by A', is directly congruent to the points in B. A' is obtained by reflecting the points of A relative to an arbitrary axis in its plane, e.g., the polar axis of A. The reflection about the polar axis is easily obtainable by the replacement $\theta \leftarrow 360° - \theta$, where θ is the angular coordinate.

Let A' be partitioned into classes $A'_1, \ldots, A'_n$, where A'_i is the set of points that have the same radial coordinate r_i, and $r_i < r_{i+1}$. B is partitioned into $B_1, \ldots, B_m$ in the same manner. If $n \neq m$, or if for some i, $|A'_i| \neq |B_i|$, then A' and B cannot be congruent; otherwise, they are congruent if the points in A' are a rotated version of the points in B.

To check whether A' is a rotated version of B, suppose that the polar axis $L1$ for all points in A' goes through an arbitrary point in A'_1. Suppose also that the polar axis $L2$ for all points in B also goes through an arbitrary point in B_1. Let $\sigma = \alpha_1, \alpha_2, \ldots, \alpha_k$ be a string that represents the sequence of relative angular displacements of the points in A'_1. Likewise, $\rho = \beta_1, \beta_2, \ldots, \beta_k$ represents the relative angular displacements of B_1. These two partitions are a rotated version of each other if ρ is a cyclic shift of σ, which is true if ρ is a substring of $\sigma\sigma$.

The relative rotation from A'_1 to B_1 can be computed by finding the index j by which ρ occurs in $\sigma\sigma$. The set A'_1 can be obtained from B_1 by the relative rotation $\theta = \alpha_1 + \cdots + \alpha_{j-1}$ for $j > 1$. For example in Figure 3.6, if $\sigma = \alpha_1, \alpha_2, \alpha_3 = 30°, 60°, 270°$ $\rho = \beta_1, \beta_2, \beta_3 = 60°, 270°, 30°$, then ρ is a cyclic shift of σ with $j = 2$ and $\theta = \alpha_1 = 30°$. A' and B are congruent if the same relative rotation θ holds between all other partitions A'_i and B_i.

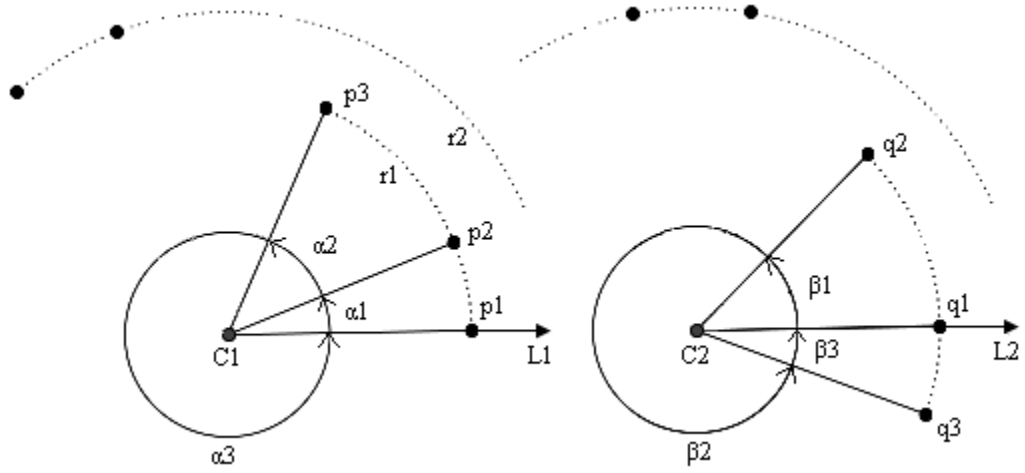


Figure 3.6: Congruency checking between the points in A' (left) and B (right).

With the above algorithm at hand, it is possible to generically check whether two aggregate ports have the correct order of subports. The constraints between the two patterns can be represented in the compact form *p.order* = *q.order*, in which the attribute '*order*' abstracts the ordering of the ports defined based on the partitioning method mentioned in the above algorithm, and '=' operator signifies that the positions, orientations, dimensions, and functions of the subports of *p* and *q* must match.

3.4 Interface Semantics

The attributive interface representation described in this chapter is clearly computer manageable; constraints are formally defined and ports as well as their attributes are represented in an object oriented model. However, a symbolic

representation of attributes by itself cannot enable information sharing in a collaborative environment. Identifying attributes in the form of x and y by one organization is often meaningless to the designers of another organization.

Interface control documents are meant to be shared. To share any interface control document, the semantics of the terms used in it should be known to its users. This applies to the formal interface representation model presented in §3.1 too. The interface model addresses the issue of *compatibility* between ports, but the model works if interface definitions that come from different sources are *consistent*, that is, defined using the same terminology. The semantics of ports and their attributes should be known before the interface model is shared by different organizations.

Information sharing between design teams who share interface definitions can be improved by using an ontology. The ontology can be organized based on the framework that has been discussed so far in this chapter, that is, to define the port classes based on their form and function attributes. This framework actually sets the architecture of the ontology. It makes the ontology objective. The top layer of Figure 3.7 shows the architecture of the ontology that can be used for interface control.

3.4.1 Ontological concepts

Any concept in the ontology must adhere to the architecture shown in Figure 3.7. Based on the figure, any port in the ontology is conceptualized in terms of its form and function attributes. A port must also have a coordinate frame. It has already been seen why coordinate frames are important: to define the ordering of subports relative to each other in aggregate ports. The coordinate frame of a port is equivalent to the coordinate frame of its form.

The domain ontology that is illustrated in Figure 3.7 can be a vast layer. The concepts in the domain ontology reflect the terminology that is used in a target domain, e.g., a flight simulator domain, an aircraft domain, etc. All concepts in

the domain ontology are *instances* of the concepts set by the ontology architecture. For example, one can define a USB port, a coupling port, etc., as instances of the port concept. These port instances must define what form and verb-flow attributes they are going to have.

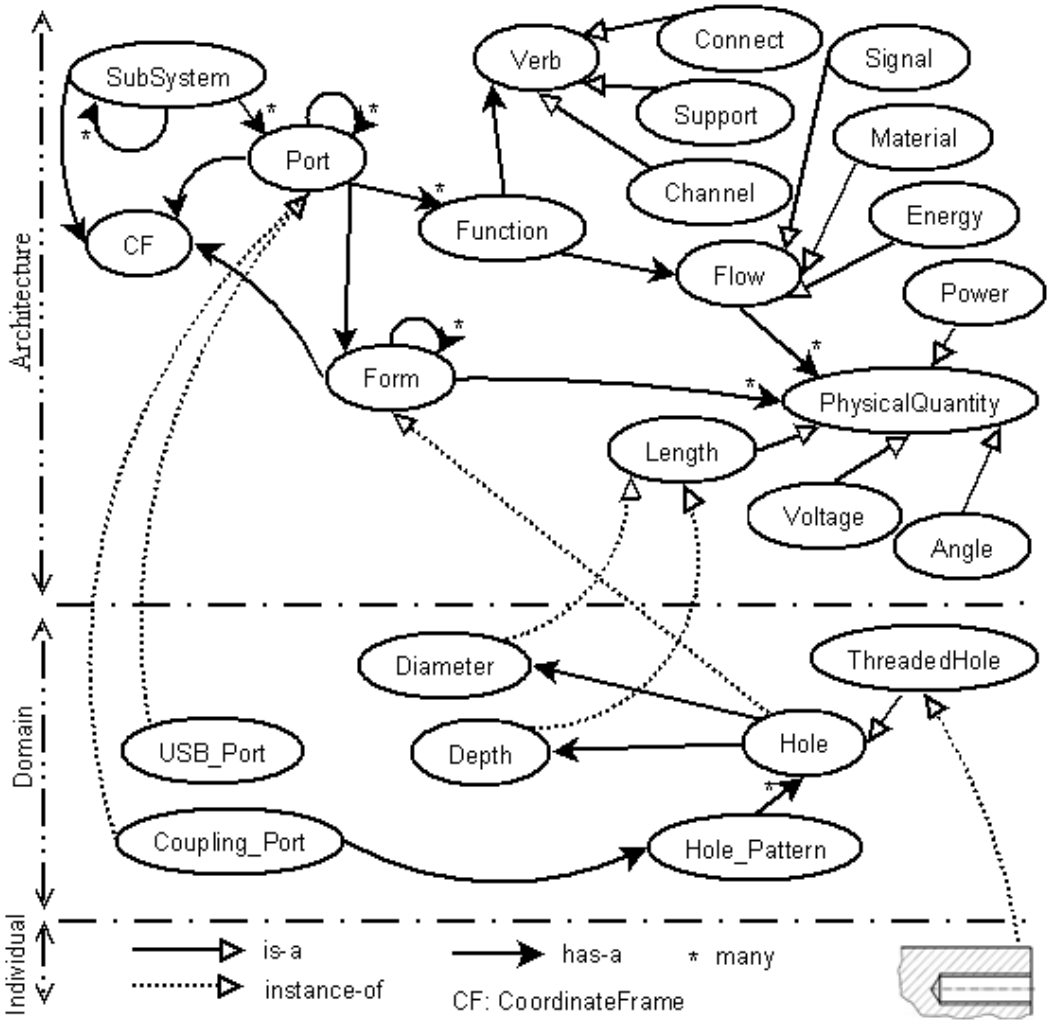


Figure 3.7: A partial port ontology.

Form and verb-flow attributes are chosen from the form and verb-flow instances that are available in the domain ontology. For example, one can define *Coupling_Port* as an instance of port that has a pattern of *Hole* as an instance of form. Hole has *Diameter* as an instance of *Length* physical quantity.

The physical quantity class shown in Figure 3.7 plays a pivotal role in checking commensurability of attributes in physical interfaces. Any *form* or *flow* attribute is eventually described by some physical quantities, which have a magnitude, a unit of measure, and a physical dimension. For example, a hole can have a hole-depth, which has the physical dimension of length (L). Any two attributes that have the same physical dimension are commensurable; hence, they can be related by a constraint.

The bottom layer that is shown in Figure 3.7 is not an ontology layer; it is where *individuals* are defined. Individuals are instances of the concepts in the domain ontology. The distinction between the concepts in the domain ontology and *individuals* can be subtle in some cases, but the distinction is clear in the ontology that is presented in this thesis. The concepts in the domain ontology are classes, whereas individuals are objects of these classes.

3.4.2 Ontological relationships

An ontology is not merely a collection of concepts; it also shows the relationships among concepts. A formal ontology should rigorously define all relationships among concepts. The relationships in ontologies are called *properties*. The arrows in Figure 3.7 represent the properties that exist in the port ontology, all of which are subproperties of *has-a* and *is-a* generic properties. *Has-a* property represents an ownership relationship whereas *is-a* property represents a subclass relationship.

For simplicity, the exact names of properties are not shown in Figure 3.7, but they can be easily understood from the context. For example, the property that maps the port class to the form class can be understood as *has_form* property of the port class. When shown with an asterisk, *has-a* property represents an aggregation.

It should be stated here that providing a full scale domain ontology is not the intention of this thesis. Such an ontology would indeed be extremely large; it is

not a goal that can be achieved by a single person or even a group of experts. Developing a large ontology may even need the use of mass collaboration tools, such as a Wiki technology (Hepp et al., 2007; Wongthongtham et al., 2009). What this thesis intends to do, however, is to show how the ontology can be used to control interfaces by computers.

3.4.3 Semantic representation of constraints

The semantics of compatibility relationships represented by (3.1) and (3.2) is defined by choosing class symbols from the domain ontology. To elaborate how interface constraints can be semantically represented, let $C(x)$ be a predicate that holds if x is an instance of class C, and $P(x, y)$ be a predicate that holds if x is mapped to y by property P. If x is a variable, it is prefixed with a question mark as $?x$. P and C are chosen from a port ontology.

A class requirement has the following generic form:

$$L = Port(?p) \wedge F(?p, x) \wedge hasAttribute(x, t) \wedge Attribute(t) \wedge \varphi_t \quad (3.5)$$

in which φ_t is a predicate that holds if a constraint on attribute t is satisfied. For example, if t is required to be a constant, φ_t is written as $\varphi_t \equiv (t = a\_constant)$. $F(?p, x)$ has either of the two following forms:

$$F(?p, x) = hasForm(?p, x) \wedge Form(x) \quad (3.6)$$

or

$$F(?p, x) = hasFunction(?p, fn) \wedge Function(fn) \wedge hasFlow(fn, x) \wedge Flow(x) \quad (3.7)$$

A class compatibility constraint between two attributes has the following generic form:

$$M = Port(?p) \wedge Port(?q) \wedge F(?p, x) \wedge hasAttribute(x, t) \wedge$$
$$Attribute(t) \wedge F(?q, y) \wedge hasAttribute(y, s) \wedge Attribute(s) \wedge \psi_{ts} \quad (3.8)$$

in which ψ_{ts} defines a compatibility constraint on two commensurable attributes t and s. For example, if t has to be less than s, then $\psi_{ts} \equiv (t < s)$.

3.5 Summary

This chapter has provided a formal interface representation model in terms of compatibility constraints among ports of two components. The interface model treats components as black boxes. It is shown later in Chapter 5 that this interface model can actually be expressed by a computer readable language such as XML and create computerized ICDs.

This chapter has also discussed what sources of information should be used to define port attributes. Any port can be specified by defining its form, function, and fit. Forms are specified by form attributes. Functions are specified by pairs of verb-flow attributes. Specification of flows constitutes the most important part of a port's functional information in interface definitions. The fit of an interface is defined based on the compatibility constraints between port attributes. The specification of form, verb-flow, and fit can describe a large number of physical interfaces.

Finally, this chapter has shown how the semantics of the ICDs should be defined so that they can be shared among different organizations. Interface semantics is captured by an ontology that provides a vocabulary to define ports, forms, and verb-flows.

The next chapter describes how violations of semantic constraints can be checked and *traced* by a knowledge base. Tracing the violations in a meaningful manner speeds up collaboration. A meaningful manner means that the exact location and

cause of the violation should be reported to designers. In this way the designers know what exactly needs to be done to fix the problem.

4 INTERFACE CONTROL

This chapter presents a control mechanism that can be used by a piece of software to check the status of interfaces. The software operates on semantic interface data. In this chapter, first, the control functionality of the software is formally presented. Next, a possible architecture for the interface control software is proposed.

Interface control software needs to check the consistency and compatibility of interface definitions. The former requires binding the software to an ontology to ensure the interface definitions that come from different sources are consistent, i.e., correct and complete. The latter requires a checking mechanism that detects errors in the value assignments to port attributes. This chapter discusses the mechanism that is used for compatibility checking. The consistency checking functionality is mainly discussed in the next chapter where the details of the ontology binding mechanism are given.

4.1 Control[10] mechanism

One of the main purposes of interface control processes is to ensure compatibility of interface definitions. Interface control software should play the same role. Such software is primarily a *checker*, i.e., a *detective tool*, not a corrective tool. This means that interface control software cannot decide a value for a port attribute, for example, by trying to find a solution for interface constraints. Such an assignment is almost certainly meaningless to component designers because interface constraints are just a part of the information that is needed for the design of components, not all of it. Port attribute values should be assigned by component designers based on both component design criteria and interface constraints. What

[10] The use of the term "control" here is due to its widespread use since otherwise ICDs are reference documents that are used to check the status of interfaces; hence "checking mechanism" would be a better term in this context.

the interface control software can do is to ensure value assignments by collaborative component designers do not violate interface constraints.

The first thing that can be checked is to ensure a user does not carelessly define a bad constraint. Suppose two attributes x and y are constrained by the following compatibility and requirement constraints:

$x \leq y + \delta$
$x \in [x_{min}, x_{max}]$, $y \in [y_{min}, y_{max}]$

The above formulation is satisfiable if

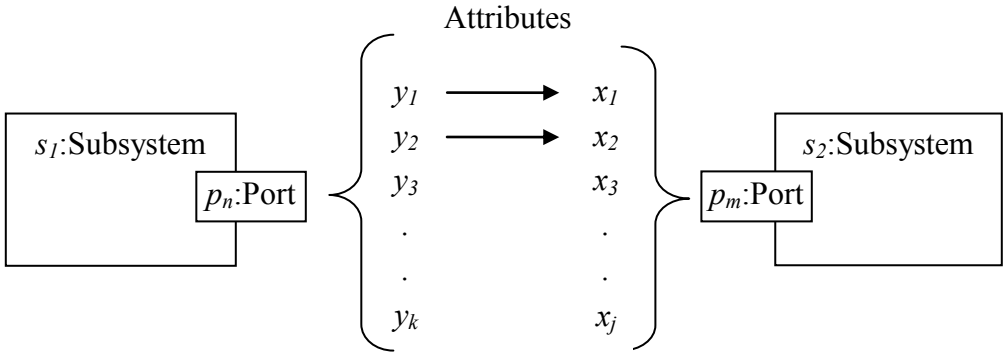
$x_{min} \leq y_{max} + \delta$.

This test still focuses on the definition of constraints, not the value assignment to port attributes during subsystem design. It should be done immediately when the user defines the constraints, that is, before the constraints are used for compatibility checking.

A compatibility checking mechanism checks whether the value assignments to port attributes are legitimate. Figure 4.1 illustrates the compatibility checking mechanism that is used in this thesis. The figure illustrates a port requirement and two compatibility constraints, which are going to be monitored by interface control software. Any value assignment to a port attribute that violates a constraint is reported to designers in a *traceable* and *meaningful* manner.

Violations of requirement and compatibility constraints are recorded in a traceable way as illustrated in Figure 4.1. To record a requirement violation, two elements are needed: a port identifier and a requirement constraint identifier. An error message is created based on the requirement constraint identifier and added to the list of errors maintained by the port. To record a compatibility violation, three elements are needed: two port identifiers and a compatibility constraint identifier.

An error message is created based on the compatibility constraint identifier and added to the error lists maintained by both ports.



Requirement:	$\{\langle s_2.p_m.x_1 \leq Constant\rangle , \ldots\}$
Compatibility:	$\{\langle s_2.p_m.x_1 = s_1.p_n.y_1\rangle , \langle s_2.p_m.x_2 = s_1.p_n.y_2\rangle , \ldots\}$
Control:	Record (p_m , *"violation of* Rx_1*"*)
	Record (p_n , p_m , *"violation of* Rx_1y_1*"*)
	Record (p_n , p_m , *"violation of* Rx_2y_2*"*)

Figure 4.1: Illustration of the interface control method. *Rxy* is an identifier for the constraint over $\{x , y\}$.

The essence of any interface *control* statement in Figure 4.1 is to *record* a violation message *if* a constraint is violated. This means that any interface control statement is actually a *rule*. It is critical to note that interface control rules can by no means be predefined in any piece of software or program. The user should be free to define a constraint that better suits his design needs. Therefore, interface control rules should be arbitrarily defined outside of any software that operates on them. They can be defined in a rule based knowledge base and managed by rule engines. Such a collection of interface control rules in this thesis is called an interface control knowledge base, or simply interface knowledge base.

Equations (3.5) and (3.8) can be used to define the generic forms of interface control rules in the interface knowledge base. Each constraint that corresponds to these equations forms the left hand side (LHS) of an interface control rule; so, any interface control rule has either of the two following generic forms:

$$Port(?\,p) \wedge \ldots \wedge Attribute(t) \wedge \sim\varphi_t \longrightarrow h_t \tag{4.1}$$

or

$$Port(?\,p) \wedge Port(?\,q) \ldots \wedge Attribute(t) \wedge$$
$$Attribute(s) \wedge\sim \psi_{ts} \longrightarrow h_{ts} \tag{4.2}$$

which indicates an action h must be taken when a constraint is violated; hence, φ_t and ψ_{ts} appear in complementary forms.

One can observe that the LHS of (4.1) and (4.2) consists of two parts: the one that defines the context *C* in which the attributes are defined and the one that defines the constraint itself; so, for the sake of an analysis of the rules, they can be simplified as:

$$C(t) \wedge \sim\varphi_t \longrightarrow h_t \tag{4.3}$$

$$C(t\,,s) \wedge\sim \psi_{ts} \longrightarrow h_{ts} \tag{4.4}$$

The context specification and the attribute symbols in the above formulas can be removed by using indices that uniquely identify each attribute; so, (4.3) and (4.4) can be rewritten as $\sim\varphi_i \longrightarrow h_i$ and $\sim \psi_{ij} \longrightarrow h_{ij}$, respectively. In this way, we also do not need to distinguish a port requirement from a compatibility constraint, since ψ_{ii} can be used in place of φ_i. Therefore, an interface control rule can be simply denoted as:

$$\sim \psi_{ij} \longrightarrow h_{ij} \tag{4.5}$$

The negative rules per (4.5) specify what actions must be taken if a constraint is violated. As illustrated in Figure 4.1, an action can be the recording of violation message for a given port. A port may have several error messages if several constraints are violated. This makes it necessary to maintain an *error list* for every port in the system. A port must also have an *error flag* that alerts designers whenever its constraints are violated. The error flag indicates an *unresolved* status for the port if the error list associated with the port is not empty; otherwise, it indicates a *resolved* status for the port.

Let E_p be the list of error messages that is maintained for port p. An action h in (4.5) means adding an error message s to E_p. This can be stated as $h_{ij} = E_p \cup \{s_{ij}\}$. Since every h changes the content of E_p, the reverse of h must also be added as a rule to the interface knowledge base to delete the error message whenever the interface constraint becomes satisfied again. This means that the positive rule $\psi_{ij} \longrightarrow h'_{ij}$ must also accompany its negative counterpart in the knowledge base, where $h'_{ij} = E_P - \{s_{ij}\}$.

Let n be the total number of constraints that are defined between ports p and q. The interface control knowledge pertinent to p and q can be represented by the following set of rules:

$$\begin{array}{rcl} \sim\psi_{11} & \longrightarrow & h_{11} \\ \psi_{11} & \longrightarrow & h'_{11} \\ \vdots & & \\ \sim\psi_{ij} & \longrightarrow & h_{ij} \\ \psi_{ij} & \longrightarrow & h'_{ij} \\ \vdots & & \\ \sim\psi_{nn} & \longrightarrow & h_{nn} \\ \psi_{nn} & \longrightarrow & h'_{nn} \end{array} \tag{4.6}$$

4.2 Analysis of the interface knowledge

A knowledge base that is a collection of rules is called a rule based system. A rule based system should be verified with regard to the redundancy, subsumption, determinism, reducibility, and completeness of the knowledge it represents (Ligeza, 2006).

A rule based system K is redundant if after removing some of the rules in K, a new rule based system K' is obtained that behaves exactly as K. Evidently, the system represented by (4.6) is not redundant. In practice, the users may make the system redundant by defining identical rules. Identical rules should be disallowed by the interface control software.

Subsumption means that there is a rule that has either a more general precondition or a more specific conclusion than another similar rule. Because all constraints are independent, all interface control rules defined by (4.6) are independent; so, the only occasion in which subsumption may occur is when there are two rules such that the LHS of one rule is a *class* compatibility/requirement constraint ψ_{ij} and the LHS of the other rule is an *object* compatibility/requirement ψ_{ij}. When a class rule is defined, the object rules with the same LHS should be disallowed.

Determinism issues are mainly caused by the existence of ambiguous rules. Ambiguous rules have the same preconditions, but different conclusions. The system represented by (4.6) is not ambiguous. The interface control software automatically assigns the right hand side (RHS) of the rules itself; so, there is no way for the users to define ambiguous rules.

Reducing a rule based system means there are some rules that can be combined into a single equivalent rule. The system represented by (4.6) is not reducible, again because all individual rules are independent.

A rule based system is logically complete if the disjunction of all LHSs of its rules is a tautology. This means that every possible input condition satisfies the LHS of at least one rule. It can be observed that the interface control system represented by (4.6) is logically complete. It is composed of pairs of complementary rules; so, the disjunction of all LHSs is a tautology:

$\vDash \psi_{11} \vee\sim \psi_{11} \vee \ldots \psi_{ij} \vee\sim \psi_{ij} \ldots \vee \psi_{nn} \vee\sim \psi_{nn}$.

4.3 Interface control software architecture

In the absence of a computer based methodology, an interface control process that uses ICDs is done manually. Designers must consult ICDs to ensure what they design is compatible to the rest of the system. Computers however can make this process faster and less erroneous. The aforementioned interface control knowledge base can be managed by interface control software, which is a piece of software that has the following use cases:

U1. it controls the status of interfaces during collaborative product development.
U2. it allows easy creation and modification of interface knowledge base.

The first use case means the software architecture should facilitate communication of interface violations to designers in a collaborative environment. The second use case means the software should provide some means for the users to specify interface control rules, i.e., it should have a convenient graphical user interface (GUI). The first use case is more an architectural issue for the software whereas the second use case is more an implementation issue.

A collaborative system that is used for product development should provide mechanisms for communications among distributed organizations. In document based interface control, the means of communication can be emails, video

conferences and/or meetings. In computer aided interface control, the means of communication can be a co-design software architecture that allows synchronized modification of interface data and automatic tracking of interface violations.

In a synchronized co-design system, designers are allowed to work on the same model (Li and Qiu, 2006). Synchronization ensures changes to the model data are done safely. In a synchronous system, design tasks are scheduled by using control tokens. A designer may modify a file once he has obtained the control token for that file. When his task is complete, the control token is free to be taken by other designers.

Figure 4.2 shows a software architecture that can be used for collaborative interface control. Software that has this architecture operates on interface specifications and rules to fulfill use cases U1 and U2. User actions in Figure 4.2 are shown with dashed arrows. The rest of the arrows represent system actions. As can be seen in the figure, the architecture has a server and client. The server maintains interface specifications, interface control rules and interface ontology in a sharable format. It also synchronizes access and modifications to these files.

The user interface of the client application allows creation and modification of port attributes. Port attributes are *loaded* from the shared repository that is maintained by the interface control server, and transformed into a convenient form that can be displayed by the client application. Once designers are done with modifying port attributes, the changes must be *committed* back to the shared repository.

The client application also allows edition of interface control rules. To do this, designers only need to define interface constraints. The software automatically transforms the constraints into rules. Checking the status of interfaces requires an *inference engine*. Once the inference engine is run, all violations of interface constraints are revealed to users.

The solid arrow shown in Figure 4.2 represents a dependency between port descriptions and the ontology. It means that the software is responsible to ensure port descriptions are bound to the ontology. Binding to the ontology means to ensure the interface terminology is consistent with the framework and the vocabulary that is set by the ontology. This ensures every organization that develops a different part of the system uses the same terminology; hence, their specifications can be compared. Binding to the ontology must be done before beginning an interface specification or control session by designers.

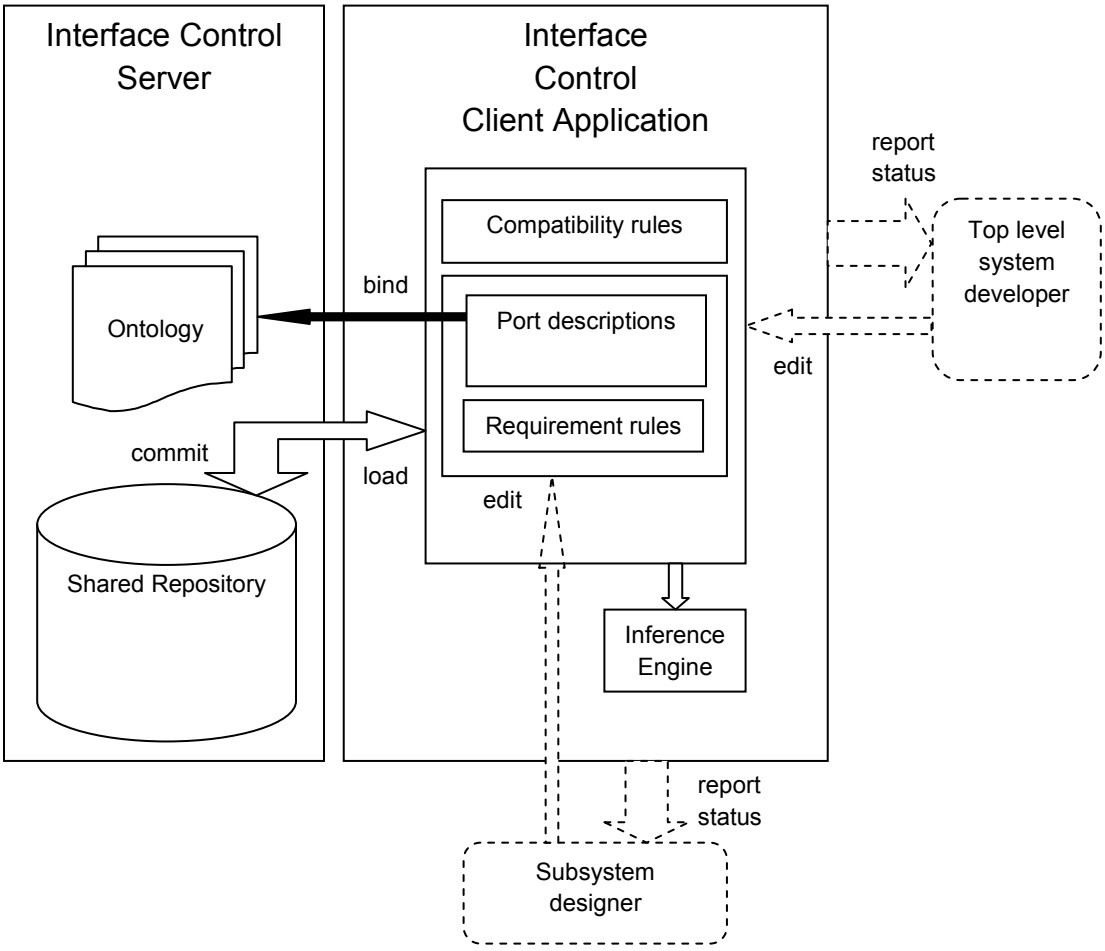


Figure 4.2: The architecture of interface control software.

Figure 4.2 also shows that the users of the interface control system can be divided into two groups with different access permissions. The first group consists of the subsystem designers. They are allowed to define and modify port descriptions and requirement rules for their own subsystems. The second group consists of the top level system developers. A top level developer has access to all descriptions and rules in the system. He/she also decides which group should have access to which subsystems. File permissions are managed by the server. Users may be given read, or write permissions to a file. The server also enforces synchronization. It ensures only one user can commit changes to any file at one time.

4.4 Summary

This chapter has presented the elements of an interface knowledge base and the control mechanism that can be used to detect port errors in the interface knowledge base. It has been shown that any interface control statement can be represented as a rule, in which the LHS is a constraint and the RHS is an action to be taken if the constraint is violated. The chapter has also presented a possible architecture for interface control software. The next chapter presents a prototype implementation of the interface control software, and shows how the current information sharing languages can be used to create computerized ICDs. The language used to share interface knowledge in this thesis is XML.

5 PROTOTYPE IMPLEMENTATION

This chapter presents a prototype implementation of the interface control software whose architecture has been defined in Figure 4.2. It also demonstrates how the consistency and compatibility checking functionalities of the software can be implemented. The consistency checking functionality ensures interface definitions are in accordance with an ontology. Compatibility checking functionality detects errors in value assignments to port attributes based on the mechanism that has been described in the previous chapter. The interface control software that is presented in this chapter is implemented in Java.

The interface control software operates on the interface information stored in a shared repository (Figure 4.2) that is accessible via a network. The interface information in the shared repository is the computer manageable version of an ICD; hence, it is called computerized ICD. The computerized ICD contains interface constraints and port specifications that are expressed by XML.

5.1 Interface and ontology definition languages

Interface definitions (specifications) in a shared repository are eventually going to be transported to different design systems that interoperate with the interface control software. As seen before, interface definitions can be created based on the binary constraints among ports attributes. The constraints have a structured format that is expressible by a computer readable language. Recall from §2.2 that port attributes and interface constraints must be defined semantically and XML is a popular language to create semantic content; hence, this thesis uses the XML language to specify port attributes and interface constraints.

As mentioned in §2.2, the structure of XML documents can be defined by XSD, which is also capable of expressing simple ontologies (Klein et al., 2000). In this thesis, the interface ontology is expressed by XSD.

Note that the types of relationships among objects that can be represented by XSD are limited. XSD is not the most expressive language that can be used to define ontologies. However, this language has been chosen to define the interface ontology in this thesis for two main reasons. First, the ontological relationships in Figure 3.7 are subtypes of *is-a* and *has-a* relationships. XSD is perfectly capable of defining these two types of relationships. Second, using XSD/XML simplifies the implementation effort to a great extent. The compilers that check the consistency of XML documents to an XSD schema are already available. Therefore, by using XSD, we do not need to implement the ontology binding functionality of the interface control software (Figure 4.2) from scratch.

Every class in an XSD file is defined by the *xsd:complexType* tag. The elements that belong to the class are indicated by the *xsd:sequence* tag. The numbers of times the elements are repeated in each sequence are marked by the tag multiplicity attributes, i.e., *maxOccurs* and *minOccurs*. Both multiplicities are set to one if they are not explicitly defined. Figure 5.1 shows the XSD file that defines the core classes of the ontology. These are the classes from which all other classes in the ontology are obtained by extension.

Figure 5.1 indicates that a port must have a name, a resolved status flag, a form, possibly several functions, and possibly several subports. It also indicates that a function can provide or require exactly one flow, and both form and flow are defined by a sequence of physical quantities. Finally, it indicates that a form has a coordinate frame, and it can be composite, i.e., having subforms. Note that in this chapter and the rest of the thesis, the term ‘verb’ in the verb-flow representation of functions is implicitly defined by the name of the function class. There is no need to explicitly define verb classes.

```
<xsd:complexType name="Port">
    <xsd:sequence>
      <xsd:element   name="name"    type="xsd:string"/>
      <xsd:element   name="resolved"    type="xsd:boolean"/>
      <xsd:element   name="form"    type="Form"/>
      <xsd:element   name="function"   type="Function"  maxOccurs="unbounded"/>
      <xsd:element   name="port"   type="Port"/   maxOccurs="unbounded"   minOccurs="0"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Function">
   <xsd:sequence>
      <xsd:element    name="provided"  type=" xsd:boolean "/>
      <xsd:element    name="flow"  type="Flow/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Flow">
   <xsd:sequence>
      <xsd:element    name="physicalQuantity"    type="PhysicalQuantity"
maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Form">
   <xsd:sequence>
      <xsd:element    name="CoordinateFrame" type="CoordinateFrame"/>
      <xsd:element    name="physicalQuantity"   type="PhysicalQuantity"
maxOccurs="unbounded"/>
      <xsd:element    name="form" type="Form"   maxOccurs="unbounded"   minOccurs="0" />
   </xsd:sequence>
</xsd:complexType>
```

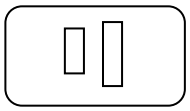
Figure 5.1: Core classes in XSD format.

Other classes in the ontology are defined by *extending* the above mentioned core classes. In this way the ontology is organized as a hierarchy, the architecture of which is defined by the core classes. This is illustrated in Figure 5.2. For example, the electrical AC energy flow and AC power coupling port can be defined as subclasses of flow and port. More specific subclasses of these latter classes can still be defined in the same way. The extended subclasses contain new attributes in addition to the ones they inherit from their super-classes. Extending and not changing the previously defined ontology prevents older XML files that are bound to the ontology from becoming corrupt. The ellipsis in Figure 5.2 is for brevity.

Port objects are defined in XML files whose structures are bound to the ontology classes. Figure 5.3 shows an example in which a power plug subsystem has an AC power coupling port. The *xsi:type* keyword specifies to which particular subclass of a core ontology class the object belongs. For example *<flow xsi:type="AC"/>* refers to an unnamed object of the electrical AC class. Objects are named only in case of subsystems and port instances because they are the ones that need to be traced. Figure 5.3 shows that *plug*1.*outlet* transmits a 110V AC energy. Its form is composed of two rectangular holes with the given dimensions. The ellipsis in Figure 5.3 is for brevity.

```
< xsd:complexType name="Transmit">
    <xsd:complexContent>
      <xsd:extension base="Function">
      …
<xsd:complexType name="Energy">
   <xsd:complexContent>
     <xsd:extension base="Flow">
       <xsd:sequence>
         <xsd:element name="power"     type="PhysicalQuantity"/>
         …
<xsd:complexType name="Signal">
   <xsd:complexContent>    <xsd:extension base="Flow">
     …
<xsd:complexType name="Material">
   <xsd:complexContent>    <xsd:extension base="Flow">
    …
 <xsd:complexType name="AC">
   <xsd:complexContent>
    <xsd:extension base="Energy">
      <xsd:sequence>
        <xsd:element name="voltage"     type="PhysicalQuantity"/>
        <xsd:element name="frequency"     type="PhysicalQuantity"/>
        …
<xsd:complexType name="Hole">  <!-can be sub-classed into Rectangular and Cylindrical…->
   <xsd:complexContent>    <xsd:extension base="Form">
     …
<xsd:complexType name="ACPowerCoupling">
   <xsd:complexContent>    <xsd:extension base="Port">
     …
```

Figure 5.2: Extension of core classes.



Power plug

```
<subsystem  xsi:type="PowerPlug">
    <name>plug1</name>
    <port  xsi:type="ACPowerCoupling"/>
        <name>plug1.outlet</name>
        <function  xsi:type="Transmit">
            <provided>true</provided>
            <flow  xsi:type="AC"/>
                <power>
                    <magnitude>60.0 </magnitude>
                    <unit>W</unit>
                </power>
                <voltage> …110.0V… </voltage>
                <frequency>…50.0H…</frequency>
        </function>
        <form>
            <form xsi:type="RectangularHole">
                <length>
                    <magnitude>1.0</magnitude>
                    <unit>cm</unit>
                <width>…0.2cm…</width>
            </form>
            <form  xsi:type="RectangularHole">
                <length>…0.5cm…</length>
                </width>…0.2cm…</width>
            </form>
        </form>
    </port>
</subsystem>
```

Figure 5.3: An example XML file representing the ports of a subsystem.

Constraints are also defined in XML files. Figure 5.4 shows a simple example in which a constraint between the voltage of *plug*1.*outlet* port and *chord*1.*inlet* port is defined. Each constraint has two blocks of operands defined between <*OP*1> and <*OP*2> tags. The relational operator between the two blocks is defined between <*ROP*> tags. The <*LCST*> and <*RCST*> tags define the two positive constants that set the lower and upper limits of the fit between two attributes per (3.3). Every constraint is identified by an *ID* tag so that it can be traced in the system if it is violated. Either of the operands in a constraint defines a fully qualified name of an attribute; hence, the constraints represented in Figure 5.4 read as:

```
<constraint>
    <ID>Constraint-C1</ID>
    <OP1>
        <class>ACPowerCoupling</class>
        <name>plug1.outlet</name>
        <function xsi:type="Transmit">
            <provided>true</provided>
            <flow xsi:type="AC"/>
                <attribute>voltage</attribute>
                <unit>V</unit>
            </flow>
        </function>
    </OP1>
    <ROP> <= </ROP>
    <OP2>
        <class>ACPowerCoupling</class>
        <name>chord1.inlet</name>
        <function xsi:type="Transmit">
            <provided<false</provided>
            <flow xsi:type="AC"/>
                <attribute>voltage</attribute>
                <unit>V</unit>
            </flow>
        </function>
    </OP2>
   <LCST>5.0</LCST>
   <RCST>5.0</RCST>
</constraint>

<requirement>
    <ID>Requirement-R1</ID>
    <Class>ACPowerCoupling</class>
    <name>plug1.outlet</name>
    <function xsi:type="Transmit">
        <flow xsi:type="AC"/>
            <attribute>voltage</attribute>
                <unit>V</unit>
                <MIN>90.0</MIN>
                <MAX>120.0<MAX>
        </flow>
     </function>
</requirement>
```

Figure 5.4: Definition of a constraint in an XML file.

*Constraint-C*1:

$ACPowerCoupling(chord1.inlet).Transmit.AC.voltage - 5.0 \leq$

$ACPowerCoupling(plug1.outlet).Transmit.AC.voltage \leq$

$ACPowerCoupling(chord1.inlet).Transmit.AC.voltage + 5.0$

*Requirement-R*1:

$90.0 \leq ACPowerCoupling(plug1.outlet).Transmit.AC.voltage \leq 120.0$

5.2 Binding port attributes to the ontology (schema)

In order to have automatic information sharing among subsystem designers, the interface control software must ensure they all define their XML files consistently, i.e., using the ports and their attributes as defined by the ontology. For example, the XML file shown in Figure 5.3 uses a legitimate vocabulary according to the ontology defined in Figure 5.2. Ensuring that port attributes are correctly chosen from the ontology is called consistency checking. This section discusses the mechanism that is used for checking the consistency of port attributes in this thesis. Since port specifications are defined by XML and the ontology is defined by XSD, checking the consistency of port attributes with regard to the ontology can be implemented by a mechanism that ensures the consistency of XML files with regard to XSD files.

The technology to ensure the consistency of XML files with regard to some given XSD files is already available and is called Java Architecture for XML Binding (JAXB) (Oracle, 2012). JAXB provides a binding compiler, called *xjc*, which derives a set of Java classes from XSD files and allows a user to use these classes to define or read XML files. For example, the hierarchy of the classes that are derived by JAXB from the schema file of Figure 5.2 is shown in Figure 5.5. In this figure, the triangular arrow represents a subclass relationship and the lozenge represents an aggregation relationship.

After binding, XML documents can be "unmarshalled" by JAXB's API; that is, a tree of content objects that portray the XML document is made based on the generated classes. The consistency of XML documents to the XML schema is automatically checked by JAXB during unmarshalling, which relaxes us from implementing this functionality from scratch. Unmarshalled objects can be accessed and modified by a Java program. It is also possible to marshall the

objects back to the XML files. Figure 5.6 shows the unmarshalled objects that correspond to the XML file of Figure 5.3.

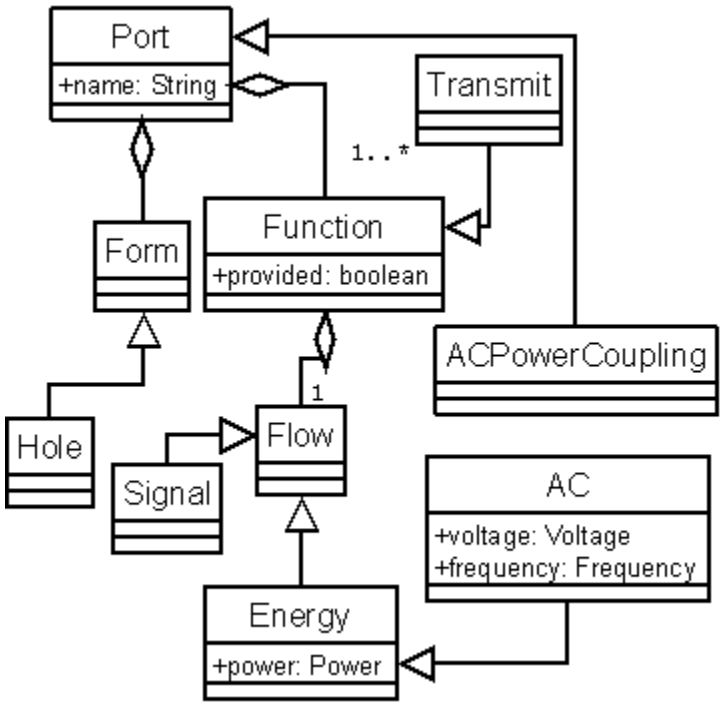


Figure 5.5: The classes that correspond to the partial XSD file in Figure 5.2.

The port attributes of different subsystems are defined in different XML files. The following lines of code briefly summarize how a subsystem XML file is unmarshalled by JAXB:

JAXBContext jc = JAXBContext.newInstance (*packageName*);
Unmarshaller u = jc.createUnmarshaller( );
Subsystem subsystem = (*Subsystem*) *u.unmarshall* (*newFile* (*XMLfilePath*));

in which the *packageName* identifies the place where Java classes that represent the ontology (XSD) are stored. By executing the above lines of code, the content of every XML file is put into a different *subsystem* object in a Java program. Every subsystem object contains all the information about its ports.

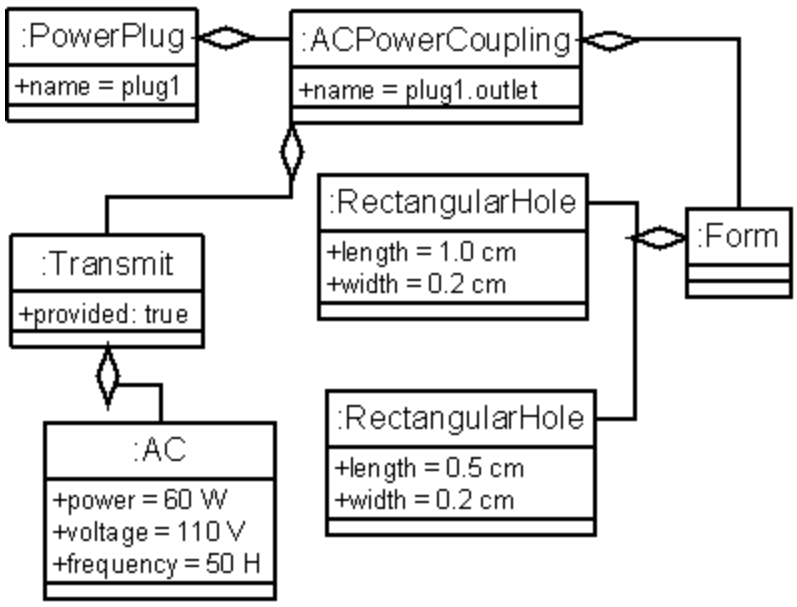


Figure 5.6: The objects that correspond to the XML file in Figure 5.3.

5.3 Editing port specifications and constraints

The unmarshalled objects from XML files can be accessed and modified via a Java program. In normal applications of JAXB, this is easy to do because the set of classes defined by the XSD file is *fixed*; so, the Java programmer already knows what classes have been defined and what methods they provide to modify their objects. A good example is a program that displays the information about a collection of books from an XML file that is bound to an XSD file. The classes in this case are all known and fixed, e.g., a book class, a book collection class, etc. Therefore, the methods to modify the objects of these classes are also known when the program is created, e.g., *a_collection.getBook*(*ISBN*) to retrieve a book from a collection, or a*_book.setISBN*() to set the *ISBN* of a book. The most important feature of this simple example is that the *entire* set of classes, i.e., *Book* and *Collection*, is known at *compile time*.

However, interface control software is a different situation. The set of classes that are defined in the schema files can be very large. More importantly, the XSD file that describes the ontology is not even fixed; it can continually evolve as new

classes are added to the ontology. This makes it impossible to preprogram all the method calls that modify the objects of these classes at compile time. Such an evolving set of classes have to be dealt with at *run time*. Using *reflection* in computer programming is a way to do this. Reflection is a mechanism by which a program can observe and modify its own structure and behavior at run time.

In an object oriented programming language such as Java, reflection allows inspection of types and names of classes, methods, attributes, etc., at runtime without knowing their types and names at compile time. It also allows instantiation of objects from classes at run time and invocation of methods on these objects. Reflection in Java is supported by a number of special classes that provide the required mechanisms for run time investigation. The most important reflection classes are *Class*, *Field*[11], and *Method*. For example, to investigate the type of a port object and enumerate its attributes at run time, one can write:

```
Class  portClass = port.getClass ( );
Field[ ]  fields = portClass.getDeclaredFields ( );
For (Field f : Fields)
        Class  fieldClass = f.getClass ( );
        Object  value = f.get (port);
```

The above statements return the type of the port and the list of the attributes it holds at run time. Once this list is available, the type of each attribute *f* and the value assigned to it can also be investigated. Modification of the attribute value is accomplished by modifying the *value* object obtained by the above method calls.

It should be mentioned here that the above explanation merely describes the outline of using Reflection in the presented implementation. Lots of technical details are skipped here that would consume a lot of space, but add little to the understanding of the functionality of the software. For example, one limitation of

[11] Attributes are called instance fields in Java.

the *getDeclaredFields*( ) method is that it only enumerates direct fields of an object. It does not enumerate the object's inherited fields from its super classes. Inherited fields have to be enumerated by recursive method calls and so on.

5.4 Inference engine

An inference engine operates on the interface control knowledge base and checks the violation of interface control rules. A business rule engine can be used for this purpose. The rule engine used in this implementation is Jess, the rule engine for the Java platform (Friedman-Hill, 2003). Using Jess enables us to write a Java program that has the capacity to reason about the knowledge that is supplied to it. Jess is a small and fast rule engine that can fully operate in a Java program.

Jess always operates on a collection of knowledge items called facts. Every fact has a template. Facts and templates are analogous to objects and classes in Java. Every template has a name and a set of data slots, which are also analogous to class names and class attributes in Java. Facts are instantiated from templates by assigning values to the slots; likewise, Java objects are instantiated from Java classes by assigning values to class attributes.

Every fact in Jess is represented as a list. For example, a fact that represents a flow of AC energy is represented as follows:

(*AC* (*power* 60.0W) (*voltage* 110.0V) (*frequency* 50.0H))

Facts can be created entirely by Jess, or derived from the objects in a Java program. The latter are called shadow facts to emphasize that they are linked to objects from the outside of Jess. To create shadow facts, first, Jess templates are associated with Java classes. For example, an *AC* flow template in Jess can be associated with an *AC* class in a Java program as follows:

(*deftemplate AC* (*declare* (*from-class*(*AC*)))

Next, Java objects are added to Jess' working memory to create shadow facts. This is done by instantiating a Jess rule engine inside a Java program and calling *engine.add* (*object*) method to add an individual object or *engine.addAll* (*objects*) to add a set of objects to Jess' memory.

Jess rules are declaratively defined in the form of LHS => RHS, in which LHS stands for left hand side and RHS stands for right hand side. A declarative rule engine is different from a procedural one in that the rules in the former can be arbitrarily executed when their preconditions are satisfied, whereas the rules in the latter are executed in the same order that is defined by the programmer. This makes Jess orders of magnitude faster than a procedural rule based system (Friedman-Hill, 2003). It should be noted that the sequence of rules in (4.6) is immaterial.

```
(defrule Requirement-1
      (Port (name /plug1.outlet/) (OBJECT ?port))
      (Function (flow ?flow) (OBJECT ?function))
      (ErrorLogger (OBJECT ?logger)
      (test (and
         ((?port getFunction) contains ?function)
         (= ((?function getClass) getName) "Transmit")
         (= ((?flow getClass) getName) "AC")))
      (test (and
         (>= ((?flow getVoltage) getMagnitude) 90.0)
         (<= ((?flow getVoltage) getMagnitude) 120.0)))
   =>
      ((?logger getErrorList ?port) remove "Requirement-1")
      (?port setResolved (?logger isResolved ?port)))
```

Figure 5.7: A Jess rule.

Jess considers the LHS of any rule as a pattern. The satisfiability of LHS is checked by pattern matching. Figure 5.7 shows an example Jess rule. The pattern to match in this example is a port that is named as *plug1.outlet*. The test conditions in this rule check whether the function and the flow have the right types, and whether the magnitude of the *voltage* of the *AC* flow is between 90.0

and 120.0. The tests for the units of measures are ignored for brevity. The slashes / / are used to find an exact match between two strings. The *OBJECT* is a Jess artifact for storing a pointer to the object whose pattern is matched so that it can be used in other statements within the rule. When the LHS of this rule is satisfied, an error message is removed from the port's error list.

In the example shown in Figure 5.7, the name of the rule is the same as the name of the requirement constraint it checks; so, the satisfiability of the LHS of the above rule means the *Requirement-1* constraint is satisfied. Therefore, in RHS, the violation message that has been added as a result of the violation of *Requirement-1* must be removed. All violation messages are stored in the *logger* object. The logger object contains a hash map that maintains all error messages for all ports in the system. A hash map is a fast way of randomly accessing data stored in a memory. A hash map contains key-value pairs in which a value is returned by providing a key.

The hash map that belongs to the logger object of a system is defined as $HashMap < Object\ ,\ List < String > >$. The object keys in this hash map are chosen from the collection of all ports in the system. By using a port as a key to this hash map, one can access the list of all error messages associated with that port, which contains the names of the interface constraints that have been violated, e.g., *Requirement-1*. In this way, the violated constraint can be traced in the system. Back to the example shown in Figure 5.7, the RHS of the rule uses *?port* key to get the error list of the port, and removes *Requirement-1* message from the list. The RHS of the rule also decides the resolved status of the port by invoking the *isResolved* method on the logger object. The *isResolved* method returns true if the error list that is associated with the port is empty.

The above mentioned example illustrates a positive rule. As discussed in §4.1, every positive rule must have a negative counterpart. This means that in the previous example, the *Requirement-1-negative* rule must be defined too. The negative rule can be automatically derived from the positive one. The negative

counterpart of *Requirement_1* only differs in the test conditions on the power and the action defined in the RHS, hence

```
(defrule  Requirement-1-negative
   (…same as before…)
   (test  (not  (and
      (>= ((?flow  getVoltage)  getMagnitude)  90.0)
      (<= (?flow  getVoltage)  getMagnitude)  120.0))))
   =>
   ((?logger  getErrorList  ?port)  put  "Requirement-1")
   (?port  setResolved  (?logger  isResolved  ?port)))
```

which puts *Requirement-1* in the error list of the port.

5.5 Summary

Putting everything together, the functionality of the prototype interface control software can be summarized by Figure 5.8. The repository side of the architecture contains the XSD file that defines the ontology, the XML files that define port specifications, and the XML files that define constraints. The binding compiler of JAXB generates classes that correspond to the XSD file while JAXB API transforms subsystem XML files into the objects that are instances of these classes. The constraint XML files are transformed into Jess rules by the *XML2Jess* translator. Jess rules are applied to the shadow facts, i.e., port objects, to check the status of ports. The software shows whether a port has unresolved constraints. If so, the software shows which port constraints have been violated.

Figure 5.8 captures the core functionality of the interface control software, but it does not describe its user interface. Without a convenient user interface it may be hard to use the software in practical situations. Obviously, the users of the interface control software are better off with a more user friendly representation of the rules than either of the XML or the Jess representation. These sorts of issues

should be taken care of in the design of the software's user interface. In this research, a graphical user interface (GUI) is implemented that allows designers to readily edit port attributes and interface constraints. The form and functionality of the GUI is exemplified in the next chapter.

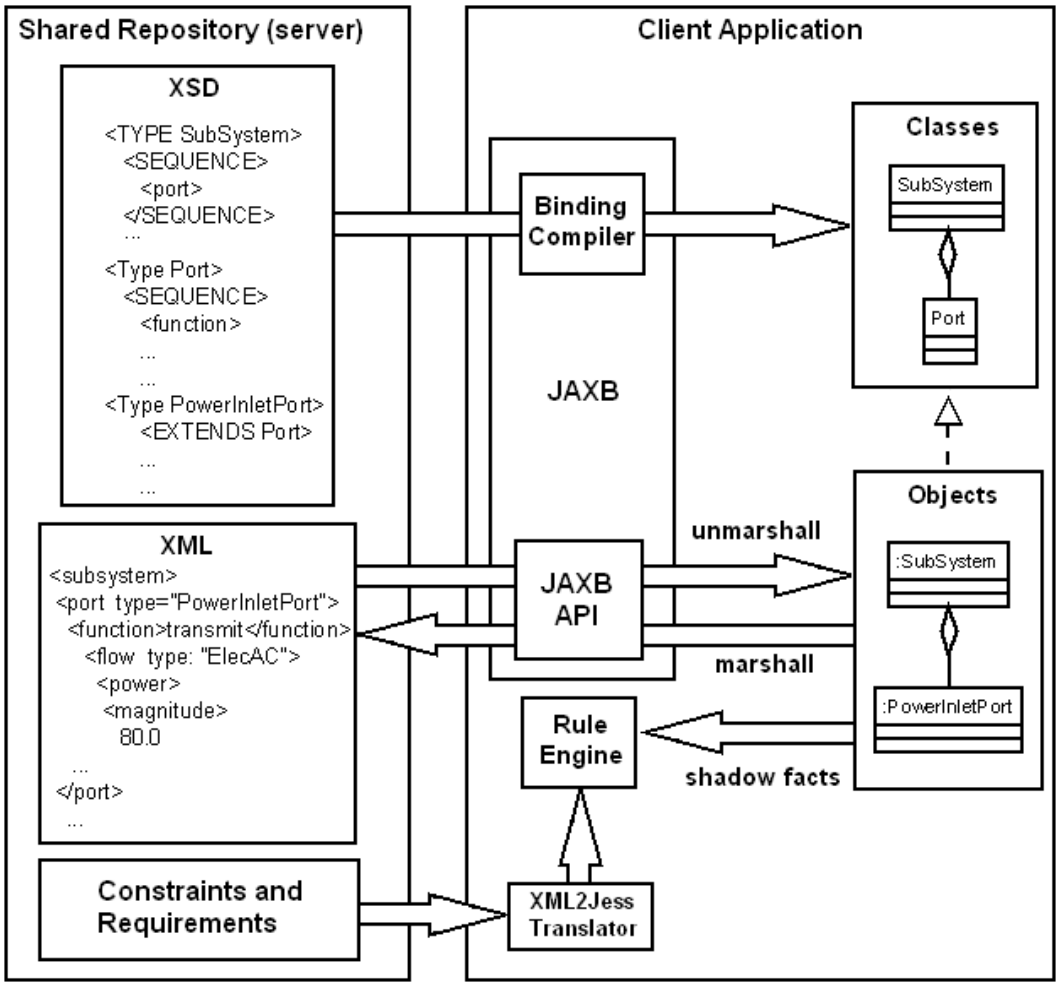


Figure 5.8: The architecture of the prototype interface control software. The XSD file is shown in pseudo form.

6 EXAMPLES

This chapter presents a piece of software that can check the consistency of computerized ICDs. The chapter illustrates how binary interface constraints can be defined by the software's GUI. Included are the functionality of the implemented software, its GUI, and its capability of doing computer aided interface control within the boundaries of this thesis.

The difference that the port to port interface model and the proposed software architecture make in product development is that they provide a connectivity model that can automatically tell designers whether they are defining interface attributes correctly. The real benefit of using such a system is evident when there are a large number of interfaces that need to be checked. Interface constraints can be created once, and the values of port attributes can be changed as wished since the software guarantees to report incorrect assignments. Manual ICDs on the other hand do not provide any connectivity between two subsystems. They are isolated documents that need to be consulted every time a change in the design of interfaces on one of the subsystems is requested. Finally, formal and structured interface definitions can pave the way to include interface specifications into CAD/PDM systems.

6.1 Flight simulator example

Figure 6.1 shows a simplified representation of a flight simulator with a few components shown as a SysML internal block diagram. *Cabinet*1 consists of an off-board computer (*simComp*) that runs a piece of simulation software. C*abinet*2 is onboard, and contains many subsystems (not shown) that provide the computing infrastructure to generate the required signal for the panel in the *cockpit*. The other subsystems of interest in this product are the two power supplies shown in the figure. The ports in Figure 6.1 are named *p*0, …, *p*5 and *s*0,

…, *s*2. These ports belong to different subsystems. In the dotted notation, the names of these ports should be read with regard to the names of their subsystems, for example, port *p*2 of subsystem *hub*1 is read as *hub*1.*p*2.

This example is intended to demonstrate the GUI and the functionality of the implemented software. It shows how port specifications and constraints are defined in the software by taking as examples the power coupling port *hub*1.*p*2 and the signal coupling port *hub*1.*p*3 of Figure 6.1. The figure also shows the flows through these ports. For example, there is an *AC* flow into *hub*1.*p*2 and there is a digital signal flow from *simComp.p*1 to *hub*1.*p*3. A section of the ontology classes that is relevant to this example is shown in Figure 6.2. For brevity, the form attributes of the ports are mostly ignored in this example.

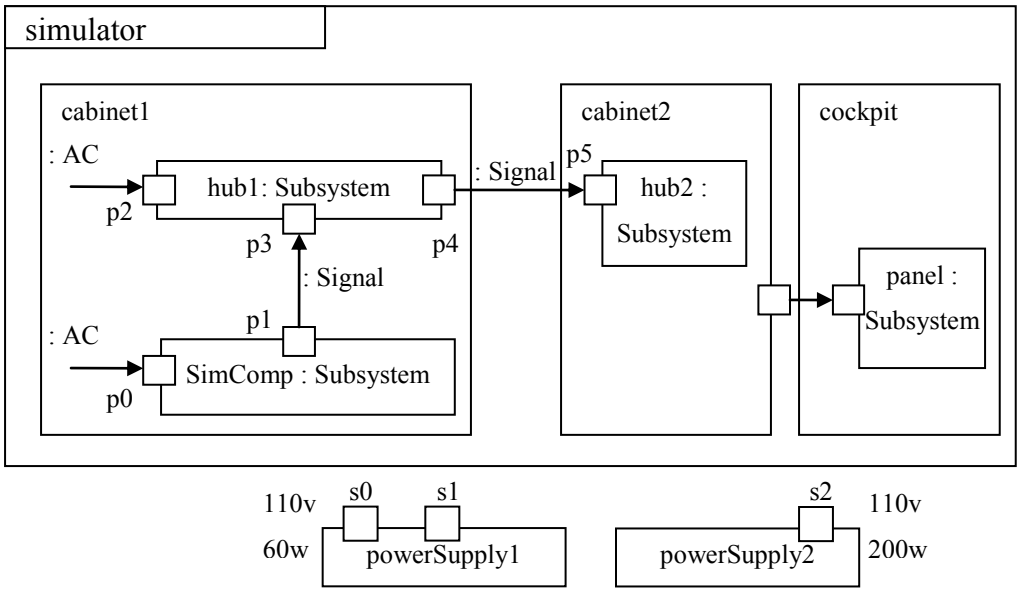


Figure 6.1: Simplified internal block diagram representation of a flight simulator.

As can be seen in Figure 6.2, the *SignalCoupling* class is defined as an aggregation of signal *Pins*. An easy interface definition case happens as shown in Figure 6.2 when a port is standard. For example, it may be possible to fully define a standard signal coupling port by giving its standard label, its number of pins,

and the male/female labeling of its form (not shown in the figure). Non-standard ports should be fully specified in terms of their form and function.

The mate attribute of a port shown in Figure 6.2 is a string that contains the name of the port to which it is connected. This attribute is significant when class compatibility constraints are defined (§3.1). This way of constraint representation reduces the effort of otherwise individually specifying the same constraints between every pair of port objects that belong to two specific classes. The class compatibility constraints have a scope, which is the flight simulator system in this example.

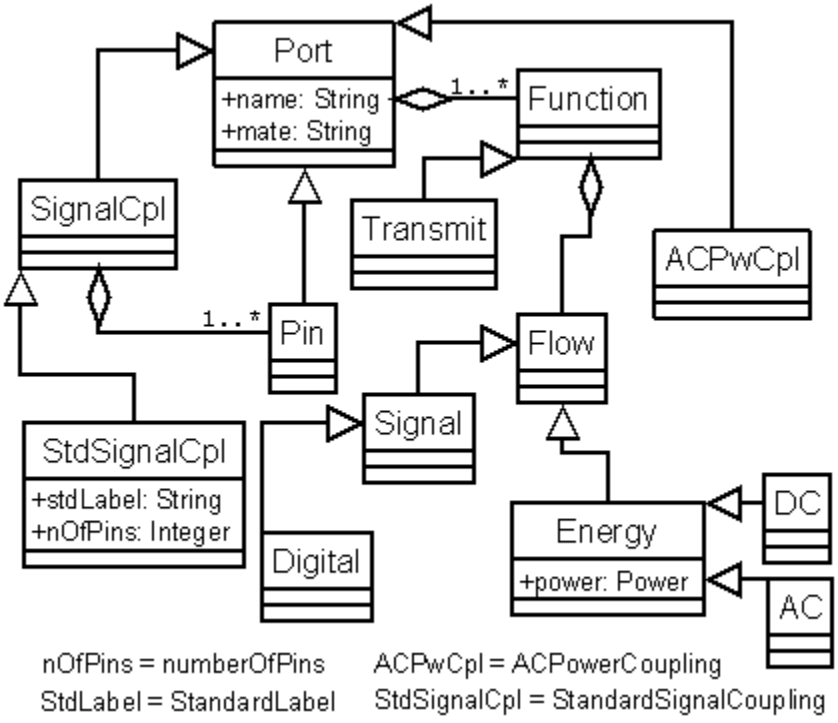


Figure 6.2: The class hierarchy that represents the section of the ontology used in the flight simulator example.

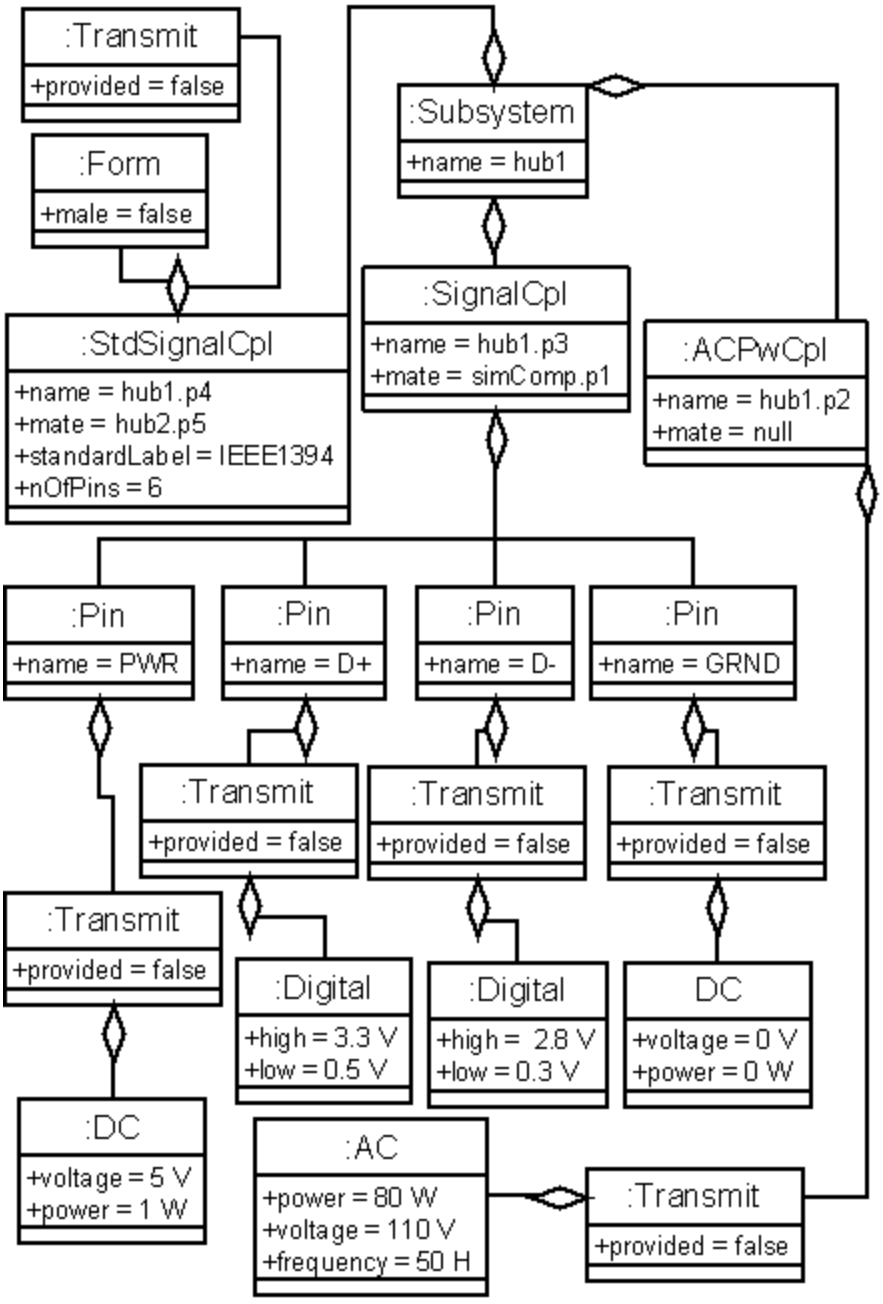


Figure 6.3: Content objects corresponding to the *hub1* subsystem specification.

The initial state of the *hub*1 subsystem is shown in Figure 6.3. The figures shows the port objects that are unmarshalled from *hub1*'s XML file and loaded into the interface control application. The *PWR* pin of *hub*1.*p*3 requires a constant voltage of 5V. The *GRND* pin is a reference voltage pin for the other three pins in

*hub*1.*p*3. The pins *D*+ and *D*- are used to transmit a digital signal. Port *hub*1.*p*4 is a standard IEEE1394 signal coupling, which is represented with the standard label, the number of pins, and a very abstract representation of its form, i.e., the male form. Port *hub*1.*p*2 is an AC power coupling port.

According to Figure 6.3, *hub*1.*p*3 is mated to *simComp.p*1. They are aggregate ports that contain pin subports. In *hub*1.*p*3, assume the pins have the arrangement as shown in Figure 6.4. The pins in *hub*1.*p*3 are ordered with regard to the partitioning method explained in §3.3. If the polar axis of *hub*1.*p*3 is placed in its centroid and directed toward *hub1.p3.D*-, the ordering of its subports becomes *D*-, *D*+, *GRND* and *PWR*. Now, if the polar axis of *simComp.p*1 is placed in its centroid and directed toward *simComp.p*1.*D*+, the ordering of the subports of *simComp.p*1 has to be *D*+, *D*-, *PWR* and *GRND* so that the constraint *hub*1.*p*3.*order* = *simComp.p*1.*order* can be satisfied, as illustrated in Figure 6.4. Note that the ports in this figure are viewed from the outside of the subsystems; so they correctly coincide when the page that contains the figure is folded along its vertical middle line.

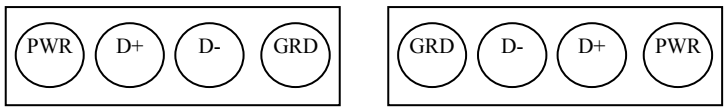


Figure 6.4: Order matching between *hub*1.*p*3 and *simComp.p*1.

A snapshot of the application's graphical user interface (GUI) is shown in Figure 6.5. This GUI implementation is based on Java Reflection classes as mentioned in §5.3. The GUI has four distinct areas: the *Project* tree, the subsystem *Navigator* tree, the *Editor* area and the *Output* area.

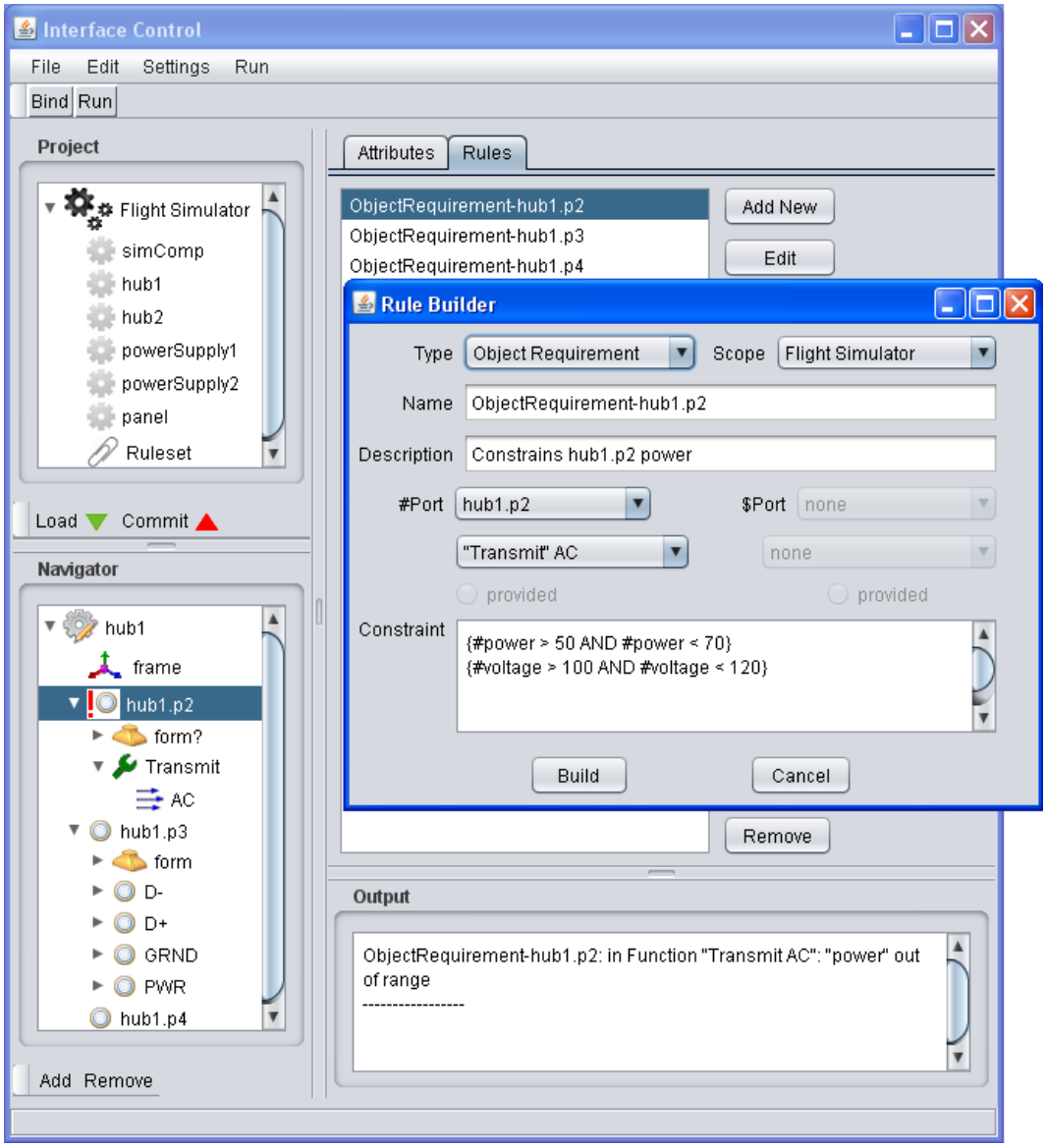


Figure 6.5: The graphical user interface of the prototype interface control application.

The *Project* tree shows all the content that is created for a system in a shared repository on the server. Before starting a project, the user must bind the overall system to an ontology by clicking on the *bind* button on the top of the *Project* tree. There is a node for each subsystem XML file in this tree. The *Project* tree also contains a *Ruleset* node that represents all the compatibility rules created for this project. By selecting a subsystem node from the *Project* tree and clicking on

the *Load* button, the user can load the contents of that subsystem from the server into the *Navigator* tree for modification if he/she has the write permission. When done with the modification, the user can click on the *Commit* button to save changes to the files on the server.

The *Navigator* tree in Figure 6.5 shows the contents of a subsystem in terms of its ports. For example, the figure shows that subsystem *hub*1 uses three ports *hub*1.*p*2, *hub*1.*p*3 and *hub*1.*p*4 for interaction. The signal coupling port *hub*1.*p*3 is an aggregate port and contains *D*-, *D*+, *GRND*, and *PWR* contact pins. The *hub*1.*p2* port in the *Navigator* tree is expanded to show its content. The question mark next to the *form* node means the form of this port has not been defined yet. This port has a *Transmit AC* function.

The editor area in Figure 6.5 contains two tabs for editing the attributes and rules in a project. The *Rules* tab shows either a list of port requirements or a list of compatibility constraints depending if a node from the *Navigator* tree is selected or the *Ruleset* node from the *Project* tree is selected. Selecting a rule from the *Rules* tab and then clicking on the *Edit* button opens the *Rule Builder* window as shown in Figure 6.5. The figure shows the content of *Requirement-hub1.p2* in the *Rule Builder* window. Every rule has a name and an optional description. The ports for which the rule is built are selected from the *#Port* and *$Port* combo boxes. In a requirement rule, the *$Port* combo box is not needed and disabled.

As shown in Figure 6.5, the *Rule builder* window requires that the type and scope of each rule be selected from the *Type* and *Scope* combo boxes. The scope of all the rules in this example is the *Flight Simulator* system. The type combo box allows selection of one of the following four types: object requirement, class requirement, object compatibility, and class compatibility. The example shown in Figure 6.5 illustrates an object requirement for *hub*1.*p*2.

The next row in the *Rule Builder* as shown in Figure 6.5 requires the user to select an available function or form for which he/she wants to build a rule, e.g.,

Transmit AC in the figure. Finally, the *Constraint* area at the button of the *Rule Builder* window allows the user to define the constraints on the attributes, i.e., φ_t or ψ_{ts} per (3.5) and (3.8), which constitute the LHS of the rules.

A constraint on a single attribute or a pair of commensurable attributes is inserted between curly brackets as shown in Figure 6.5. Any attribute that refers to the *#Port* must be prefixed by '*#*' symbol, and any attribute that refers to *$Port* must be prefixed by '*$*' symbol. The *$Port* is enabled for defining compatibility constraints. All constraints that apply to the same form or flow must be defined at once and be separated by curly brackets as illustrated in Figure 6.5. The system does not allow separate definitions of constraints on the same form or flow to prevent occurrence of identical or subsumptive rules. This is a preventive measure given that Jess itself completely detects and removes such rules, although it is not explicit to the user; so, identical and subsumptive rules cannot occur in the system.

The example in Figure 6.5 illustrates the constraint on the power and the voltage of *Transmit AC* function of *hub*1.*p*2. The "<" and "≤" symbols are used interchangeably and both represent "≤".

The user defined constraints are then translated into Jess rules. The constraints constitute the LHS part of Jess rules. The software automatically creates the RHS of the rules. When user defined constraints are translated into Jess, they are also broken down to atomic rules on individual attributes. For example, the requirement defined in Figure 6.5 is broken down into two Jess rules that separately represent the constraints on the power and the voltage. When this is done, the error messages are also specifically adjusted to include the attribute name in addition to the constraint name.

The software needs both positive and negative rules to operate properly (§5.4), but this does not concern the users. The software itself automatically creates both positive and negative rules from the user defined constraints. After the rules are

translated into Jess, the user can run a status check by clicking on the *Run* button on the main toolbar (Figure 6.5).

During a status check, the *Output* area of the GUI shows whether there are any errors within the system. The errors can be viewed by clicking on a port node (or any of its sub-nodes) in the *Navigator* tree. When a port node is clicked, the *Output* area shows the error messages associated with that port. For example, Figure 6.5 shows that the power value of *hub*1.*p*2 is out of range. This is expected because the power of *hub*1.*p*2 was initially set to 80W as in Figure 6.3. A red exclamation mark also appears next to a port node icon in the *Navigator* tree if the port is in error state.

After seeing an error, the users can refer to the *Attribute* editor area to fix it. This is illustrated in Figure 6.6. The Attribute area shows the *editable* content of each selected node from the *Navigator* tree. By clicking on the *AC* flow of *hub*1.*p*2 in the *Navigator* tree, the *Attribute* tab shows its voltage, power and frequency. The user can change the attribute values or units in here. A question mark is shown in a table cell to notify the user in the case where a value is undefined. In the example shown in Figure 6.6, the user changes the power value to 60W, and then runs the system again. This time an 'OK' message is shown in the *Output* area to indicate that *hub*1.*p*2 does not have any errors since the power of 60W now satisfies the *ObjectRequirement-hub1.p2*.

Let us now illustrate how a class compatibility rule is defined and checked by the software. A rule that is defined between two port classes is applicable to every pair of objects that belong to these classes. Class compatibility rules are derived from user defined class compatibility constraints. An example of a class compatibility constraint definition is shown in Figure 6.7. This constraint is applicable to all *AC* power couplings within the scope of the flight simulator system. It signifies that any two objects of the *ACPowerCoupling* class that satisfy the specified constraints between their powers and voltages are compatible if one of them requires the *AC* flow and the other provides it.

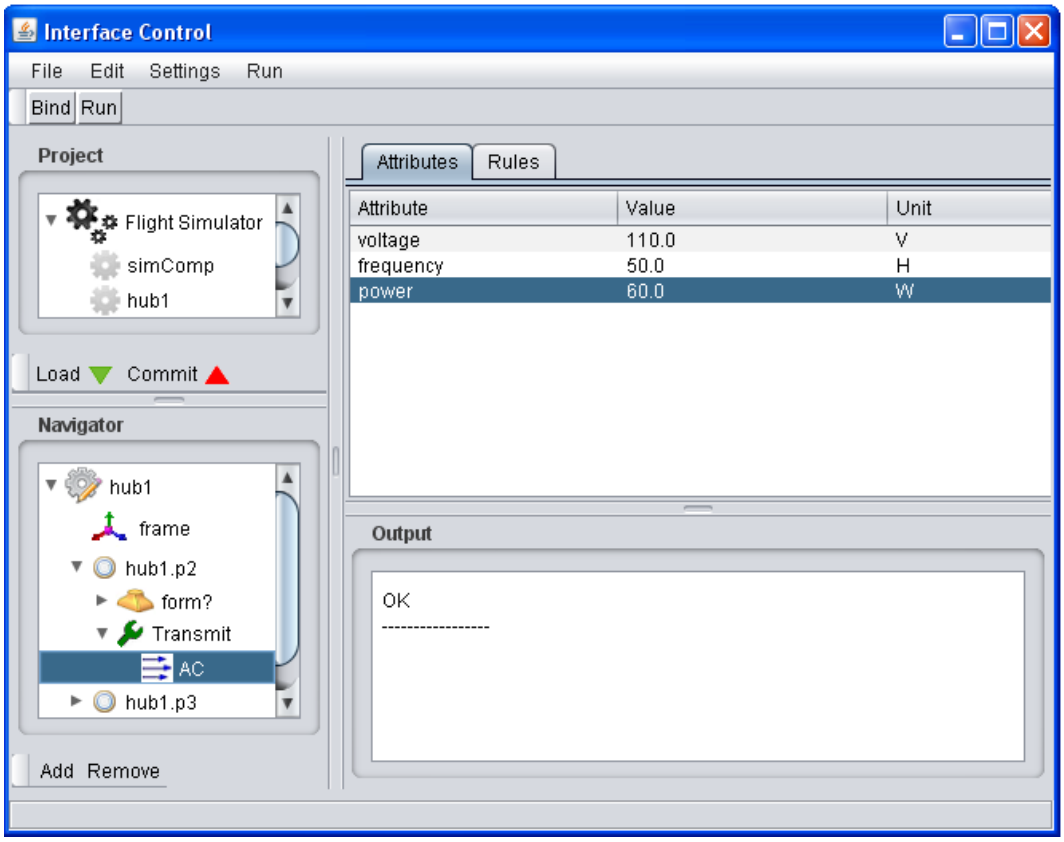


Figure 6.6: Editing an attribute value.

Defining a class compatibility constraint is a useful and time saving way of representing a constraint that uniformly applies to all objects of two classes in a system's scope. The items available in *#Port* and *$Port* combo boxes change depending on whether *Object Compatibility* or *Class Compatibility* is selected from the type combo box of the rule builder window (Figure 6.7). In the former case the *#Port* and *$Port* combo boxes show the list of objects in the system's scope whereas in the latter case they show the list of classes. In the case of a class compatibility constraint, the user must also indicate which one of the ports is the provider of the function by selecting either of the radio buttons. These radio buttons are disabled in the case of object rules because the *provided* attribute is explicitly defined for port functions.

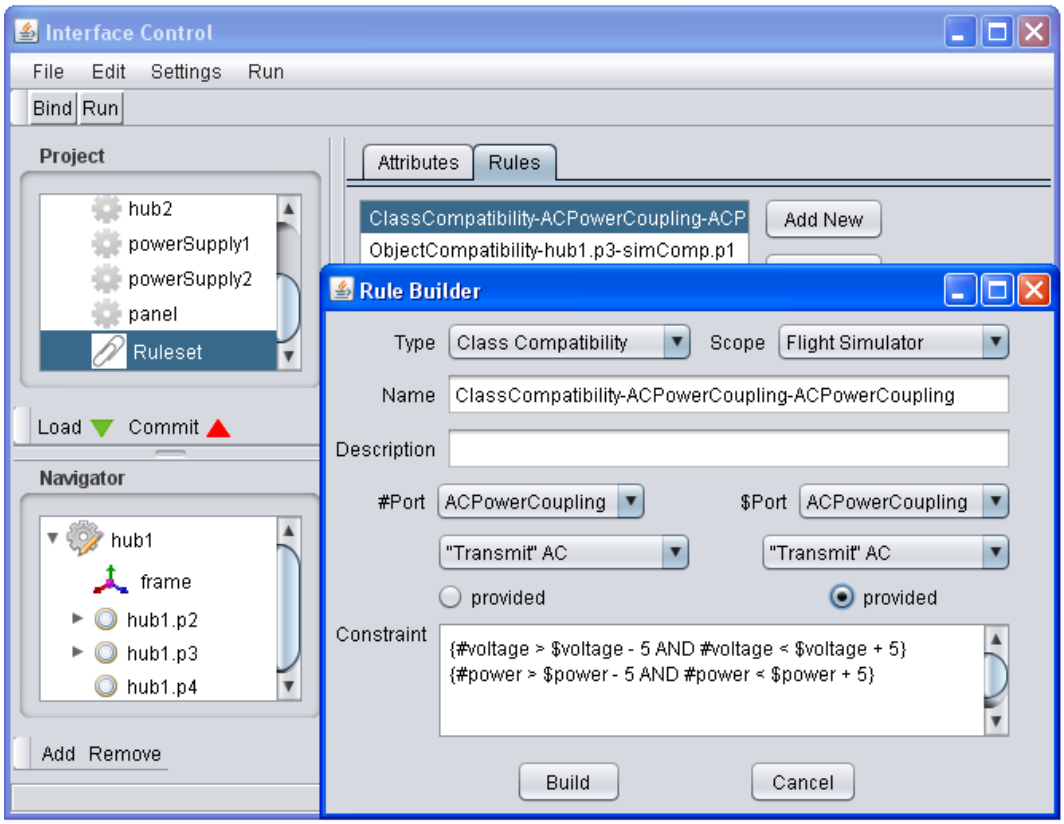


Figure 6.7: A class compatibility constraint between two AC power couplings.

Similar to requirement constraints, one must edit all the compatibility constraints on the same form or flow at once, as illustrated in Figure 6.7. The software disallows partially defining compatibility constraints between the same pairs of forms or flows in separate places. Moreover, when a compatibility constraint between two classes is defined, the software disallows defining the same constraint between the objects of these classes.

A class compatibility constraint can be used together with the mate of a port to run a status check. For example, the class compatibility constraint in Figure 6.7 becomes applicable to *hub*1.*p*2 if it is mated to another power coupling port. Note from Figure 6.3 that initially *hub*1.*p*2 does not have a mate. To mate this port to another port, the user can select *hub*1.*p*2 node from the *Navigator* tree and then set its mate to *powerSupply*2.*s*2, as illustrated in Figure 6.8.

Once the mate of *hub*1.*p*2 is set, another status check is run that results in a new error as shown in Figure 6.8. The *Output* area indicates that the power of *hub*1.*p*2 is in conflict with *powerSupply*2.*s*2. This happened because the class compatibility constraint between these two ports requires that the power of *hub*1.*p*2 be within the ±5W range of any mating port's provided power, including *powerSupply*2.*s*2. Changing *hub*1.*p*2's mate to *powerSupply*1.*s*0 (its power and voltage are shown in Figure 6.1) can resolve this issue since it has the correct matching power.

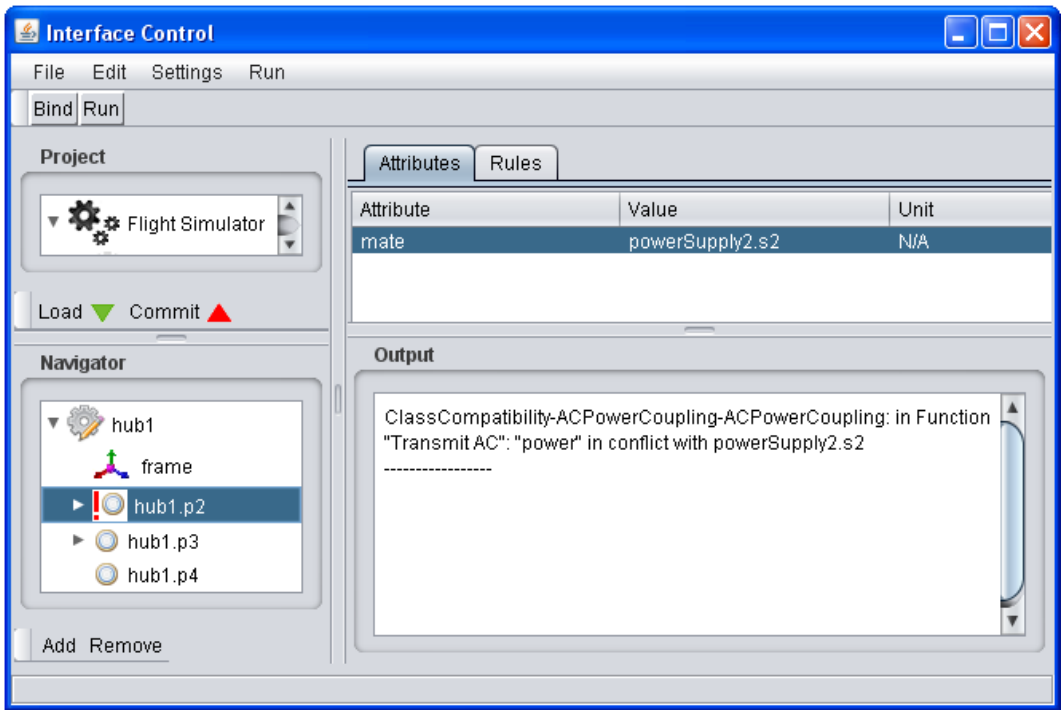


Figure 6.8: Setting the mate of a port.

6.2 Pylon example

The second example in this chapter demonstrates how the interface control process that is proposed in this thesis can be methodologically applied to a product development project. Figure 6.9 shows a few of the interfaces beween the fuselage, pylon and engine in an aircraft that is taken from a student project in a Master of Aerospace program under CAMAQ (Fortin et al., 2006). In the project,

the students are supposed to design a pylon so that a given jet engine can be integrated with an aircraft fuselage. One of the important pieces of equipment to design in this project is the mounting device that connects the engine to fuselage through the pylon.

It should be noted that in practice, to completely define interfaces in the pylon example, CAD and assembly models need to be drawn as a part of interface definitions, but that is not the concern of this example. This example is only intended to demonstrate the methodological application of the interface control process described in this thesis to a given project: defining the ontology, identifying ports, identifying forms and verb-flows for the ports, and defining the constraints between port attributes.

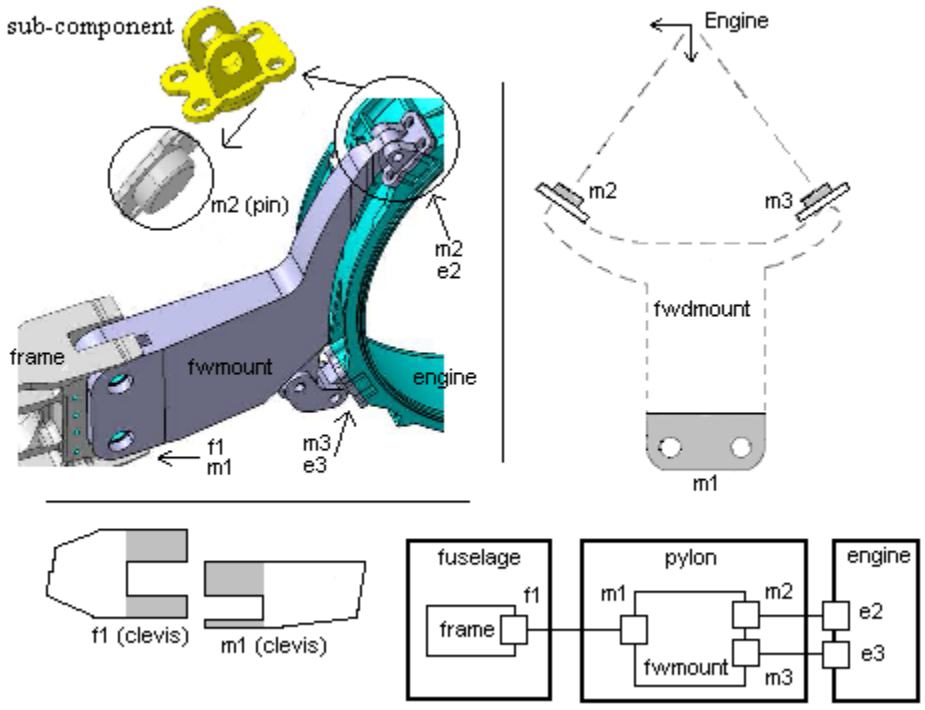


Figure 6.9: Interfaces between the pylon, engine and fuselage.

To design the mounts, students are provided with ICDs that describe the forms of the connections on the engine and the fuselage sides of these mounts as well as

the specification of the engine's axial thrust force. Let us take *fwmount* (forward engine mount) as an example component for which the interface knowledge is going to be created. One can identify three areas in Figure 6.9 that capture the interaction of this component with the rest of the system. The ports of interest in this example are named *m*1, *m*2, *m*3, *e*2, *e*3, *f*1. Let us ignore the rest of ports in this example to simplify illustration.

Ports *m*2 and *m*3 are two mechanical pins that have identical forms. The subcomponent shown in the upper left part of the figure contains port *m*2. A slightly different subcomponent is used in *fwmount* that contains *m*3. The two hole ports of the engine that receive these ports are *e2* and *e3*, which are hidden below *m2* and *m3*.

The shaded areas in the upper right side of Figure 6.9 highlight ports *m*1, *m*2 and *m*3. The shaded areas in the bottom left side of the figure highlight ports *m*1 and *f*1. The bottom right side of the figure shows connectivity of these ports in a SysML diagram. The complete pylon and engine equipments are not shown in the figure.

The semantics of the ports involved in this example are captured by the partial ontology shown in Figure 6.10. The engine thrust force in this example is captured by the mechanical energy flow through the transmit function of *e*2 to *m*2 as well as *e*3 to *m*3. Mechanical energy also flows between two clevis ports *m*1 to *f*1. The port requirement and compatibility constraints in this example can be easily specified based on the form and function attributes that are defined in the ontology, e.g., the constraint between the wall thickness of *m*1 and the slot width of *f*1.

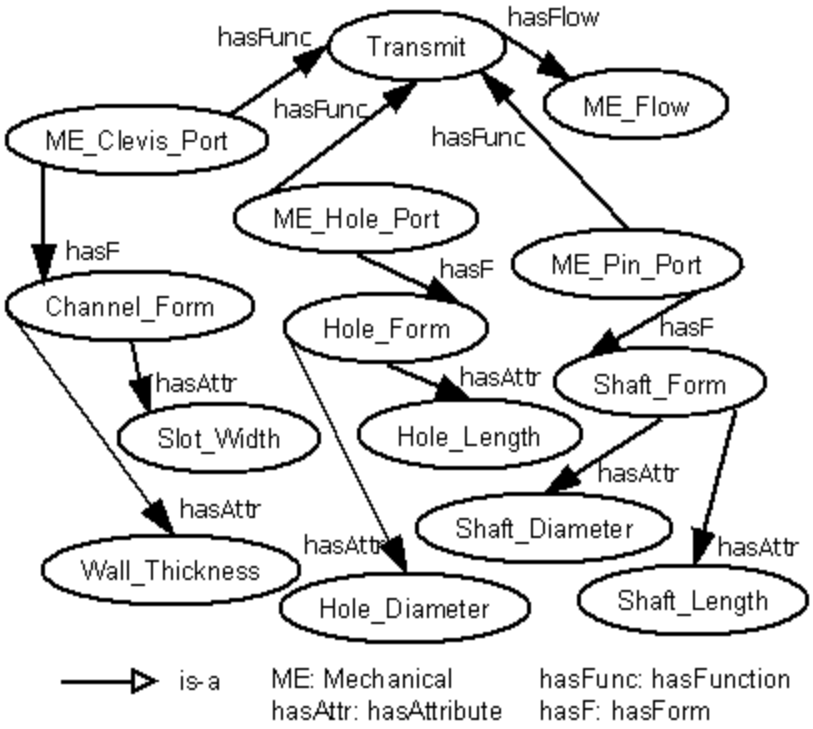


Figure 6.10: A partial ontology that captures the semantics of the pylon example.

7 CONCLUDING REMARKS

Despite its importance, no research has been published on a comprehensive method for checking the completeness, correctness, connectivity and consistency of ICDs by computers. Managing ICDs by computers can speed up collaborative design activities that use ICDs. This thesis is intended to bring some academic insights into the issue of computer aided interface control in product development.

ICDs are particularly important artifacts in the development of multidisciplinary and distributed systems. In such systems it is not possible to capture all interactions of subsystems in a domain specific CAD model. When it comes to define the interfaces between these subsystems, we are still relying on manual documentation. This thesis is the first attempt to shift from document based interface control practices to computer based interface control practices.

The thesis proposes a computer aided methodology and software architecture to manage the information contained in ICDs. The methodology uses port based representation of interfaces to capture the interactions of subsystems. Port based representations fit well with applying the principle of encapsulation in subsystem design. An encapsulated subsystem has hidden internal structure. ICDs should exclusively define the interactions among the boundaries of subsystems irrespective of what the internal structures of these subsystems are.

The principle of encapsulation provides an objective criterion to create sufficiently detailed ICDs, as proposed in this thesis. Having sufficient amount of detail in ICDs has been a vague idea in the communities that use ICDs. By applying the principle of encapsulation to the definition of interfaces, the specification of interfaces is separated from that of components, making ICDs sufficiently detailed.

The thesis also discusses the architecture and the main functionalities of interface control software. The thesis identifies two main functionalities of such software: checking the completeness and consistency of interface definitions, and checking the connectivity and compatibility of interface constraints.

Checking the consistency of interface definitions is crucially important if the interface information is shared among different organizations, which is a very likely situation for most products. Each organization may use a different terminology to define interfaces, which cannot be understood by another organization. The interface information from all such diverse sources is going to be managed by the interface control software; hence, the software must have a mechanism to ensure they have been defined consistently. This is accomplished by using an ontology that defines the semantics of port attributes that constitute interface definitions. Some of the artifacts of the semantic web technology such as XML and XSD can be used for this purpose.

Checking the compatibility of interfaces is also an important issue to prevent design errors. In the absence of a computer management tool, the designers must manually read ICDs to ensure their subsystems are designed according to the interface agreements. Having a piece of software that automatically checks the compatibility of interfaces and precisely shows what errors happened in the system certainly helps to speed up the design process.

7.1 Contributions

The contributions of this thesis can be summarized with regard to its objectives that have been defined in §1.5. This thesis has proposed a computer readable language, software architecture and a methodology that can be used to:

1. *consistently* define interfaces,
2. identify missing interface information, i.e., *completeness*,
3. automatically check the *compatibility* of interfaces,

4. *communicate* violations of interface compatibilities to all stakeholders and precisely track the violation.

The above objectives have been accomplished by using the following ingredients:

1. A consistency checking mechanism: This is done by using an ontology to explicitly specify the semantics of interfaces. The ontology provides a common vocabulary for interface definitions; so, it is the basis for consistency checking by interface control software. The software ensures all collaborating agents commit to the terms in the ontology.
2. A mechanism to find missing interface information: This is also done with the aid of the ontology. The ontology precisely defines the set of attributes for a port. If any of these attributes is missing in the interface definition, the software indicates it by showing a question mark in the place of the missing attribute.
3. A rule based system for interface knowledge representation: This is called interface control knowledge base. A control mechanism is proposed that operates on the interface control knowledge base and finds incompatibilities between the value assignments of port attributes. The interface control knowledge base is a collection of interface control rules. The rules are automatically derived from requirement and compatibility constraints that are defined by the user. The constraints are defined according to a formal model.
4. A software architecture that allows communication of violated interfaces: The software reports the erroneous ports, the constraints that are violated, and the names of the attributes that are involved in violations.

To demonstrate the above contributions, the thesis presented a prototype implementation of the interface control software that can check the consistency of interface definitions and report violations. This has been done by implementing a piece of software that has a rule engine, can bind to an ontology, and has a graphical user interface that makes editing of interface specifications easy. The

prototype implementation has shown the viability of the proposed computer aided interface control methodology.

7.2 Future research

The research that has been presented in this thesis can be extended in different ways as mentioned in the following.

7.2.1 Checking interface status between different CAD systems

To have a fully automatic interface control process, future CAD vendors should include some of the proposed functionalities of the interface control software in their CAD systems. Creating and editing port attributes can be done in CAD systems, and the interface control software can then be used as a status checker. In other words, the interface control software can extract port specifications from different CAD systems to check the status of interfaces. In this way the interface control software acts as a communication medium rather than a port specification tool. The XML/XSD technology is a great asset in facilitating such communication.

7.2.2 Component compatibility

Component compatibility issues can play an important role in the design of distributed systems. In a distributed system, components are connected by means of wires, pipes, wireless communication, etc., where the exact orientation and spatial location of components does not matter. In such systems, compatibility of ports approximately becomes a *sufficient* condition for compatibility of components. By approximately we mean there can still be unintended interferences between components, which may or may not be negligible. Port compatibility can only guarantee the correctness of intended interactions between components.

By using a formal model for component compatibility, it may be possible to synthesize a distributed physical system from a repository of well encapsulated

components that is available on a network or the internet. Future research should investigate this idea. Compatible components in such a repository can be found by a search engine. A system designer can pose a query to the repository to find compatible components that can be used in his/her system.

REFERENCES

ARMSTRONG, D. J. 2006. The quarks of object-oriented development. *Communications of the ACM,* 49 (2)**,** 123–128.

ATALLAH, M. 1984. *International Journal of Computer and Information Science,* 13 (4)**,** 279–290.

BERNERS-LEE, T., HENDLER, J. & LASSILA, O. 2001. The semantic Web: A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American,* 284 (5)**,** 34–43.

BETTIG, B. & GERSHENSON, J. K. 2010. The representation of module interfaces. *International Journal of Product Development,* 10 (4)**,** 291–317.

BLAIR-SMITH, H. 2010. System integration issues in Apollo 11. *Digital Avionics Systems Conference (DASC)*, Salt Lake City, UT.

BLYLER, J. 2004. Interface management: Managing complexity at the system interface. *IEEE Instrumentation and Measurement Magazine,* March 2004, pp. 32–37.

BROWNING, T. R. 2001. Applying the design structure matrix to system decomposition and integration problems: A review and new directions. *IEEE Transactions on Engineering Management,* 48 (3)**,** 292–306.

CAO, D. & FU, M. W. 2011. Port-based ontology modeling to support product conceptualization. *Robotics and Computer Integrated Manufacturing,* 27 (3)**,** 646–656.

CAO, D., RAMANI, K., FU, M. W. & ZHANG, R. 2009. Port-based ontology semantic similarities for module concept creation. *ASME International Design Engineering Technical Conferences & Computer and Information in Enginerring Conference (IDETC/CIE)*, San Diego, CA.

CLEMENTS, P. C. 1995. From subroutines to subsystems: Component-based software development. *The American Programmer (now: Cutter IT Journal),* November 1995.

DANILOVIC, M. & BROWNING, T. R. 2007. Managing complex product development projects with design structure matrices and domain mapping matrices. *International Journal of Project Management,* 25 (3)**,** 300–314.

DAU 2001. *Systems engineering fundamentals,* Defence Acquisition University Press (Department of Defence), Systems Management College, Belvoir, Virginia.

DECHTER, R. 2003. *Constraint processing,* Morgan Kaufmann Publishers, San Francisco, CA.

DIJKSTRA, E. W. 1982. *Selected writings on computing: A personal perspective,* Springer-Verlag, New York, NY.

EYLES, D. 2004. Tales from the lunar module guidance computer. *27th annual Guidance and Control Conference of the American Astronautical Society*, Breckenridge, Colorado.

FAA 2006. *Systems engineering manual*, Federal Aviation Administration.

FENVES, S. J., FOUFOU, S., BOCK, C., SUNDARSAN, R., BOUILLON, N. & SRIRAM, R. D. 2004. *CPM2: A revised core product model for representing design information (NISTIR 7185)*, National Institute of Standards and Technology (NIST), Gaithersburg, MD.

FORTIN, C., HUET, G., SANSCHAGRIN, B. & GAGNÉ, S. 2006. The CAMAQ project: a virtual immersion in aerospace industry practices. *World Transactions on Engineering and Technology Education,* 5 (2)**,** 287-290.

FRIEDENTHAL, S., MOORE, A. & STEINER, R. 2008. *A practical guide to SysML: The system modeling language,* Morgan Kaufmann OMG Press, Burlington, MA.

FRIEDMAN-HILL, E. 2003. *Jess in action: Java rule-based systems,* Manning Publications, Greenwich.

GOMEZ-PEREZ, A. 1997. Knowledge sharing and reuse. *In:* LIEBOWITZ, J. (ed.) *The handbook of applied expert systems.* CRC Press.

GRUBER, T. 1995. Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies,* 43 (5–6)**,** 907–928.

HEPP, M., BACHLECHNER, D. & SIORPAES, K. 2007. Harvesting Wiki consensus - using Wikipedia entries as ontology elements. *IEEE Internet Computing,* 11 (5)**,** 54-65.

HIRTZ, J., STONE, R. B., MCADAMS, D. A., SZYKMAN, S. & WOOD, K. L. 2002. A functional basis for engineering design: reconciling and evolving previous efforts. *Research in Engineering Design,* 13 (2)**,** 65–82.

HOLLAND, W. V. & BRONSVOORT, W. F. 2000. Assembly features in modeling and planning. *Robotics and Computer Integrated Manufacturing,* 16 (4)**,** 227–294.

HORROCKS, I., PATEL-SCHNEIDER, P. F. & VANHARMELEN, F. 2003. From SHIQ and RDF to OWL: The making of a web ontology language. *Web Semantics,* 1 (1)**,** 7–26.

HORVATH, I., VERGEEST, J. S. M. & KUCZOGI, G. 1998. Development and application of design concept ontologies for contextual conceptualization. *ASME Design Engineering Technical Conferences, (DETC 98)*, Atlanta, GA.

ISO 2004. *ISO 10303-109: Industrial automation systems and integration—product data representation and exchange—part 109: Integrated application resource: Kinematic and geometric constraints for assembly models*, International Organization for Standardization, Geneva, Switzerland.

KIM, K. Y., MANLEY, D. G. & YANG, H. 2006. Ontology-based assembly design and information sharing for collaborative product development. *Computer-Aided Design,* 38 (12)**,** 1233–1250.

KITAMURA, Y. & MIZOGUCHI, R. 2003. An ontological schema for sharing conceptual engineering knowledge. *International Workshop on Semantic Web Foundations and Application Technologies*, Nara, Japan.

KLEIN, M., FENSEL, D., HARMELEN, F. V. & HORROCKS, I. 2000. The relation between ontologies and XML schemas. *ECAI00 Workshop on Applications of Ontologies and Problem-Solving Methods*, Berlin.

LALLI, V. R., KASTNER, R. E. & HARTT, H. N. 1997. *Training manual for elements of interface definition and control (NASA Reference Publication 1370)*, National Aeronautics and Space Administration, Lewis Research Center, Cleveland, Ohio.

LEE, K. & GOSSARD, D. C. 1985. A hirearchial data structure for representing assemblies Part 1. *Computer-Aided Design,* 17 (2)**,** 15–19.

LI, W. D. & QIU, Z. M. 2006. State-of-the-art technologies and methodologies for collaborative product development. *International Journal of Production Research,* 44 (13)**,** 2525–2559.

LIANG, V. C. & PAREDIS, C. J. J. 2004. A port ontology for conceptual design of systems. *Computing and Information Science in Engineering,* 4 (3)**,** 206–217.

LIGEZA, A. 2006. *Logical foundations for rule-based systems,* Springer-Verlag, Berlin.

LIU, Y. & LIM, S. C. J. 2011. Using ontology for design information and knowledge management: A critical review. *In:* BERNARD, A. (ed.)

Global product development: Proceedings of the 20th CIRP Design Conference, Nantes, France.

MACCORMACK, A., RUSNAK, J. & BALDWIN, C. 2006. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science,* 52 (7)**,** 1015–1030.

NECHES, R., FIKES, R., FININ, T., GRUBER, T., PATIL, R., SENATOR, T. & SWARTOUT, W. R. 1991. Enabling technology for knowledge sharing. *AI Magazine,* pp. 36–56.

ORACLE 2012. *Java Architecture for XML binding* [Online]. Available: http://www.oracle.com/technetwork/articles/javase/index-140168.html [Accessed 8/11/2011].

PAHL, G. & BEITZ, W. 2005. *Engineering design: A systematic approach,* Springer-Verlag, London.

PAREDIS, C. J. J., DIAZ-CALDERON, A., SINHA, R. & KHOSLA, P. K. 2001. Composable models for simulation-based design. *Engineering with Computers,* 17 (2)**,** 112–128.

PARNAS, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM,* 15 (12)**,** 1053–1058.

PATIL, L., DUTTA, D. & SRIRAM, R. 2005. Ontology-based exchange of product data semantics. *IEEE Transactions on Automation Science and Engineering,* 2 (3)**,** 213–225.

PIMMLER, T. & EPPINGER, S. 1994. Integration analysis of product decompositions. *ASME Design Theory and Methodology Conference*, Minneapolis, MN.

ROSENBERG, R. C. & KARNOPP, D. C. 1983. *Introduction to physical system dynamics,* Mcgraw-Hill College, New York.

RUMBAUGH, J., JACOBSON, I. & BOOCH, G. 2004. *The Unified Modeling Language Reference Manual,* Addison Wesley, New York, NY.

SAE 2004. *AS5609: Aircraft/Store common interface control document format standard*, Society of Automotive Engineers International.

SHAH, J. J. & ROGERS, M. T. 1993. Assembly modeling as an extension of feature-based design. *Research in Engineering Design,* 5 (3–4)**,** 218–237.

SINGH, P. & BETTIG, B. 2003. Port-compatibility and connectability based assembly design. *Computing and Information Science in Engineering,* 4 (3)**,** 197–205.

SINHA, R. 2001. *Compositional design and simulation of engineered systems.* PhD Thesis, Carnegie Mellon University.

SOMMERVILLE, I. 2007. *Software engineering,* Pearson Education.

SOSA, M. E., EPPINGER, S. D. & ROWLES, C. M. 2003. Identifying modular and integrative systems and their impact on design team interactions. *Journal of Mechanical Design,* 125 (2)**,** 240–252.

SZYPERSKI, C., GRUNTZ, D. & MURER, S. 2002. *Component software: Beyond object oriented programing,* Addison-Wesley.

ULRICH, K. 1995. The role of product architecture in the manufacturing firm. *Research Policy,* 24 (3)**,** 419–440.

USAF 2005. *Systems engineering: concepts, processes, and techniques,* US Air Force, Space and Missile Systems Center.

USCHOLD, M. & GRUNINGER, M. 1996. Ontologies: Principles, methods, and applications. *Knowledge Engineering Review,* 11 (2)**,** 93–106.

W3C 1999. *HTML 4.01 specification* [Online], World Wide Web Consortium. Available: http://www.w3.org/TR/html401/ [Accessed 17/1/2012].

W3C 2006. *XML 1.1 specification* [Online]. Available: http://www.w3.org/TR/2006/REC-xml11-20060816/ [Accessed 18/1/2012].

W3C 2012. *XML Schema Definition Language 1.1* [Online], World Wide Web Consortium. Available: http://www.w3.org/TR/2012/PR-xmlschema11-1-20120119/ [Accessed 7/10/2010].

WASSON, C. S. 2006. *System analysis, design, and development: concepts, principles, and practices,* John Wiley & Sons Inc., New Jersey.

WIDEMAN, R. M. 2002. *Wideman comparative glossary of project management terms, v3.1* [Online]. Available: http://www.maxwideman.com/pmglossary/PMG_I03.htm [Accessed 1/1/2012].

WONGTHONGTHAM, P., KASISOPHA, N. & KOMCHALIAW, S. 2009. Community-oriented software engineering ontology evolution. *International Conference for Internet Technology and Secured Transactions ICITST*, London.