ENHANCING UML PORTS AND CONNECTORS TO INCREASE THE REUSABILITY AND PERFORMANCE OF AVIONICS SOFTWARE

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ALPER TOLGA KOCATAŞ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
COMPUTER ENGINEERING

JANUARY 2023

Approval of the thesis:

ENHANCING UML PORTS AND CONNECTORS TO INCREASE THE REUSABILITY AND PERFORMANCE OF AVIONICS SOFTWARE

submitted by **ALPER TOLGA KOCATAŞ** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** ______________

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering** ______________

Prof. Dr. Ali H. Doğru
Supervisor, **Computer Engineering, METU** ______________

Examining Committee Members:

Prof. Dr. Halit Oğuztüzün
Computer Engineering, METU ______________

Prof. Dr. Ali H. Doğru
Computer Engineering, METU ______________

Prof. Dr. Onur Demirörs
Computer Engineering, İzmir Institute of Technology ______________

Assoc. Prof. Dr. Hande Alemdar
Computer Engineering, METU ______________

Assist. Prof. Dr. Muhammed Çağrı Kaya
Computer Engineering, Ardahan University ______________

Date:05.01.2023

ABSTRACT

ENHANCING UML PORTS AND CONNECTORS TO INCREASE THE REUSABILITY AND PERFORMANCE OF AVIONICS SOFTWARE

Kocataş, Alper Tolga
Ph.D., Department of Computer Engineering
Supervisor: Prof. Dr. Ali H. Doğru

January 2023, 92 pages

Model-driven software development (MDSD) techniques have evolved vastly over the recent decades. MDSD aims to raise the abstraction level, allowing developers to produce accurate designs which are also easier to verify. The focus of this research is on developing methods in MDSD that can be utilized in software development. In the scope of this research, we first present a method for enriching the UML connectors with behavioral specifications for the exogenous coordination of the components. The aim is to free the components from the coordination responsibility, increasing their reusability. Second, we present an efficient, lightweight approach for the realization of the UML ports in object-oriented programming languages. The approach results in improved runtime performance and a significant decrease in code size. The first approach is validated using example connectors and cases from real-life large-scale avionics software. The second approach has been field-tested in actual flying avionics software for the last six years and has been proven to be successful.

Keywords: ALF, behavior, connector, fUML, model transformation, port, QVT, UML

ÖZ

AVİYONİK YAZILIM PERFORMANSINI VE TEKRAR KULLANILABİLİRLİĞİNİ ARTIRMAK İÇİN UML KAPI VE BAĞLAYICILARINI İYİLEŞTİRME YÖNTEMLERİ

Kocataş, Alper Tolga
Doktora, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Ali H. Doğru

Ocak 2023 , 92 sayfa

Model-güdümlü geliştirme (MGG) teknikleri son yıllarda oldukça ilerlemiştir. MGG soyutlama seviyesini yükselterek geliştiricilerin daha doğru ve doğrulaması daha kolay olan tasarımlar üretmesini sağlamaktadır. Bu araştırmanın odağı, yazılım geliştirme alanında kullanılabilecek olan MGG yaklaşımları geliştirmektir. Bu araştırma kapsamında, öncelikle UML bağlayıcılarını davranışsal betimlemelerle zenginleştirerek yazılım bileşenlerinin dışarıdan koordinasyonunu sağlayan bir yöntem sunulmaktadır. Bu yöntemle bileşenlerin koordinasyon sorumluluklarını azaltarak, tekrar kullanılabilirliklerini artırmak hedeflenmektedir. İkinci olarak, UML kapılarının nesneye yönelik programlama dillerinde etkin gerçeklenmesine yönelik bir yöntem sunulmaktadır. Sunulan bu yöntem yazılımın kod satır sayısının azalmasını sağlamakta ve çalışma performansını artırmaktadır.

Anahtar Kelimeler: ALF, davranış, bağlayıcı, fUML, model dönüşümü, kapı, UML

To Ahmet Alp, Elif and Ayşe Sena,
for there is always hope,
and for I wish that you walk in the bright trails of science and truth
to even farther distances.

ACKNOWLEDGMENTS

I would like to thank the people who supported me on this long journey. First, I would like to thank my advisor Prof. Dr. Ali Hikmet Dogru, and the TIK committee members, Prof. Dr. Halit Oğuztüzün and Prof. Dr. Onur Demirors, for their patience, support, and valuable feedback on this research. I would also like to thank Prof. Dr. Bedir Tekinerdogan for his help in my Ph.D. study in earlier years. I also want to thank Assoc. Prof. Dr. Hande Alemdar, and Assist. Prof. Dr. M. Çağrı Kaya for participating in my thesis jury and for their valuable feedback.

I want to thank ASELSAN for providing the resources for working on the research problems. I also want to thank my managers at ASELSAN, especially Yüksel Serdar and Dr. Şener Yılmaz, for motivating and supporting me throughout the program. I am also thankful to all academic and departmental staff of the Computer Engineering Department at METU, but most of all, to Sultan Arslan for her timely replies to my inquiries about administrative aspects of the Ph.D. program.

This would not be possible without the support of my family. I want to send my lovely thanks to my parents, Nedime Kocataş and Ziya Kocataş, for their unmatched support in whole aspects of my life.

Finally, I would like to thank all my close friends who motivated and supported me. I would like to thank Prof. Dr. Fatih Akbıyık, M.D, for keeping constant pressure on me to proceed. I am thankful to Soner Demiray for always being there, Pelin Ünver for her generous support, Assist. Prof. Dr. Esengül Özdemir for her guidance, and Dr. Erennur Tufan for her decades-long friendship. Last but not least, I would like to thank my climbing partners Uğur Eylem Alkan, Erkin Çakmak, Assist. Prof. Dr. Hasan Fehmi Özdemir, Prof. Dr. Bora Mavis, Eylem Elif Maviş, Mert Türedioğlu, Elif Yesim Kocoglu for their company on the hardest climbs and for the motivation they provided to keep on going.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ADL	Architecture Description Language
ALF	Action Language for Foundational UML
API	Application Programming Interface
ARINC	Aeronautical Radio, Incorporated
ATL	ATLAS Transformation Language
AVGRT	Average runtime duration (metric)
COSE	Component-Oriented Software Engineering
COSEML	Component Oriented Software Engineering Modeling Language
CSP	Communicating Sequential Processes
FIFO	First-In, First-Out
fUML	Foundational UML
IBM	International Business Machines
IMA	Integrated Modular Avionics
LSLOC	Logical Source Lines of Code (metric)
MARTE	The UML Profile for “Modeling and Analysis of Real-Time and Embedded Systems”
MDSD	Model-Driven Software Development
OCL	Object Constraint Language
OVRHD	Overhead of code generated for UML ports (metric)
QVT	Query/View/Transformations
QVTo	QVT Operational Mappings Language
ROOM	Real-time Object-Oriented Modeling
SBIN	Size of binary (metric)
SysML	Systems Modeling Language

TGEN	Time to generate code metric for UML ports (metric)
UML	Unified Modeling Language
UNIX	Uniplexed Information Computing System
VHDL	VHSIC Hardware Description Language
VHSIC	Very High-Speed Integrated Circuit

CHAPTER 1

INTRODUCTION

A model is a description of a system, where "system" is meant in the broadest sense and may include not only software and hardware but organizations and processes. It describes the system from a certain viewpoint for a certain category of stakeholders and at a certain level of abstraction [1]. Although models can be used as sketches that informally describe the design of the software and discuss it with stakeholders, the use of models can be taken beyond by application of the modern model-driven software development (MDSD) methods. In MDSD, models act not only as documentation, but the software implementation is also automatically generated from the models.

Among the many advantages of MDSD, the first one is that the design of the software is easier to understand and maintain due to the use of visual modeling notations. Second, the design documents are automatically updated, decreasing the software architecture erosion [2]. Furthermore, the well-formedness rules defined in the model can increase the quality of the software and support verification.

Among some disadvantages of MDSD, the most important ones are the cost of the initial effort to set up the modeling environment and the special training required to use the modeling languages and tools. Additionally, software systems whose requirements change rapidly during the development process are challenging to MDSD because changing software requirements cause continuous modifications to models. On the other hand, developing safety-critical real-time software systems encompasses challenges like rigorous verification and certification processes. In these systems, requirements tend to stay relatively stationary; hence MDSD can be more beneficial.

The Unified Modeling Language (UML) is the predominant modeling language mostly used for software design. UML allows modeling both structural and behavioral aspects of software systems. It has incorporated the most widely used features from existing modeling languages and Architecture Description Languages (ADLs). Ports and connectors introduced later to UML enabled the structural definition of software components independent from their environments. Since the 2.0 revision, UML also allows using ports and connectors in the definition of encapsulated classifiers, allowing the ports to be used in lower-level software design.

UML ports and connectors are suitable for developing avionics software. The term avionics is a blend of the words "aviation" and "electronics," and it means "aviation electronics." Examples of avionics devices in an aircraft are the radios used for communication, navigation devices, central control computers, and multi-function displays. Avionics software is the software running on these devices. According to the functions assigned to the device, the size of avionics software can range from hundreds of lines of code up to several millions of lines of code. The system functions allocated to the avionics software are categorized according to their safety-criticality level. The highest level of criticality implies that erroneous behavior of the software can lead to the loss of the aircraft.

Due to the safety-criticality aspect of the software, avionics software typically runs in a real-time environment. The hard real-time requirements require the software to run on real-time operating systems. Alternatively, the software can run on a deterministic scheduler designed using interrupts with no operating system.

Avionics software requires a precise definition of the software design, followed by rigorous and costly verification activities. Due to its ability to precisely define the structure of software, using UML ports and connectors fits well for the software architecture specification. Additionally, ports and connectors can be used to define the lower-level design elements of the avionics software.

This thesis aims to provide improved methods for using ports and connectors in the development of software. We focus on increasing the reusability of the software components and eliminating ambiguities with the help of behavior specifications associated with UML connectors. Additionally, we provide methods for the lightweight, efficient realization of UML ports in object-oriented programming languages to minimize the performance and verification overhead of using ports. We use large-scale avionics software projects for our case studies.

The rest of this chapter explains our research problems. Then, we explain the contributions of the thesis. We finally conclude the chapter with the outline of the thesis.

1.1 Motivation and Problem Definition

We utilized UML ports and connectors to develop large-scale avionics software in Aselsan using the IBM Rational Rhapsody Developer modeling tool [3]. The syntax of the UML covered by the tool allowed us to define and maintain projects with millions of source lines of code. Additionally, we developed various domain-specific languages and model transformations (e.g., [4], [5]) to raise the abstraction level to the extent that enabled seamless integration of the software components. We also employed rigorous model-checking rules that increase quality. This research was motivated by the problems and limitations of using UML ports and connectors we have encountered throughout the development of such large-scale avionics software projects.

UML ports can be used in the definition of the components, but they can also be used in the definition of lower-level classes. This enables modeling the structural definition of the software architecture, along with the detailed design. However, extensive usage of UML ports increases the code size significantly and causes additional runtime overhead. Therefore, we need an approach for realizing UML ports in programming languages with minimal runtime overhead and decreased code size. The increased runtime efficiency is beneficial in real-time software to fit in tight execution time resources. On the other hand, the majority of the cost in the development of avionics software comes from the activities related to verification. The reduced code size means less effort for the manual code reviews and the structural test coverage analyses required by the DO-178C [6] certification.

Solving the problem of efficient realization for UML ports is beneficial in the implementation domain. However, there is still room for improvement at the design level. UML specification supports n-ary connectors that can connect more than two ports.

However, using n-ary connectors can lead to ambiguities, which cannot be resolved using the modeling mechanisms provided by UML. Additionally, the connectors in UML are only intended to specify *links*, which connect ports and other connectable elements. This means that the required coordination logic must reside within the components. However, implementing the coordination within the components complicates their designs and reduces their reusability. Therefore, we need an approach to define the coordination behaviors outside the software components. We aim to solve the issues related to ambiguity and the problem of externally coordinating software components by associating behaviors with connectors.

1.2 Contributions and Novelties

The main contribution of this dissertation is related to improving the usability of UML ports and connectors. Specifically, this includes associating behaviors with UML connectors to fill the gaps in UML regarding ambiguous cases that n-ary connectors may cause. Associated behaviors also enrich the UML connectors with the capability to perform exogenous coordination of the components. The exogenous coordination capability of the connectors frees the components from their coordination responsibilities, helping their designs become more cohesive and reusable.

We first presented the concept of enhancing UML connectors with behavioral specifications in a conference publication [7]. The publication introduced the motivation for the enhanced connectors and proposed a conceptual solution without presenting an implementation. Later, we completed the work by providing a solution implemented with model transformations. We also presented the results from our research on the applicability of the concept in real-life large-scale avionics software projects. We published the studies completing the conference paper in [8].

Finally, we published the lightweight realization method for UML ports in [9]. While the studies in [7] and [8] focused on improving design capabilities using UML, the lightweight UML port realization study focused on improving methods for transforming UML designs with ports and connectors into the implementation domain.

In summary, we produced three publications as side-products of this dissertation work. The contributions of this thesis research can be summarized as follows:

- We propose a novel method that associates behaviors with UML connectors for the exogenous coordination of software components.
- Based on the proposed method, we present example connectors specified as a proof-of-concept.
- We present cases from real-life large-scale avionics software projects where connectors with associated behaviors can simplify the design and increase the reusability of the components.
- We propose a novel method for efficiently realizing UML ports in object-oriented programming languages,
- We show that the proposed realization approach for the UML ports results in reduced code size and increased runtime performance in real-life large-scale avionics software projects.

1.3 The Outline of the Thesis

This thesis is organized into two main topics. Chapters dedicated to the main topics include their own introduction, motivation, related work, discussion, and conclusion sections. The two chapters are followed by a conclusion chapter that presents our final remarks, the relations between the two main topics, and the impact of the study. The rest of this thesis is organized as follows:

- Chapter 2 explains the method for enhancing UML connectors with behavioral specifications for the exogenous coordination of software components. The approach is validated using example connector specifications and cases from real-life avionics software projects.
- Chapter 3 presents the method for the lightweight realization of UML ports in object-oriented programming languages for improved runtime performance and reduced code size.
- Chapter 4 concludes the thesis with our remarks and possible future work.

CHAPTER 2

ENHANCING UML CONNECTORS WITH BEHAVIORAL ALF SPECIFICATIONS FOR EXOGENOUS COORDINATION OF SOFTWARE COMPONENTS

Connectors are powerful architectural elements that allow the specification of interactions between software components. Since the connectors do not include behavior in UML, the components include the behavior for coordinating the components, complicating the designs of components and decreasing their reusability. In this chapter, we propose the enrichment of UML connectors with behavioral specifications. The goal is to provide separation of concerns for the components so that they are freed from coordination duties. The reusability of the components may increase as a result of such exogenous coordination. Additionally, using the associated behaviors, we aim to resolve the ambiguities that arise when n-ary connectors are used. We use a series of QVTo transformations to transform UML models that include connector behaviors in ALF specifications into UML models which include fUML activities as connector behavior specifications. We present a set of example connectors specified using the proposed method. We execute the QVTo transformations on the example connectors to produce models that represent platform-independent definitions of the coordination behaviors. We also present and discuss cases from real-life large-scale avionics software projects in which using the proposed approach results in simpler and more flexible designs and can increase component reusability.[1]

[1] The study described in this chapter was published in Applied Sciences Journal in 2023 [8].

2.1 Introduction

Using models in software development to cope with complexity and increase quality is a proven approach employed by model-driven software development (MDSD). In MDSD, models do not only constitute documentation but they are also considered equal to code, as their implementation is automated [10]. MDSD saves time by increasing productivity and reducing implementation errors. The approach has become more popular since the introduction of the Unified Modeling Language (UML) [1]. Due to its ability to precisely define the structural and behavioral aspects of the software systems, UML is widely used in safety-critical real-time embedded software development. UML specification has matured over the years. Extensions for real-time software development and systems engineering are introduced. Open-source and commercial tool support for UML also increased significantly.

Although there are languages for software architecture description, the current version of UML covers the core concepts, such as components, ports, and connectors. UML provides various diagrams for modeling the structural and behavioral aspects of software systems. Component diagrams describe the structures of the components and the communication among them using ports and interfaces. The ports define interfaces of software components and their interactions with the environment using required and provided interfaces. Starting with UML 2.0, ports can also be included in the designs of encapsulated classifiers.

Among various advantages of using ports, the most important one is to decouple components and encapsulated classifiers from their environments. Second, ports provide an additional encapsulation for the interfaces of components and structured classifiers by providing or requiring specific interfaces and inhibiting access to others. Finally, since the ports provide unique interaction points, the same interface provided by different ports can semantically correspond to distinct capabilities. This makes it possible for an encapsulated classifier to provide an interface more than once as part of port contracts for distinct purposes.

Connectors are used to connect ports inside the definition of components and the encapsulated classifiers. Connectors are powerful architectural elements that allow the specification of interactions between encapsulated software components. While the components and ports provide context-independent encapsulated behaviors, connectors provide context-setting interaction patterns [11]. UML enables this distinction between the context of use and the use-independent context by providing encapsulated classifiers and connectors as separate entities.

Besides binary connectors that connect two ports, UML also supports n-ary connectors that can connect with more than two ports. UML states that the semantics is the same when a port is connected to multiple ports using an n-ary connector or using multiple binary connectors representing the same n-ary connector. However, the semantics is left unspecified for either case. Furthermore, connectors are used only to specify *links* that connect connectable elements in UML. Therefore, the connected components must include the coordination behavior for routing requests to specific destinations. However, this entangling of coordination logic inside the components adds unnecessary complexity to their designs and decreases their reusability.

This chapter presents an approach to specify coordination logic outside the connected components by associating additional behavior specifications with UML connectors. We use the term *enhanced connectors* for the connectors with associated behaviors. Behaviors associated with the connectors enable them to perform exogenous coordination for the connected elements by relying exclusively on externally specified behaviors. Furthermore, the approach also provides a method for resolving ambiguities when n-ary connectors are used to coordinate more than two connectable elements.

We use Action Language for Foundational UML (ALF) [12] for specifying connector behaviors. ALF can be used to declare the structural and behavioral parts of Foundational UML (fUML) [13] models using textual notation. The textual notation simplifies the definition and maintenance of the models that become too complex using the graphical notation.

Using a series of QVT Operational Mappings Language (QVTo) [14] model transformations, we transform models with enhanced connectors into models that contain classes, objects, and fUML activities that implement connector behaviors. The resulting models include platform-independent definitions of the structural and behavioral specifications required for coordination. They can be used to generate code, or they can be transformed into other design models.

We illustrate the connector behavior specification method and the model transformations using a set of example connectors. We then present cases from real-life avionics software projects where we can use the method to simplify design, increase reusability, and resolve ambiguities. Furthermore, we evaluate the method's applicability for asynchronous coordination and its relation with the ProtocolStateMachines[2] in UML.

We first presented the enhanced connector concept in our previous study [7]. In the previous study, we presented the motivation for the enhanced connectors and proposed a conceptual solution without presenting an implementation. This chapter completes the previous study by providing a working solution implemented with model transformations. Additionally, we present results from our research on the applicability of enhanced connectors in real-life large-scale avionics software projects.

The rest of this chapter is organized as follows: Section 2.2 presents the related research. Section 2.3 describes the problem in detail and provides the motivation for the solution of the problem. Section 2.4 presents the approach for the specification of connector behaviors using ALF. Section 2.5 describes the model transformations. Section 2.6 presents the set of example connectors developed using the proposed approach. Section 2.7 lists the cases from real-life avionics software projects where we apply enhanced connector concepts. Section 2.8 presents an analysis of how the approach can be applied to asynchronous coordination. Section 2.9 includes an analysis of how enhanced connectors are related to ProtocolStateMachines in UML. Section 2.10 presents our discussions and future work. Finally, section 2.11 wraps up our conclusions.

2.2 Related Work

Connectors were studied before UML was introduced. One of the earlier studies defines architectural connectors as explicit semantic entities [11]. The study specifies the connectors as a collection of protocols that characterize the role of each partici-

[2] This chapter follows UML's general convention, which capitalizes the first letter of classifier names such as *Behavior, OpaqueBehavior, Property, ProtocolStateMachine*, etc.

pant in an interaction and how these roles interact. They illustrate how this scheme can be used to define a variety of architectural connectors, and they provide a formal semantics for connectors based on the Communicating Sequential Processes (CSP) [15]. Using the formal semantics definition, they describe a system in which architectural compatibility can be checked analogously to type-checking in programming languages. They use an architectural language (Wright)[16], augmented with a formal notation (CSP), which clears out ambiguities and allows more precise architectural definitions. The study also supports complex interaction patterns using the first-in, first-out (FIFO) ordering of messages.

The study in [17] evaluates the capabilities of UML 2.0 for documenting component and connector views. The study mentions that the lack of connectors and structured classifiers in prior UML versions weakened the language significantly. Therefore, the introduction of the connector concept in UML 2.0 added considerable strength to the language. They mentioned that a connector in UML is just a link between two or more connectable elements which cannot be associated with a behavioral description or attributes that characterize the connection. To compensate for the lack of behavior association, they considered using associations or association classes to represent the connectors in component and connector views.

UML-RT [18] brings some of the concepts in Real-time Object-Oriented Modeling (ROOM) [19] into UML. In UML-RT, connectors correspond to ROOM bindings, which are abstract views of signal-based communication channels interconnecting two or more ports. UML-RT protocols represent the behavioral aspects of connectors. The protocol concept was imported into later versions of UML from UML-RT as ProtocolStateMachines. ProtocolStateMachines are state machines that specify valid communication sequences. They can be associated with ports, interfaces, and classifiers.

MARTE [20] defines a real-time connector concept that inherits UML connectors and adds attributes specific to the real-time domain. While MARTE does not limit itself to binary connectors, it also does not add additional power to UML for the specification of connector behavior.

SysML [21], the modeling language created for systems engineering, is an extension of UML 2.5. SysML introduces new concepts, such as multi-level nesting of connector ends and connector properties. On the contrary, to keep the language simple, some of the complex features related to ports and connectors, such as ProtocolStateMachine, ProtocolConformance, Collaboration, and CollaborationUse, are excluded. However, the exclusion of notational and metamodel support for n-ary connectors is the most crucial concern for our study. Lacking n-ary connectors in SysML implies that the connectors for coordinating more than two connectable elements are only possible if an equivalent structure is formed using multiple binary connectors.

Outside of UML, Reo coordination language [22] is one of the mechanisms for coordination. By defining what is called an interaction machine, Reo frames the motivation for coordination [23]. An interaction machine is different from a Turing machine because while its next decision depends on its state and the current input in the input tape, it also depends on the inputs it obtains from its environment. The study argues that although the coordination protocols are crucial and error-prone parts of software, they are often embedded implicitly in the behavior of the software. This makes maintenance and modification of the coordination software difficult, and its reuse becomes almost impossible. As a result, the study argues that the coordination of the components should be handled separately and should be implemented outside of the components. The formal semantics of Reo has been presented in [24] using timed data streams.

The study presented in [25] also draws attention to exogenous communication. They argue against components calling methods from each other. The study argues that the components can be fully decoupled from each other only if connectors possess the whole control flow in the system. Therefore, they indicate that objects are not good candidates for components because they call methods from each other. Their work excludes n-ary connectors.

The work presented in [26] proposes an architectural definition based on components, ports, and connectors; however, they deliberately exclude n-ary relations. They introduce FIFO queues on component ports to aid asynchronous communications.

The study in [27] presents a formal framework to support the rigorous design of software architectures focusing on the communication aspects. They use UML class diagrams to describe the high-level architecture model, where classes represent the software components, and associations represent the relationships between them. They propose using Alloy [28] to formalize the communication styles and to verify the conformance of the communication styles at the model level. As a reusable library of connectors, they provide four basic connector behaviors: message passing, message passing with FIFO ordering, remote procedure call, and distributed shared memory. The proposed approach supports the validation of communication behavior at the software architecture design level of a distributed system.

The study presented in [29] uses the π-ADL [30] to provide a formal semantics for the SysADL [31] models. The study maps SysADL architecture descriptions to π-ADL using model transformations defined in the ATLAS Transformation Language (ATL) [32]. The behaviors of connectors in SysADL are described as compositions of ports on either side of the binary connectors. SysADL also enables using ALF statements to define details of component behaviors and to instantiate elements in the model. However, since SysADL is an architecture description language based on SysML modeling language, it inherits the same omissions from UML regarding the n-ary connectors.

The problem of making the connectors first-class citizens by giving them roles in component coordination and responsibility for the coordination logic was studied previously in the field of component-oriented software engineering (COSE) [33]. The study in [34] introduces a set of connectors to a component-based development environment where a variability model drives the configuration mechanisms in the flow of the application, components, and connectors. The study in [35] introduces an approach for managing hyper-connectivity in the IoT through connectors equipped with variability capability. The study only allows binary connectors to adapt different protocols. The study in [36] extends the XCOSEML [37] language with connector variability support to make it able to define configurable interaction options among the components. The studies in [38],[39],[40] enhance the component-oriented development approach with the capability to represent the dynamic behavior of the final system through a process model. Suggested architecture connects components to a central process so that if an interaction is required between two components, that is also managed through the process model.

2.3 Motivation and Problem Statement

In this section, we first present the motivation for our research. We then describe the problem in detail using examples. Finally, we illustrate the specific problems that should be addressed by the connector behavior specification method.

2.3.1 Motivation

In its current form, a connector in UML connects two compatible connector ends, but it does not play an active role in coordinating the connected entities. Consequently, the coordination problem is addressed inside the connected components. This solution, called *endogenous coordination*, makes the design of the connected elements unnecessarily complex because their design must also include coordination logic. It also reduces reusability since we cannot use the components in a scenario that involves different coordination requirements.

Alternatively, we can implement the coordination logic within the connectors to achieve *exogenous coordination*. This type of coordination can increase the reusability of the connected components and can simplify their design since the connectors take on the coordination responsibility. It can also allow the reuse of the connector behavior specifications in similar coordination scenarios.

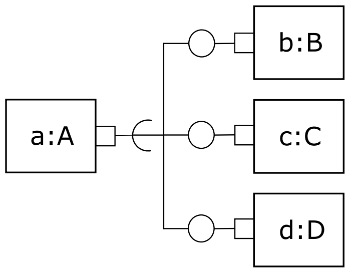


Figure 2.1: Single requester connected to multiple providers.

Our motivation is to achieve exogenous coordination by associating behaviors with the connectors. Besides increasing reusability, associated behaviors can help resolve ambiguities when we use n-ary connectors. In UML, we can connect a port that requires an interface to multiple ports providing the same interface. Figure 2.1 presents an example of this scenario. We can construct the scenario using three binary connectors or using a single quaternary connector that has four connector ends.

2.3.2 Problem Statement

The previous versions of UML leave the case shown in Figure 2.1 as a semantic variation point and do not define which providers will receive the request. The current version of UML [1] mentions that semantics is the same when we use three binary connectors or a single quaternary connector. It also indicates that the instance that will handle the request is determined at execution time. As a result, UML still does not provide a method to define the routing of requests for this case.

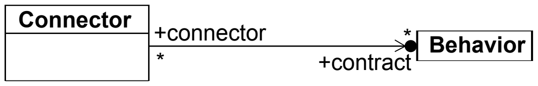


Figure 2.2: Abstract syntax in UML showing the contract role.

Figure 2.2 shows a part of the abstract syntax for the connectors defined in the UML. According to the definition, the *Behavior* classifier is associated with the *Connector* classifier using the *contract* role[3]. As an explanation for the contract role, UML indicates that *behaviors may be associated with connectors as contracts to specify valid interaction points across the connector*. However, the kind of behaviors that can be associated with connectors is not defined. Conversely, for other types of behaviors, such as *ProtocolStateMachines*, the exact purpose is documented, constraints are determined, and the application of the concept is demonstrated with concrete examples [1].

We propose using the behaviors indicated by the contract role to define the coordination behavior. As a result, the complex coordination logic implemented inside the connected elements can be taken outside and implemented as behaviors of connectors. Separating the coordination concern from other responsibilities of the elements can increase their reusability. Behaviors associated with connectors also help resolve the ambiguities which may arise when we use n-ary connectors to connect more than

[3] Note that the *contract role of a connector* is distinct from the *contract of a port* specified by the required and provided interfaces of the port.

two connectable elements. Finally, we can reuse the connector behavior specifications across scenarios that involve similar coordination requirements.

2.3.3 Merging Replies and Requests

Before going into details of connector behavior specification, we describe the specific issues the connector behavior specification method should cover. In the example scenario in Figure 2.1, let us assume that the connector behavior is designed so that when object *a* makes a request, the request is sent to all connected providers. Let us also assume that the interface used to specify the contract of the ports is *XIfc*. Finally, let us assume that the interface declares an operation *xOp*, which returns an integer.

When object *a* makes a request, the request will be sent to all three providers *b, c*, and *d*. Providers will handle the request and reply with an integer value. There will be three different replies from the providers, but the original requester object *a* expects only one reply. We can modify the design of object *a*, as shown in Figure 2.3, so it expects three distinct replies from three separate ports, but doing so will reduce its reusability. The reduction in reusability is caused by having to modify the design of object *a* if we add another provider.

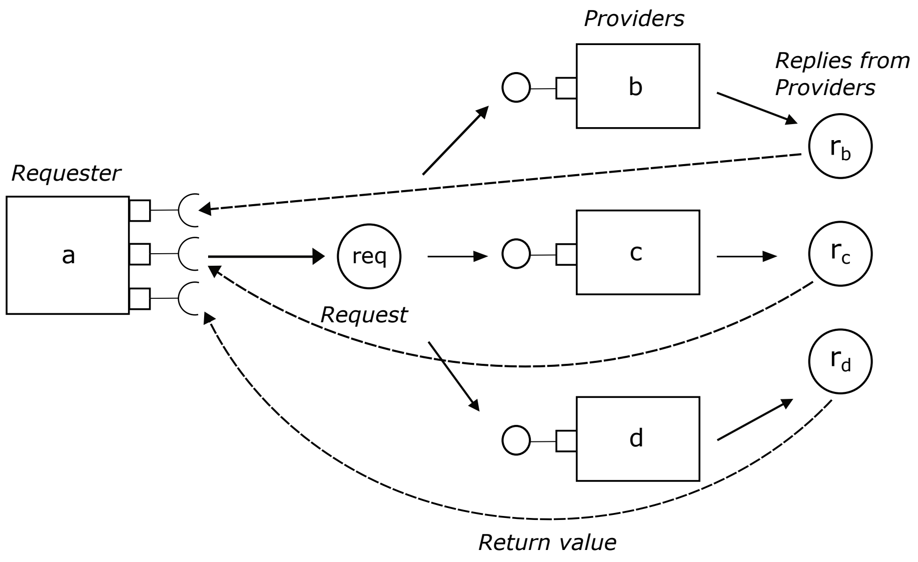


Figure 2.3: Modifying object *a* in Figure 2.1 to receive replies from multiple ports.

Without modifying the requester, the connector behavior can perform one of the operations described in Figure 2.4, 2.5, and 2.6 to solve the problem caused by multiple replies. In the first strategy shown in Figure 2.4, the connector behavior discards replies from all providers and sends a default reply to the requester. In the second strategy shown in Figure 2.5, the connector behavior chooses the reply from one of the providers. In the third strategy shown in Figure 2.6, the connector behavior uses the replies from all providers to calculate a *merged reply* and sends this merged reply to the requester.

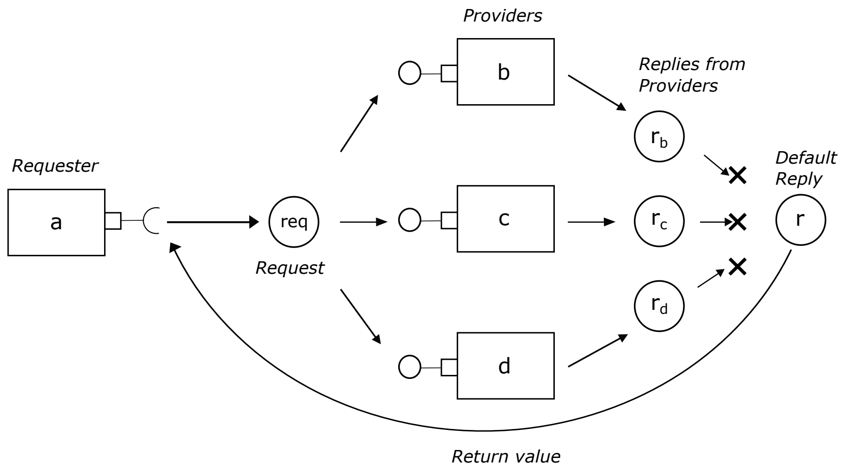


Figure 2.4: Reply merge strategy for the example case presented in Figure 2.1: Replies from providers are discarded, and a default reply is returned.

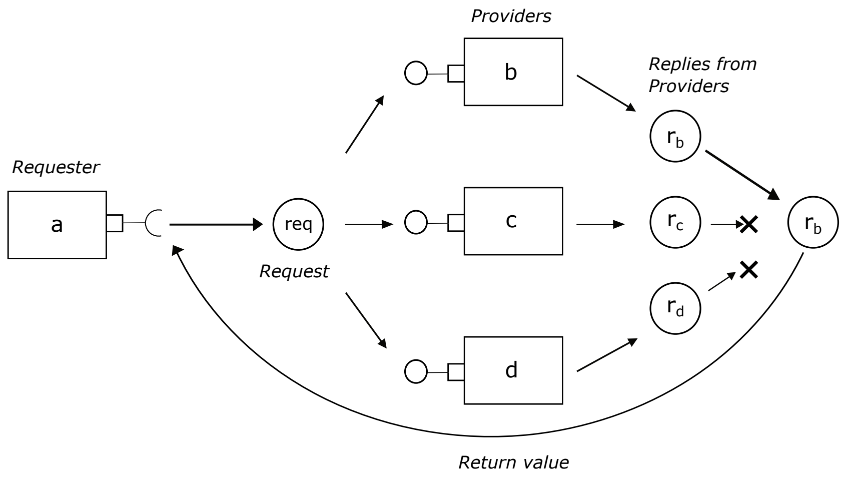


Figure 2.5: Reply merge strategy for the example case presented in Figure 2.1: One of the replies is chosen.

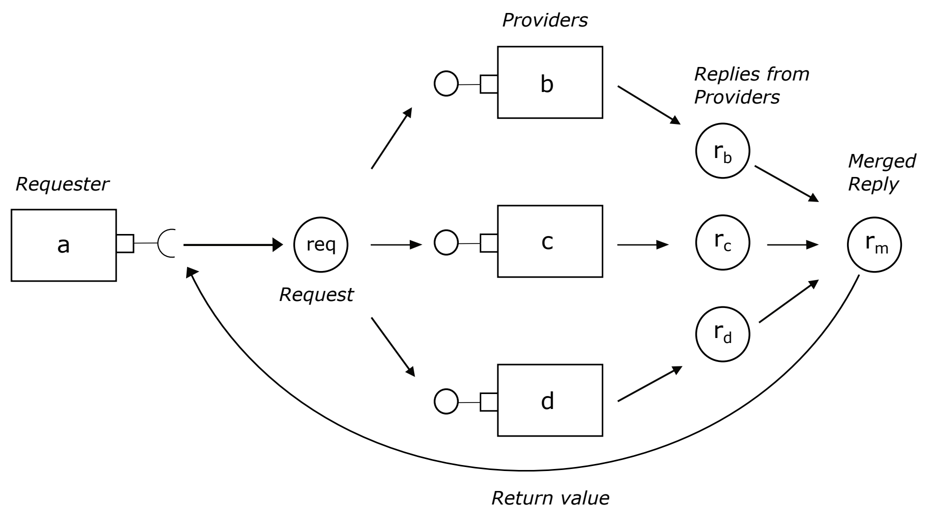


Figure 2.6: Reply merge strategy for the example case presented in Figure 2.1: Replies of providers are merged into a single reply.

A similar problem can exist for merging requests when there are multiple requesters, as shown in Figure 2.7. For this scenario, requests received from objects *a, b*, and *c* can be merged into a single request and sent to the provider object *d*. Alternatively, the connector can choose one of the requests and forward it to the provider, discarding other requests.

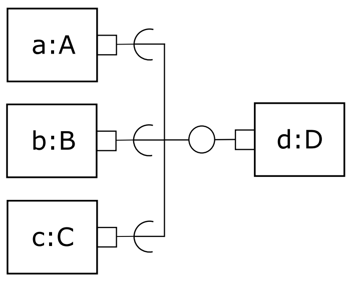


Figure 2.7: Multiple requesters connected to a single provider.

As a result, the two examples demonstrate the need for merging replies for cases involving multiple providers and the need for merging requests for cases involving multiple requesters. In the next section, we describe how to specify connector behaviors that also include the merge functionality.

2.4 Connector Behavior Specification

Behaviors associated with connectors should address the following concerns:

1. They should allow specifying the routing of requests. Routing of requests includes deciding the destinations for forwarding the requests and the conditions for forwarding the requests.
2. They should provide mechanisms for designing behaviors for merging the requests or replies when required.
3. They should enable the reuse of specified connector behaviors.
4. They should be platform-independent to allow transformation into platform-specific models.

In this section, we describe the connector behavior specification method which satisfies the above objectives.

2.4.1 Using Buffers for Replies and Requests

We use request and reply buffers for the specification of merge behaviors. We model the *reply merge* operation required in the "single requester-multiple providers" scenario (see Figure 2.1) using buffers dedicated to providers. We store the replies from providers in the *reply buffers*. Then, we operate a merge strategy over the elements in the reply buffers to combine the replies into a merged reply.

Similarly, we model the *request merge* operation required in the "multiple requesters-single provider" scenario (see Figure 2.7) using buffers dedicated to requesters. In this case, we store the requests from multiple requesters in the *request buffers*. Then, we operate a merge strategy over the elements stored in the request buffers to combine the requests into a merged request. We define two types of buffers for requests or replies:

- *Single-copy buffers* are for storing a snapshot copy of requests or replies in which subsequent requests or replies overwrite the previous ones.

- *FIFO buffers* are for storing subsequent requests or replies using first-in, first-out ordering.

For example, in the case presented in Figure 2.6, we store the three replies returned from the providers in three separate single-copy buffers. Then we iterate over each buffer and combine the returned replies into a merged reply.

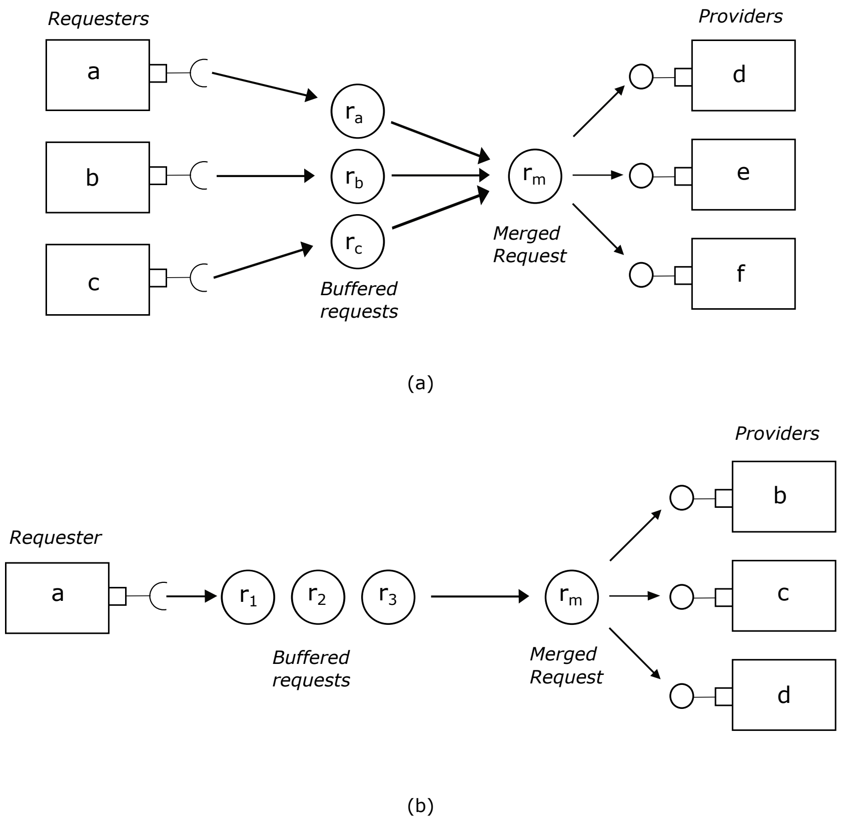


Figure 2.8: Merging requests using buffers: (**a**) Storing requests from multiple requesters in single-copy request buffers. (**b**) Storing subsequent requests from a requester in a FIFO request buffer.

Figure 2.8.a shows requests received from multiple requesters. In this example, we store the requests in single-copy request buffers. After receiving requests from all requesters, we combine the requests stored in buffers as a merged request and forward the merged request to the providers. Figure 2.8.b shows another example that involves a single requester with a single FIFO buffer. In this example, we store the subsequent requests received from the requester in the FIFO request buffer. When the FIFO buffer gets full, we combine the requests stored in the buffer as a merged request and then send that request to the connected providers.

Examples in Figure 2.4 and Figure 2.5 do not require buffers since the former connector discards all replies, and the latter chooses one of the replies and sends it back to the requester. Consequently, connector behaviors for these cases will not include the behavior for merging the replies.

2.4.2 Using ALF to Specify Connector Behavior

In UML, *Behavior* is an abstract classifier, and its specializations are *StateMachine, Interaction, Activity*, and *OpaqueBehavior*. Therefore, these are the types of behaviors that can be associated with the connectors using the *contract* role. We propose using activities specified in ALF for connector behavior specification. ALF is a surface notation for fUML. It can be used to declare a model's structural and behavioral parts using textual notation. fUML is a subset of UML, which only includes a specific set of UML concepts to allow executable models.

Because ALF is a surface notation for fUML, complete models specified using ALF are also limited to the subset of UML covered by fUML. However, there are other intended uses for ALF. While ALF maps to the fUML subset to provide its execution semantics, in case the execution semantics is not required, ALF is also usable in the context of models not restricted to the fUML subset [12], [41]. By using ALF only to specify the coordination behaviors of the enhanced connectors, we allow other parts of the model to use the UML concepts outside of the fUML context. ALF provides significant advantages for the specification of activities. When we use graphical notation, the specification of even simple behaviors can become quite complex, hard to understand, and maintain. We can denote the same behaviors using ALF in a more readable and maintainable format. Figure 2.9, an example included in the ALF specification [12], shows the distinction between the two formats. The figure shows the implementation of the same quick sort algorithm using both ALF notation and activity diagram notation.

```
activity Quicksort(in list: Integer[0..*] sequence):
  Integer[0..*] sequence
{
  x = list[1];
  if (x == null) {
    return null;
  } else {
    list1 = list->excludeAt(1);
    return Quicksort(list1->select a (a < x))->
           including(x)->
           union(Quicksort(list1->select b (b >= x)))
  }
}
```

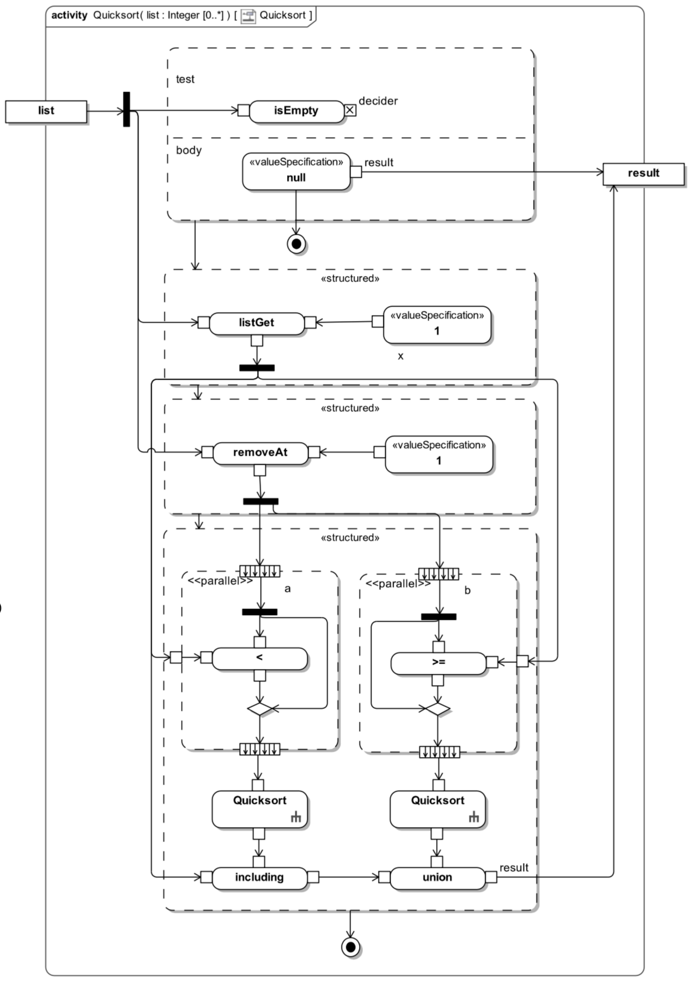
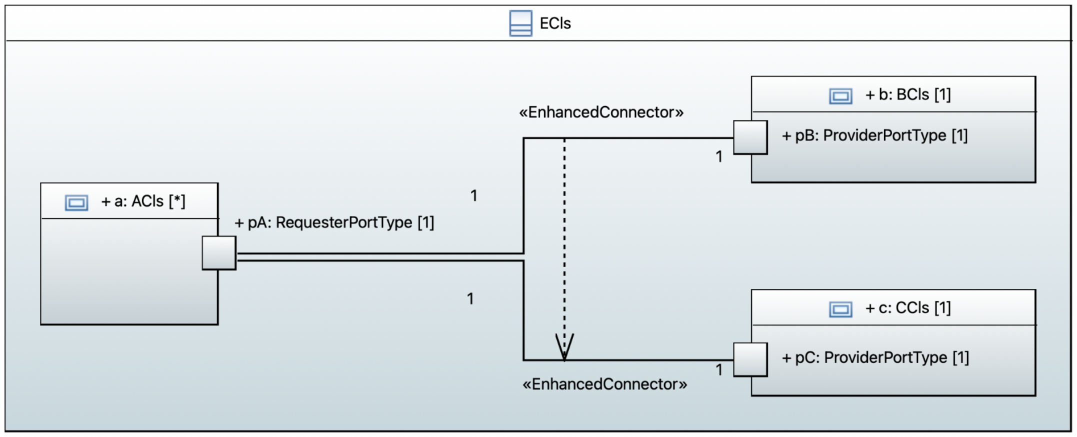


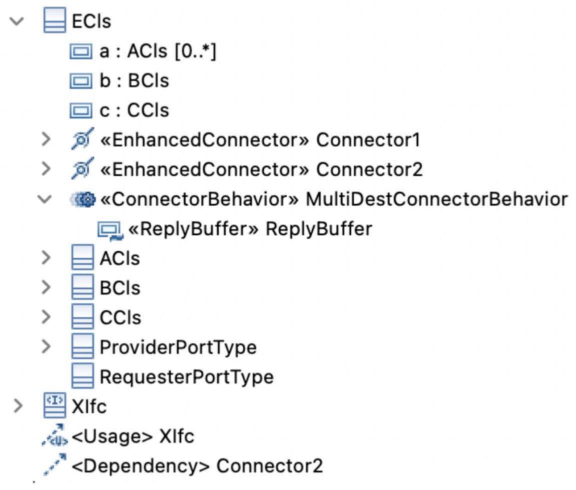
Figure 2.9: The Quicksort algorithm using ALF and graphical notation.

In our previous study [7], we proposed using *Interactions* specified by sequence diagrams for connector behavior specification. Since using imperative logic for the merge behaviors was not feasible with sequence diagrams, we have proposed using the target programming language to specify the behaviors for merging requests and replies. In this chapter, we address this problem by using ALF. The connector behaviors specified using ALF also include the behaviors for merging replies and requests, along with behaviors for routing requests. To define the connector behaviors, we use an *OpaqueBehavior* element for which we provide the specification using ALF in combination with specific *Properties* to denote the reply and the request buffers. The OpaqueBehavior allows the connector activity to have additional Properties, making room for stateful connector behaviors.

Figure 2.10 shows an example UML model where we define an enhanced UML connector. Listing 2.1 shows the body of the OpaqueBehavior developed using ALF for the connector behavior specification. The connector behavior sends incoming requests to all connected providers and then stores the replies from the providers in the single-copy reply buffers (line 3). *OutPort* denotes the array of providers (e.g., objects *b* and *c* in the figure). The term *arg* used in the *ifc.xOp(arg)* operation call represents the argument of the request.



(a)


ECls
a : ACls [0..*]
b : BCls
c : CCls
«EnhancedConnector» Connector1
«EnhancedConnector» Connector2
«ConnectorBehavior» MultiDestConnectorBehavior
«ReplyBuffer» ReplyBuffer
ACls
BCls
CCls
ProviderPortType
RequesterPortType
XIfc
<Usage> XIfc
<Dependency> Connector2

(b)

Figure 2.10: UML diagram and the UML model for the enhanced connector that sends requests to each provider: (**a**) Composite structure diagram showing the requester, providers, and the enhanced connector. (**b**) UML model showing the connector behavior.

Listing 2.1: Connector behavior specification using ALF for the model in Figure 2.10.

```
 1  Integer i = 1;
 2  for (ifc in this.OutPort) {
 3    this.ReplyBuffer[i] = ifc.xOp(arg);
 4    i++;
 5  }
 6
 7  // Reply merge strategy:
 8  i = 1;
 9  Integer sum = 0;
10  while (i <= PRV_CNT) {
11    Integer intval = this.ReplyBuffer[i];
12
13    if (intval != null) {
14      sum = sum + intval;
15      // Invalidate value stored in the buffer
16      this.ReplyBuffer[i] = null;
17    }
18    i++;
19  }
20  return sum;
```

After sending requests to all providers, we execute a reply merge strategy (lines 7-19), in which we merge replies from each provider using integer addition. The *PRV_CNT* term used in line 10 is a placeholder evaluated as the actual number of providers by the model transformations explained in the upcoming sections.

Table 2.1 shows the terms that can be used in connector behavior specifications. *OutPort* is used to route requests to a specific provider by indexing it with an integer. *ReplyBuffer[i]* is used to access the reply from a specific provider and *InPort[i].RequestBuffer* is used to access a request from a specific requester. Besides these terms, any legitimate ALF syntax can be used within the connector behavior specifications.

Table 2.1: Special terms used in connector behavior specifications.

Term	**Description**
OutPort[1..*]	Ordered array of references to the providers. An operation declared in the port contract can be called using this reference.
ReplyBuffer[1..*]	Reply buffer for storing replies from the providers.
InPort[1..*].RequestBuffer	Ordered array of references to the request buffers for each requester.
PRV_CNT	Placeholder for the number of providers.
REQ_CNT	Placeholder for the number of requesters.
arg	Name of the argument of the operation which is declared by the interface provided/requested by the port.

2.4.3 Enhanced Connector Profile

Using a profile is one of the methods for extending UML. Figure 2.11 shows the UML profile developed for representing the enhanced connector concepts. Defining stereotypes for enhanced connectors enables the model transformations to differentiate between the standard UML model elements and the model elements used for the enhanced connectors.

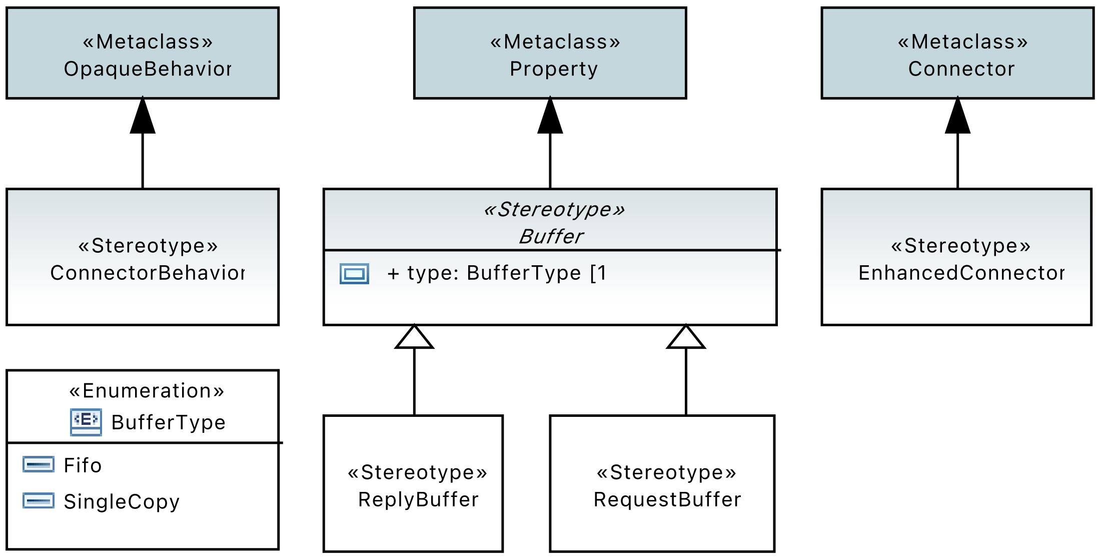


Figure 2.11: Enhanced connector profile.

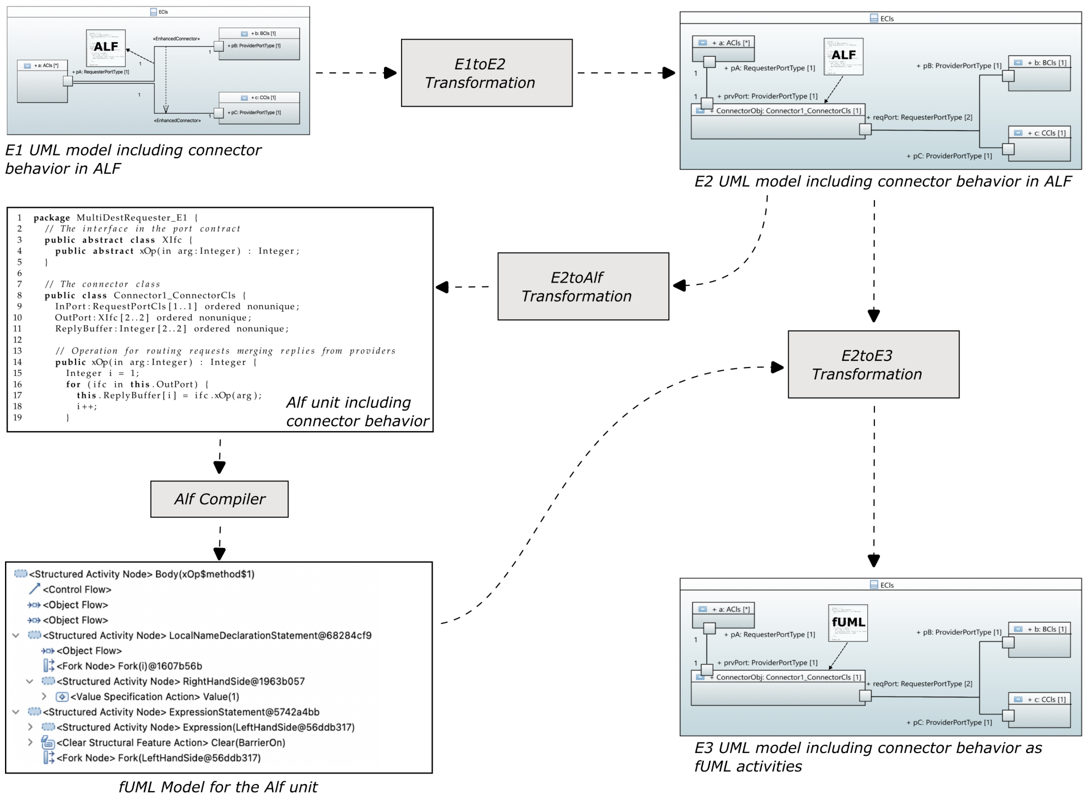
We apply the *ConnectorBehavior* stereotype to OpaqueBehaviors that we use to specify connector behaviors. In UML, since Behaviors are *Classifiers*, they are allowed to have *Properties*. We denote the reply and request buffers using specific properties of OpaqueBehaviors. *ReplyBuffer* and *RequestBuffer* stereotypes generalize the abstract *Buffer* stereotype, which extends the metaclass Property. The stereotype Buffer has a *type* property, whose type is *BufferType* enumeration. BufferType defines two enumeration literals that denote FIFO and single-copy buffers.

We distinguish the enhanced UML connectors from the standard UML connectors by setting their contract role to an OpaqueBehavior. We also apply the *EnhancedConnector* stereotype to enhanced connectors to avoid ambiguity in case the contract role is used for another purpose.

Most of the current UML tools do not support creating n-ary connectors. Therefore, we represent n-ary enhanced connectors using an *n-1* number of binary connectors. We designate one of the binary connectors as *the primary enhanced connector* by setting its contract role to an OpaqueBehavior. Then, we designate the remaining *n-2* binary connectors that are part of the n-ary enhanced connector as the *secondary enhanced connectors*. We create dependencies from the primary enhanced connector to the secondary enhanced connectors to form a group of connectors that constitute the n-ary connector. Figure 2.10.a shows one such dependency among the connectors using a dashed line. The contract role of the secondary enhanced connectors is empty.

2.5 Model Transformations

We use the stereotypes in the enhanced connector profile to develop the UML model that includes the enhanced connectors. We apply a series of model transformations that use this model as input. The model transformations convert the input model into a model which contains fUML activities that represent connector behaviors. Figure 2.12 shows the overall workflow for executing the model transformations.


E1 UML model including connector behavior in ALF
E1toE2 Transformation
E2 UML model including connector behavior in ALF
E2toAlf Transformation
E2toE3 Transformation
Alf unit including connector behavior
Alf Compiler
fUML Model for the Alf unit
E3 UML model including connector behavior as fUML activities
ALF
fUML
ECls
«EnhancedConnector»
+ a: ACls [*]
+ pA: RequesterPortType [1]
+ b: BCls [1]
+ pB: ProviderPortType [1]
+ c: CCls [1]
+ pC: ProviderPortType [1]
+ prvPort: ProviderPortType [1]
ConnectorObj: Connector1_ConnectorCls [1]
+ reqPort: RequesterPortType [2]

```
package MultiDestRequester_E1 {
  // The interface in the port contract
  public abstract class XIfc {
    public abstract xOp(in arg:Integer) : Integer;
  }

  // The connector class
  public class Connector1_ConnectorCls {
    InPort:RequestPortCls[1..1] ordered nonunique;
    OutPort:XIfc[2..2] ordered nonunique;
    ReplyBuffer:Integer[2..2] ordered nonunique;

    // Operation for routing requests merging replies from providers
    public xOp(in arg:Integer) : Integer {
      Integer i = 1;
      for (ifc in this.OutPort) {
        this.ReplyBuffer[i] = ifc.xOp(arg);
        i++;
      }
```

```
<Structured Activity Node> Body(xOp$method$1)
  <Control Flow>
  <Object Flow>
  <Object Flow>
  <Structured Activity Node> LocalNameDeclarationStatement@68284cf9
    <Object Flow>
    <Fork Node> Fork(i)@1607b56b
    <Structured Activity Node> RightHandSide@1963b057
      <Value Specification Action> Value(1)
  <Structured Activity Node> ExpressionStatement@5742a4bb
    <Structured Activity Node> Expression(LeftHandSide@56ddb317)
    <Clear Structural Feature Action> Clear(BarrierOn)
    <Fork Node> Fork(LeftHandSide@56ddb317)
```

Figure 2.12: Enhanced connector transformation process.

The model that is the input of the transformation engine is called the E1 model. We transform the E1 model into the E2 model, and then we eventually transform the E2 model into the E3 model[4].

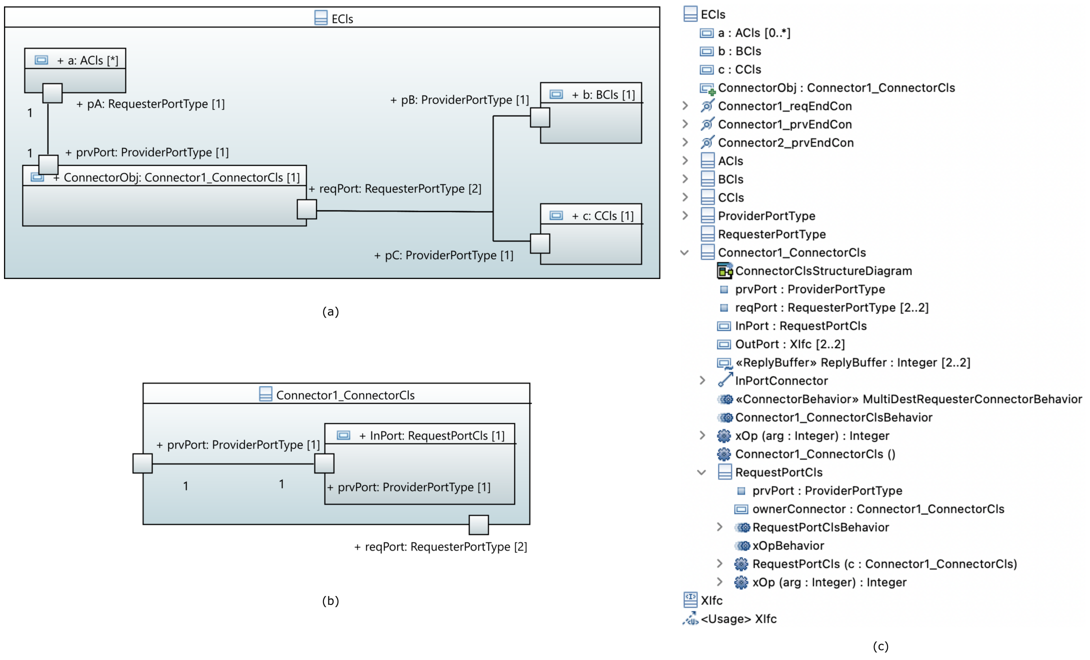
The E2 model is the platform-independent model that includes a class generated for the connector. This class implements the connector behavior, and an object of this class acts as the enhanced connector. Any specific attribute of the connector behavior in the E1 model, along with the properties denoting the reply and request buffers, becomes Properties of the class generated for the connector in the E2 model.

In the E2 model, in addition to the operation describing the connector behavior, we create operations for initializing buffers and setting up relations. We use ALF to provide the specifications of the connector behavior and the behavior of the created operations. We transform the ALF specifications into fUML activities in the E3 model.

[4] The terms E1, E2, and E3 are not part of any standard convention. They were introduced in the thesis to differentiate between multiple levels of models, which are inputs and outputs of the model transformations. The letter "E" stands for "enhanced connector'.'

The E3 model does not include any concepts specific to the enhanced connectors and thus can be transformed into a platform-independent or a platform-specific model. Alternatively, using a model-to-text transformation, the E3 model can be transformed into a programming language such as Java or C++.

We implemented the model transformations using QVTo [14]. We used Papyrus [42] to develop the UML models and the QVTo plugin of Eclipse [43] to develop and test model transformations. In the following sections, we describe the model transformations in detail.


ECls
+ a: ACls [*]
+ pA: RequesterPortType [1]
1
1
+ prvPort: ProviderPortType [1]
+ ConnectorObj: Connector1_ConnectorCls [1]
+ reqPort: RequesterPortType [2]
+ pB: ProviderPortType [1]
+ b: BCls [1]
+ c: CCls [1]
+ pC: ProviderPortType [1]
(a)
Connector1_ConnectorCls
+ prvPort: ProviderPortType [1]
+ InPort: RequestPortCls [1]
1
1
+ prvPort: ProviderPortType [1]
+ reqPort: RequesterPortType [2]
(b)
(c)

```
ECls
  a : ACls [0..*]
  b : BCls
  c : CCls
  ConnectorObj : Connector1_ConnectorCls
> Connector1_reqEndCon
> Connector1_prvEndCon
> Connector2_prvEndCon
> ACls
> BCls
> CCls
> ProviderPortType
  RequesterPortType
v Connector1_ConnectorCls
    ConnectorClsStructureDiagram
    prvPort : ProviderPortType
    reqPort : RequesterPortType [2..2]
    InPort : RequestPortCls
    OutPort : XIfc [2..2]
    «ReplyBuffer» ReplyBuffer : Integer [2..2]
  > InPortConnector
    «ConnectorBehavior» MultiDestRequesterConnectorBehavior
    Connector1_ConnectorClsBehavior
  > xOp (arg : Integer) : Integer
    Connector1_ConnectorCls ()
  v RequestPortCls
      prvPort : ProviderPortType
      ownerConnector : Connector1_ConnectorCls
    > RequestPortClsBehavior
      xOpBehavior
    > RequestPortCls (c : Connector1_ConnectorCls)
    > xOp (arg : Integer) : Integer
XIfc
<Usage> XIfc
```

Figure 2.13: E2 model for the example shown in Figure 2.10: (**a**) Composite structure diagram that contains the requester, providers, and the object of the connector class. (**b**) Composite structure diagram for the connector class. (**c**) UML model showing the connector behavior and other behaviors generated by the model transformation.

2.5.1 Transformation from the E1 Model into the E2 Model

We construct the E2 model from the E1 model using a model transformation. Figure 2.13 shows the E2 model that results from the E1 model shown in Figure 2.10. The transformation from the E1 model into the E2 model involves performing the following steps for each enhanced connector found in the E1 model:

- Let us call the class which contains the enhanced connector the *container class* (e.g., the class ECls shown in Figure 2.10 and Figure 2.13). A class with the name *ConnectorCls_<ConnectorName>* is created as a nested class of the container class. We call this class the *connector class*.

- An object of the connector class is created under the container class.

- Let *XIfc* be the interface required or provided by the ports connected by the enhanced connector. Two ports are added to the connector class with the names *reqPort* and *prvPort*, which require and provide the interface XIfc, respectively.

- Connectors that connect the requesters to the prvPort of the connector class are created.

- Connectors that connect the providers to the reqPort of the connector class are created.

- A class with the name *RequestPortCls* is created as a nested class of the connector class. An object of this class with the name *InPort* is also created as a *Property* of the connector class.

- A port with the name *prvPort* is created in RequestPortCls, which provides the interface XIfc.

- Properties of the OpaqueBehavior that represent the enhanced connector behavior, except the properties denoting the request buffers, are moved under the connector class. Properties that denote the request buffers are moved under RequestPortCls.

- A property named *ownerConnector* is created in RequestPortCls. The type of ownerConnector is set to the connector class.

- A *constructor* with a single argument is created for RequestPortCls. The type of the argument is the connector class. Implementation for the constructor is added as an OpaqueBehavior using ALF. The implementation sets the ownerConnector property to the argument of the constructor. This way, RequestPortCls knows the connector class, which owns itself.

- An operation that implements the operation declared by the interface XIfc is created under RequestPortCls. An OpaqueBehavior that acts as the *method* of this operation is also added. Implementation of the operation performs the following actions:
  - If a request buffer is defined, it stores the incoming request in the buffer allocated for the requester.
  - It forwards the request to the connector class using the ownerPort property. The connector then executes the actual coordination behavior.
- A *constructor* is created under the connector class, and an OpaqueBehavior using ALF is added as its method. The behavior of this constructor instantiates the InPort property of the connector class.
- Multiplicities of the port prvPort and the property InPort inside the connector class are set to the number of requesters.
- Multiplicities of the port reqPort and the property OutPort inside the connector class are set to the number of providers.

Behavior specifications of the created constructors and operations are specified using ALF. The E2 model contains all the structural and behavioral artifacts required to define the coordination behavior for the enhanced connector. In the next transformations, we transform the OpaqueBehaviors specified using ALF into behaviors specified using fUML activities.

2.5.2 Transformation from the E2 Model into the E3 Model

The transformation from the E2 model to the E3 model involves transforming OpaqueBehavior elements specified using ALF into fUML activities. Generated fUML activities are used as methods of operations, replacing OpaqueBehaviors.

We use the *ALF reference implementation* [44] to compile ALF specifications into fUML activities. ALF reference implementation can directly execute ALF models. Alternatively, it can transform ALF models into fUML models. However, since our

E2 model has elements outside of the fUML scope, we cannot use it directly. To use the ALF reference implementation, we generate an ALF module, which only includes the elements from the E2 model required for the enhanced connector behavior specification. The transformation responsible for generating this ALF module is called *E2toAlf*.

Listing 2.2 shows the ALF unit generated from the E2 model shown in Figure 2.13 using E2toAlf. We then use the ALF reference implementation to compile this model into an fUML model.

Listing 2.2: ALF unit generated from the E2 model shown in Figure 2.13.

```
package MultiDestRequester_E1 {

  // The interface in the port contract
  public abstract class XIfc {
    public abstract xOp(in arg:Integer) : Integer;
  }

  // The connector class
  public class Connector1_ConnectorCls {
    InPort:RequestPortCls[1..1] ordered nonunique;
    OutPort:XIfc[2..2] ordered nonunique;
    ReplyBuffer:Integer[2..2] ordered nonunique;

    // Operation for routing requests merging replies from providers
    public xOp(in arg:Integer) : Integer {
      Integer i = 1;
      for (ifc in this.OutPort) {
        this.ReplyBuffer[i] = ifc.xOp(arg);
        i++;
      }

      // Reply merge strategy:
      i = 1;
      Integer sum = 0;
      while (i <= 2) {
        Integer intval = this.ReplyBuffer[i];

        if (intval != null) {
          sum = sum + intval;
          // Invalidate value stored in the single copy buffer
          this.ReplyBuffer[i] = null;
        }
        i++;
      }
      return sum;
    }

    @Create public Connector1_ConnectorCls() {
      Integer i = 1;
      while (i <= 2) {
        this.InPort[i] =
          new RequestPortCls(this);
        i = i + 1;
      }
    }

    // The class for receiving and storing the requests
    public class RequestPortCls {
      public ownerConnector : Connector1_ConnectorCls[1..1];

      @Create
      public RequestPortCls(in c:Connector1_ConnectorCls) {
        this.ownerConnector  = c;
      }

      public xOp(in arg:Integer):Integer {
        // Forward the request to the connector object
        return this.ownerConnector.xOp(arg);
      }
    }
  }
}// end package
```

After we obtain the fUML model for the generated ALF unit, we import the fUML activities for the constructors and the operations which describe the connector behavior into the E2 model. The model transformation responsible for importing fUML activities into the E2 model is called *E2toE3*. It takes two inputs: the fUML model generated by ALF reference implementation and the E2 model. The fUML activities replace the OpaqueBehaviors that act as methods in the E2 model.

Imported fUML activities still contain references to some of the model elements which reside in the fUML model. For example, they reference the connector class, RequestPortCls, and specific Properties inside the fUML model. The referenced model elements also have their original copies in the E2 model. Therefore, the transformation updates the references to point to the corresponding model elements that reside in the E2 model. The resulting model after importing in the fUML activities and updating references is the E3 model.

2.5.3 Execution and Implementation of the Model Transformations

We developed the Java program *EcTransformer* to execute the transformations in the required order as shown in Figure 2.12. We have also experimented with ATL [32] to implement model transformations before choosing QVTo. Even though ATL could meet our requirements, we preferred QVTo because it was standardized by Object Management Group (OMG), like UML and ALF. However, we had to develop an equivalent method in QVTo for a practical feature of ATL called the *refining mode*.

In the refining mode, an ATL model transformation only changes specific elements in the input model, leaving everything else the same. The refining mode of ATL is distinct from the in-place model transformations of QVTo because it saves the output as a different model and does not modify the input model. It is suitable for our needs because we only transform the parts of the input model regarding enhanced connectors.

QVTo does not have the refining mode, but it supports in-place transformations. However, when we use this option, QVTo modifies the input model. We solve this problem by using the Java API of QVTo. We define the transformations as in-place transformations. Therefore, the model transformations modify the input models when we run them manually from inside Eclipse, using QVTo run configurations. However, when the Java program EcTransformer runs them, it takes the in-memory representation of the output models and saves them as different models. Using this method, we can obtain functionality in QVTo equivalent to the refining mode of ATL.

2.6 Example Connector Behaviors

In this section, we present example connectors developed using the proposed method. Besides clarifying the application of the method in different cases, the examples also demonstrate the expressive power of the connector behavior specification method.

2.6.1 The Round-Robin Requester

The round-robin requester connector coordinates a single requester and multiple providers. It sends each request made by the requester to the providers in a round-robin order and sends the received reply to the requester. Figure 2.10.a shows the composite structure diagram for the E1 model. Listing 2.3 shows the connector behavior specification.

Listing 2.3: Connector behavior specification for the round-robin requester.

```
Integer result = this.OutPort[this.Index].xOp(arg);
this.Index = this.Index + 1;
this.Index = this.Index % PRV_CNT;
return result;
```

The behavior uses the integer *Index* property of the OpaqueBehavior that represents the connector behavior in the E1 model. We initialize the *Index* property in the E1 model by providing a default value of *IntegerLiteral 1*. When a request arrives, the connector behavior sends the request to the provider indicated by the *Index* property, storing the reply in a local variable *result*. Then, it increments the *Index* property and applies a modulo operation to prevent the value of *Index* from overflowing past the number of providers. Finally, the connector behavior returns the value stored in the *result* as a reply to the requester.

Buffering mechanism is not required for this connector since it does not send requests to multiple providers simultaneously. The connector can also coordinate multiple requesters by sending the request it receives from any requester to the providers in a round-robin order. Additionally, based on this specification, we can define a *random destination sender* connector, which forwards incoming requests to a random provider using a similar behavior specification. We can achieve this by modifying the ALF code in line 2 to use a call to a random number generator.

2.6.2 The Multiple Destination Sender

In this type of coordination, there are multiple providers. There can be a single requester or multiple requesters. When a requester makes a request, the connector sends the request to all providers. Then, the connector merges the replies from the providers into a single reply and sends it back to the original requester. The requester does not know that the requests are sent to more than one provider. Therefore, the coordination happens in an exogenous style.

Listing 2.1 shows the connector behavior in ALF. The behavior forwards the request to each connected provider by iterating over the *OutPort* property of the connector (lines 1-5). It stores replies from the providers in the single-copy buffers. After each provider returns with a reply, the behavior executes a reply merge strategy by summing up the returned values (lines 8-19). Note that ALF mandates the *null* check in line 13. If we do not use the null check, ALF does not allow the summation of *intval* and *sum* since *intval* can be null, and multiplicities of the operands will not match in that case. The connector behavior invalidates the value stored in the single-copy buffer by setting it to *null* after using it (line 16). Finally, it returns the *sum* variable to the requester.

2.6.3 The Less Frequent Sender

The less frequent requester connector coordinates a single requester and a single provider. It can be used in cases where the requester makes frequent requests, but the provider expects them less frequently.

Listing 2.4 shows the connector behavior specification in ALF. The connector has a FIFO buffer which can store a limited number of requests. The connector achieves the coordination by collecting requests in the request buffer until the request buffer reaches a predefined size, indicated by the *MaxRequestCount* property of the OpaqueBehavior representing the connector behavior. When enough requests are collected, the connector behavior merges them as a single request by summing up the request values. Finally, the connector behavior sends the merged request to the provider.

Listing 2.4: Connector behavior specification for the less frequent sender.

```
Integer returnValue = this.DefaultReply;
if (this.InPort[1].RequestBuffer.size() == this.MaxRequestCount) {
  // Merge requests:
  Integer sum = 0;
  while (this.InPort[1].RequestBuffer.size() != 0) {
    Integer request = this.InPort[1].RequestBuffer.removeFirst();
    if (request != null) {
      sum = sum + request;
    }
  }
  returnValue = sum;
}
return returnValue;
```

The requests are dequeued from the FIFO request buffer until the buffer is empty. Figure 2.8.b presents an illustration in which subsequent requests are stored in the FIFO buffer and merged into a single request. For the requests that do not cause sending an actual request to the provider (i.e., when the number of requests in the buffer has not reached the *MaxRequestCount* limit), the connector replies to the requester with a default reply. The default reply value is stored in the *DefaultReply* property of the connector behavior, which is initialized to *IntegerLiteral 0* by providing its default value in the E1 model.

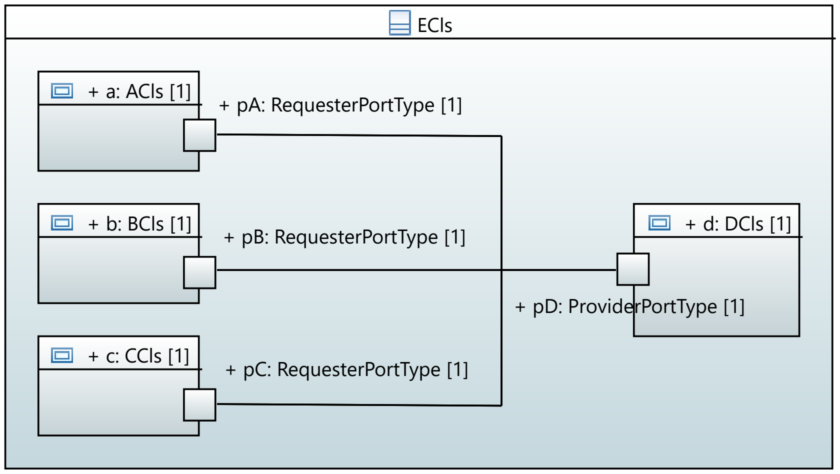

ECIs
+ a: ACIs [1]
+ pA: RequesterPortType [1]
+ b: BCIs [1]
+ pB: RequesterPortType [1]
+ c: CCIs [1]
+ pC: RequesterPortType [1]
+ d: DCIs [1]
+ pD: ProviderPortType [1]

Figure 2.14: E1 level UML model composite structure diagram for the request barrier connector.

2.6.4 The Request Barrier

The request barrier connector coordinates multiple requesters and a single provider. Figure 2.14 shows its structural configuration in the E1 model. The main functionality of the connector is to act as a barrier until all requesters make a request. It collects incoming requests in the single-copy request buffers. After the connector receives requests from each requester, it merges the requests stored in the request buffers into a single request and then forwards it to the provider. Listing 2.5 shows the connector behavior specification.

Listing 2.5: Connector behavior specification for the request barrier.

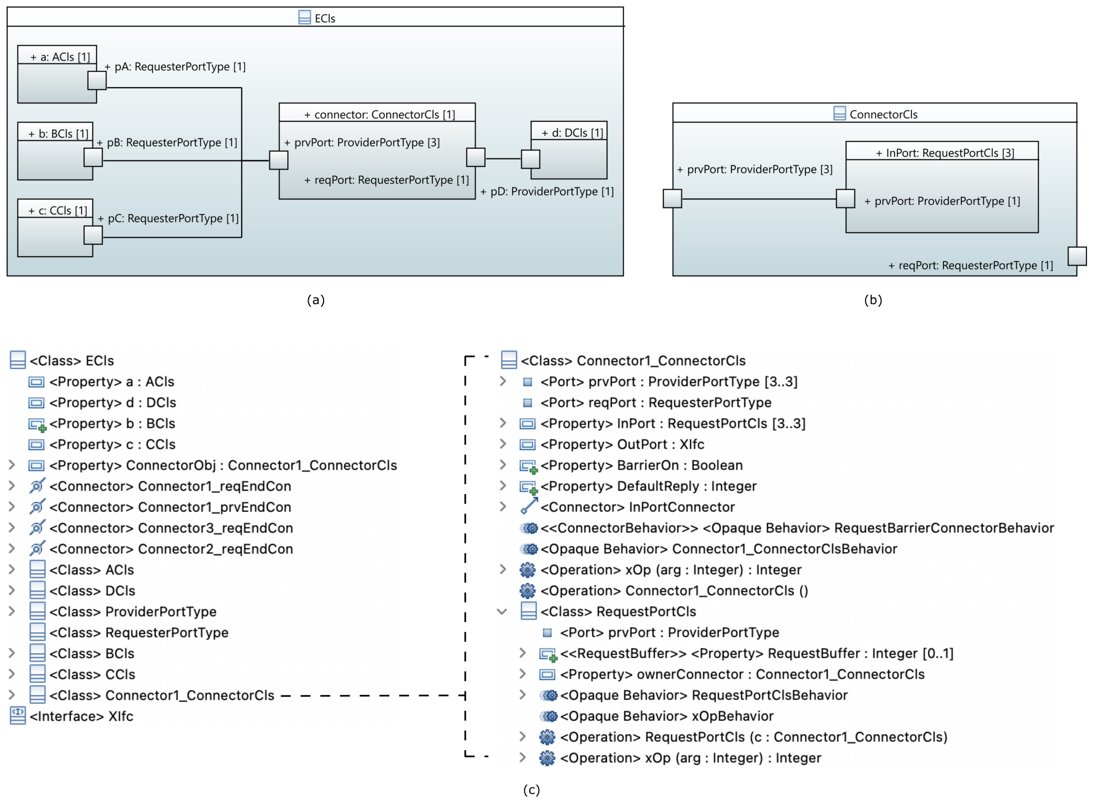
```
Integer i = 1;
this.BarrierOn = false;
while (i <= REQ_CNT) {
  Integer request = this.InPort[i].RequestBuffer;
  if (request == null) {
    this.BarrierOn = true;
  }
  i++;
}

```

```
11 Integer returnValue = this.DefaultReply;
12 if (this.BarrierOn == false) {
13   // Merge requests
14   Integer mergedRequest = 0;
15   i = 1;
16   while (i <= REQ_CNT) {
17     request = this.InPort[i].RequestBuffer;
18     if (request != null) {
19       mergedRequest += request;
20       // Invalidate the consumed request
21       this.InPort[i].RequestBuffer = null;
22     }
23     i++;
24   }
25   returnValue = this.OutPort[1].xOp(mergedRequest);
26   this.BarrierOn = true;
27 }
28 return returnValue;
```

The connector specification uses a boolean property *BarrierOn* to keep track of the connector state. When a request is received, the connector behavior checks if the request buffer of each requester is empty. It sets the *BarrierOn* flag to *true* if one of the requesters has an empty request buffer, indicating that the corresponding requester did not make any request yet (lines 1-9). Setting the *BarrierOn* flag to *true* means the barrier is closed, and the requests cannot pass through. In this case, the connector does not forward requests to the provider. Instead, it replies to the requester with a default reply (lines 11, 12, and 28).

The connector sets the *BarrierOn* flag to *false* if all requesters have non-empty request buffers, indicating that each requester made at least one request. Setting the *BarrierOn* flag to *false* means the barrier is open, and requests can pass through. In this case, the connector merges multiple requests into a single request and sends the merged request to the provider (lines 13-28).


ECls
+ a: ACls [1]
+ pA: RequesterPortType [1]
+ b: BCls [1]
+ pB: RequesterPortType [1]
+ c: CCls [1]
+ pC: RequesterPortType [1]
+ connector: ConnectorCls [1]
+ prvPort: ProviderPortType [3]
+ reqPort: RequesterPortType [1]
+ d: DCls [1]
+ pD: ProviderPortType [1]
(a)
ConnectorCls
+ prvPort: ProviderPortType [3]
+ InPort: RequestPortCls [3]
+ prvPort: ProviderPortType [1]
+ reqPort: RequesterPortType [1]
(b)
(c)

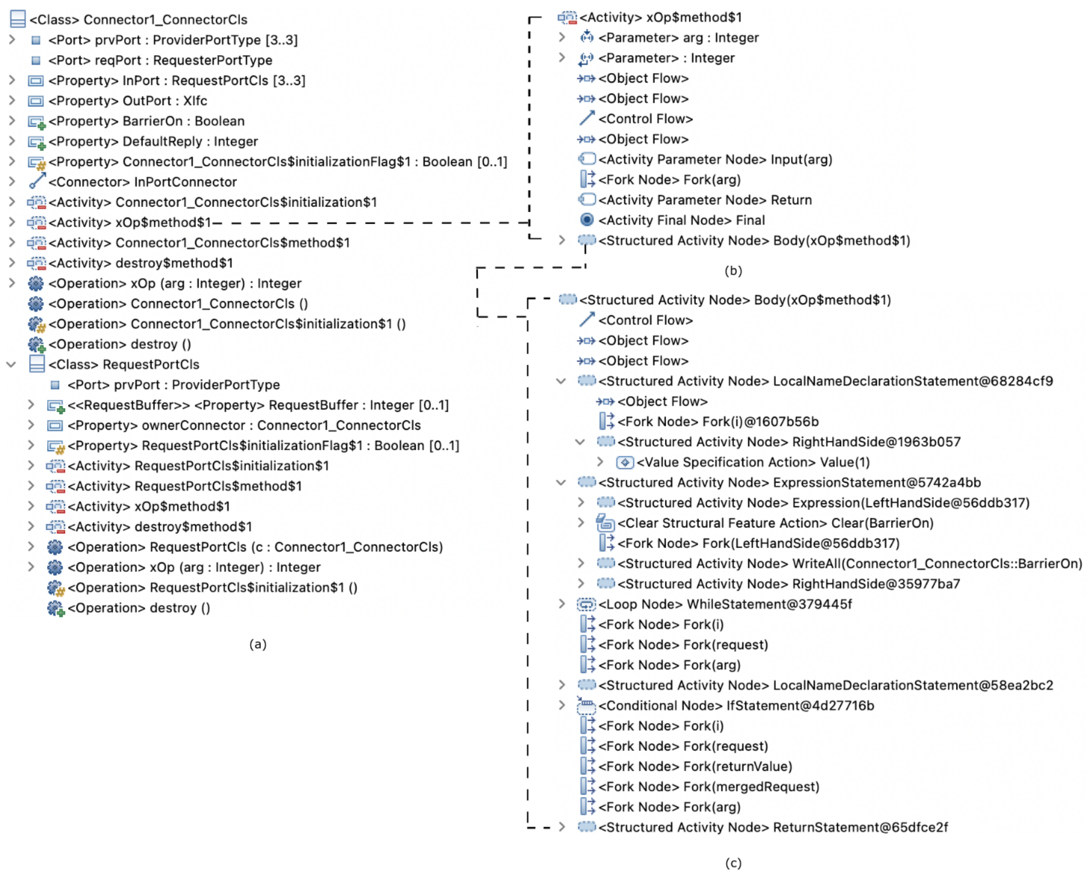
```
<Class> ECls
    <Property> a : ACls
    <Property> d : DCls
    <Property> b : BCls
    <Property> c : CCls
  > <Property> ConnectorObj : Connector1_ConnectorCls
  > <Connector> Connector1_reqEndCon
  > <Connector> Connector1_prvEndCon
  > <Connector> Connector3_reqEndCon
  > <Connector> Connector2_reqEndCon
  > <Class> ACls
  > <Class> DCls
  > <Class> ProviderPortType
    <Class> RequesterPortType
  > <Class> BCls
  > <Class> CCls
  > <Class> Connector1_ConnectorCls
<Interface> XIfc

<Class> Connector1_ConnectorCls
  > <Port> prvPort : ProviderPortType [3..3]
    <Port> reqPort : RequesterPortType
  > <Property> InPort : RequestPortCls [3..3]
  > <Property> OutPort : XIfc
  > <Property> BarrierOn : Boolean
  > <Property> DefaultReply : Integer
  > <Connector> InPortConnector
    <<ConnectorBehavior>> <Opaque Behavior> RequestBarrierConnectorBehavior
    <Opaque Behavior> Connector1_ConnectorClsBehavior
  > <Operation> xOp (arg : Integer) : Integer
    <Operation> Connector1_ConnectorCls ()
  v <Class> RequestPortCls
        <Port> prvPort : ProviderPortType
      > <<RequestBuffer>> <Property> RequestBuffer : Integer [0..1]
      > <Property> ownerConnector : Connector1_ConnectorCls
      > <Opaque Behavior> RequestPortClsBehavior
        <Opaque Behavior> xOpBehavior
      > <Operation> RequestPortCls (c : Connector1_ConnectorCls)
      > <Operation> xOp (arg : Integer) : Integer
```

Figure 2.15: E2 level composite structure diagrams and UML models for the request barrier connector: (**a**) Composite structure diagram for the request barrier connector. (**b**) Composite structure diagram for the connector class. (**c**) E2 level UML model.

Figure 2.15.a shows the composite structure diagram, and Figure 2.15.c shows the UML representation for the E2 model. The *E1toE2* model transformation replaces the enhanced connector in the E1 model with an object of the *connector class* denoting the connector. The model transformation sets the multiplicity of the port *prvPort* of the connector class to the number of requesters, which is *3*. Figure 2.15.b shows the internal structure of the connector class, which contains the *InPort* property.

The InPort property is an instance of the RequestPortCls class. The transformation also sets the multiplicity of the InPort property to the number of requesters. One of the responsibilities of the InPort property is to differentiate between requests received from different requesters. The operations defined in the connector class can access the request received from a specific requester by indexing the InPort property with the requester index. If we do not use the InPort property, we should directly imple-


```
<Class> Connector1_ConnectorCls
>  <Port> prvPort : ProviderPortType [3..3]
   <Port> reqPort : RequesterPortType
>  <Property> InPort : RequestPortCls [3..3]
>  <Property> OutPort : XIfc
>  <Property> BarrierOn : Boolean
>  <Property> DefaultReply : Integer
>  <Property> Connector1_ConnectorCls$initializationFlag$1 : Boolean [0..1]
>  <Connector> InPortConnector
>  <Activity> Connector1_ConnectorCls$initialization$1
>  <Activity> xOp$method$1
>  <Activity> Connector1_ConnectorCls$method$1
>  <Activity> destroy$method$1
>  <Operation> xOp (arg : Integer) : Integer
   <Operation> Connector1_ConnectorCls ()
   <Operation> Connector1_ConnectorCls$initialization$1 ()
   <Operation> destroy ()
v  <Class> RequestPortCls
      <Port> prvPort : ProviderPortType
   >  <<RequestBuffer>> <Property> RequestBuffer : Integer [0..1]
   >  <Property> ownerConnector : Connector1_ConnectorCls
   >  <Property> RequestPortCls$initializationFlag$1 : Boolean [0..1]
   >  <Activity> RequestPortCls$initialization$1
   >  <Activity> RequestPortCls$method$1
   >  <Activity> xOp$method$1
   >  <Activity> destroy$method$1
   >  <Operation> RequestPortCls (c : Connector1_ConnectorCls)
   >  <Operation> xOp (arg : Integer) : Integer
      <Operation> RequestPortCls$initialization$1 ()
      <Operation> destroy ()
(a)

<Activity> xOp$method$1
>  <Parameter> arg : Integer
>  <Parameter> : Integer
   <Object Flow>
   <Object Flow>
   <Control Flow>
   <Object Flow>
   <Activity Parameter Node> Input(arg)
   <Fork Node> Fork(arg)
   <Activity Parameter Node> Return
   <Activity Final Node> Final
>  <Structured Activity Node> Body(xOp$method$1)
(b)

<Structured Activity Node> Body(xOp$method$1)
   <Control Flow>
   <Object Flow>
   <Object Flow>
v  <Structured Activity Node> LocalNameDeclarationStatement@68284cf9
      <Object Flow>
      <Fork Node> Fork(i)@1607b56b
   v  <Structured Activity Node> RightHandSide@1963b057
      >  <Value Specification Action> Value(1)
v  <Structured Activity Node> ExpressionStatement@5742a4bb
   >  <Structured Activity Node> Expression(LeftHandSide@56ddb317)
   >  <Clear Structural Feature Action> Clear(BarrierOn)
      <Fork Node> Fork(LeftHandSide@56ddb317)
   >  <Structured Activity Node> WriteAll(Connector1_ConnectorCls::BarrierOn)
   >  <Structured Activity Node> RightHandSide@35977ba7
>  <Loop Node> WhileStatement@379445f
   <Fork Node> Fork(i)
   <Fork Node> Fork(request)
   <Fork Node> Fork(arg)
>  <Structured Activity Node> LocalNameDeclarationStatement@58ea2bc2
>  <Conditional Node> IfStatement@4d27716b
   <Fork Node> Fork(i)
   <Fork Node> Fork(request)
   <Fork Node> Fork(returnValue)
   <Fork Node> Fork(mergedRequest)
   <Fork Node> Fork(arg)
>  <Structured Activity Node> ReturnStatement@65dfce2f
(c)
```

Figure 2.16: (**a**) E3 UML model for the request barrier connector class. (**b**) E3 UML model for the request barrier connector fUML activity. (**c**) Inner elements of the E3 model request barrier connector fUML activity.

ment the operations declared in the coordinated interface inside the connector class. However, this implementation will cause subsequent requests to overwrite each other.

Figure 2.16.a shows the UML model for the connector class in the E3 model for the request barrier connector. The figure shows the fUML activities that replace the OpaqueBehaviors (e.g., *xOp$method$1 and Connector1_ConnectorCls$method$1*). For example, *xOp$method$1* is the fUML activity implementing the connector behavior for routing and merging requests. Figure 2.16.b shows the inner model elements for this activity, and Figure 2.16.c shows the internal model elements of the *structured activity node* inside this activity.

2.7 Application of Enhanced Connectors in Avionics Software Projects

This section presents the applications of enhanced connectors in real-life projects. We show examples from real-life projects in which using enhanced connectors simplifies the design and increases reusability. We examined six large-scale safety-critical avionics software projects. We developed a tool that searches the project model files to find cases where the application of the enhanced connectors is appropriate. We searched the model files for two cases. For the first case, we searched for classes having more than one port requiring the same interface. For the second case, we searched for classes that provide and require the same interface at multiple ports. Figure 2.17 illustrates the cases searched. We studied the results for both cases to determine if we can use enhanced connectors instead of existing connectors.

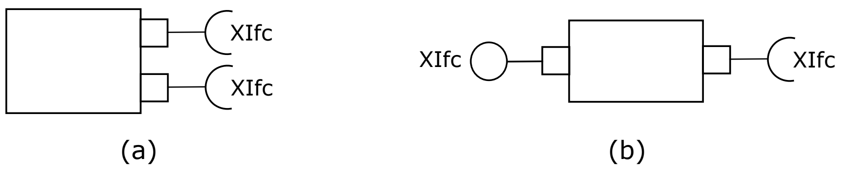


Figure 2.17: Cases searched in real-life projects: (**a**) Case 1: a class having more than one port that requires the same interface. (**b**) Case 2: a class that provides and requires the same interface at multiple ports.

In the following sections, first, we present background information regarding the development environment and the requirements for the projects we have studied. After that, we introduce the cases where using enhanced connectors is beneficial.

2.7.1 Background

The studied projects conform to *DO-178C, Software Considerations in Airborne Systems and Equipment Certification* [6]. DO-178C is the primary document by which the certification authorities such as *Federal Aviation Administration*, *European Union Aviation Safety Agency*, and *Transport Canada* approve all commercial software-based aerospace systems. Because of the inability to accurately apply reliability models to software, the concept of *development assurance* is applied by most safety-focused industries, including the aviation industry [45].

Table 2.2: DO-178C development assurance levels.

Level	**Failure condition**	**Objectives**	**With independence**
A	Catastrophic	71	30
B	Hazardous	69	18
C	Major	62	5
D	Minor	26	2
E	No Safety Effect	0	0

Table 2.2 shows the development assurance levels defined in DO-178C according to the safety impact of functionalities implemented in software. Level-A is the highest assurance level. Errors in level-A software can have catastrophic consequences leading to the loss of the aircraft, flight crew, or attendants. Level-D defines the development assurance level with the lowest safety impact. Level-E corresponds to *no safety impact*. DO-178C defines 71 objectives, all of which must be satisfied for level-A software. Besides, 30 of the objectives require independence for level-A software. For example, independence for software requirement reviews means that the engineers who develop the requirements cannot be the reviewer of the requirements, or they cannot develop the test cases for these requirements.

In addition to DO-178C, the avionics software projects we examined conform to ARINC-653 [46]. ARINC-653 is a software specification for space and time partitioning in safety-critical avionics real-time operating systems. Implementation of the standard allows hosting multiple applications of different software development assurance levels on the same hardware in the context of *integrated modular avionics* (IMA) [47] architecture.

ARINC-653 decouples running applications from the real-time operating system using the application programming interface called the *application executive* (APEX). Each application software is called a *partition*. Space partitioning indicates that each application has its own memory space, and time partitioning indicates that each application has its dedicated time slot allocated. Inside each partition, multiple processes can run, and multitasking between the processes using preemptive scheduling is allowed. In ARINC-653, a *process* is similar to a *thread* in the UNIX operating system. It shares the same address space with other processes and operates on the same global data. On the other hand, a *partition* in ARINC-653 corresponds to a *process* in the UNIX operating system, which assigns dedicated and protected virtual address spaces to the processes.

2.7.2 Case I: Multicasting Periodic Data

The first case for which the application of enhanced connectors is beneficial involves the communication between software components running inside the same ARINC-653 partition. In an ARINC-653 system, communication among the partitions is performed using *sampling ports* and *queuing ports*. A sampling port is used to send periodic data. The port retains a single copy of the data, and the recently arriving data overwrites the previous copy.

ARINC-653 allows connecting a source sampling port to multiple destination ports for sending data to more than one partition. If another partition needs the same data in the future, we only add a port to the list of destination sampling ports. We do not need to change the design of the source partition by adding another source port.

Although ARINC-653 solves the problem of multicasting data to partitions, we also have a similar problem inside the partitions. We can have multiple software components running inside a partition. The communication among these software components is modeled using UML ports, and the data should also be multicast among these components when required.

Figure 2.18 shows an example scenario where a software component in *partition A* produces data. The software component sends the data to ports that belong to two other software components in the same partition. Additionally, it sends the data to a source sampling port defined on *partition A*. The data is then sent from this port to two additional destination ports, which reside on partitions *B* and *C*.

As a result, the data produced by the source software component is sent to four destinations: *Cmp_1, Cmp_2, partition A, and partition B*. The ARINC-653 implementation

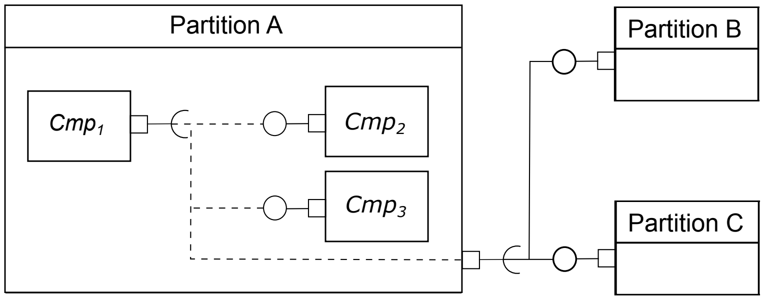


Figure 2.18: Sampling data produced by a software component sent to multiple destinations. An enhanced connector is used inside Partition A to multicast data.

handles the multicasting of sampling port data to *partition A and B* in this scenario. However, we need to implement a solution for multicasting the data to *Cmp_1, Cmp_2*, and the source sampling port of *partition A*.

Without using enhanced connectors, we have to manually multicast the data by creating three separate ports on the source software component or by inserting an additional object which replicates the data to multiple destinations using multiple source ports. Creating three ports on the source decreases reusability by coupling the design of the source component to the number of destinations. On the other hand, inserting an additional object to replicate the data requires manually designing intermediary classes and objects to perform the multicasting. We can solve this problem using the enhanced connector presented in Section 2.6.2: the multiple destination sender. By using the enhanced connector, we can decouple the source component from the multicasting concern.

In addition to multicasting periodic data among the top-level software components inside the partitions, we observed that lower-level objects inside the top-level software components also used distinct ports to send the same data to multiple objects. The multiple destination sender enhanced connector is also applicable for these cases.

2.7.3 Case II: Handling Different Configurations

Avionics software projects we studied include requirements for supporting different configurations of devices. An example is using separate communication devices in two different configurations. Version *A* of the aircraft uses the communication device *Com*$_1$, and version *B* uses *Com*$_2$. In this case, the software components which control *Com*$_1$ and *Com*$_2$ are designed to use the same interfaces to communicate with other software components. Depending on the project configuration, an intermediate class routes the requests to the active communication device.

Figure 2.19.a shows the *Selector* object in the existing design, which forwards the incoming requests to *Com*$_1$ or *Com*$_2$ device accordingly. Figure 2.19.b shows the new design, where we use an enhanced connector that reads the project configuration and forwards the requests to either device. The selector class is replaced with an enhanced connector. If a third device is used in the future, we can modify the connector behavior to handle the change.

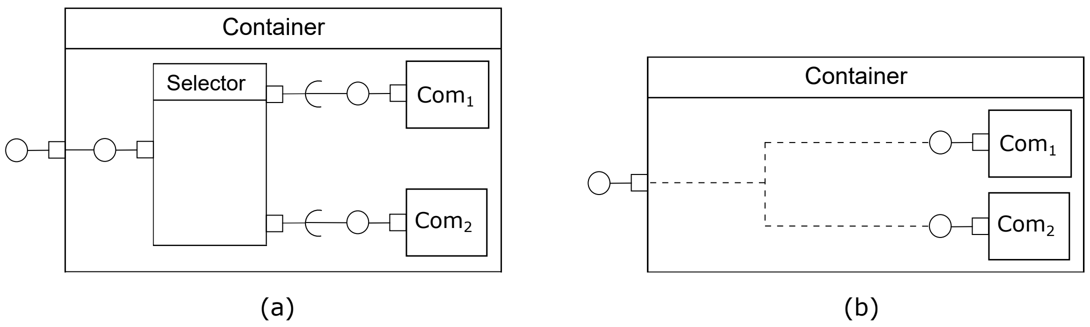


Figure 2.19: Selector pattern, which forwards requests to different communication devices depending on the project configuration: (**a**) Existing design that uses a dedicated object. (**b**) Updated design that uses an enhanced connector.

We encountered the conditional forwarding of requests in another case where two ports with identical required interfaces were connected to two ports of different objects. One of the two objects was for testing purposes, and the other was the live object. The application forwarded the requests to the test object when running in test mode and to the live object when not running in test mode. An enhanced connector can also handle this kind of conditional message routing.

2.7.4 Case III: Handling Redundant Devices on the Platform

In the projects we examined, some of the cases where multiple ports of the same class required the same interface were designed to handle the redundant devices in the platform. In avionics systems, there are often two devices of the same functionality to minimize the risk of losing critical functions. Having two devices also helps detect sensor errors by comparing readings from both devices. This system design method, known as *dual redundancy*, is used for the devices responsible for critical system functions.

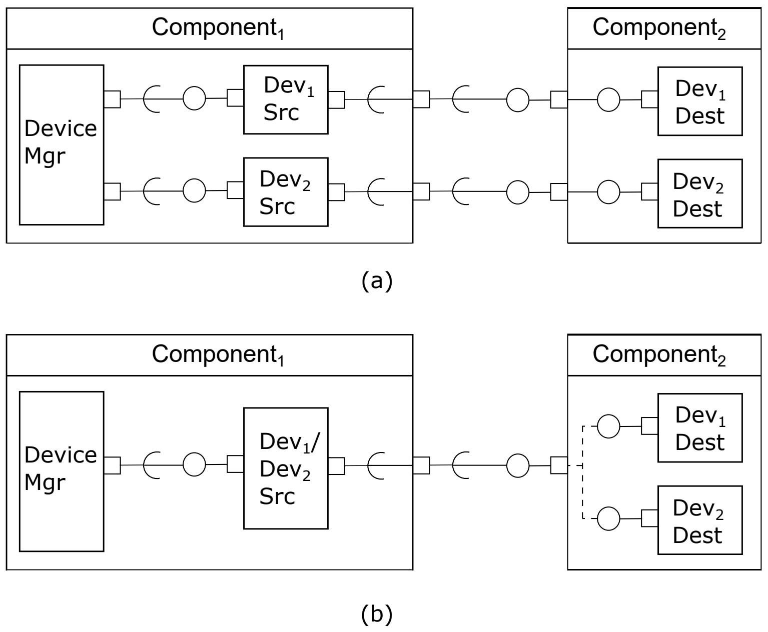


Figure 2.20: Handling Redundant Devices on the Platform: (**a**) Existing design: Dedicated communication channel for each device. (**b**) Updated design: Single communication channel for multiple devices using an enhanced connector inside Component$_2$.

Figure 2.20.a shows a typical dual redundant device management scenario in the projects we examined. In the existing design, *Component$_1$* is responsible for the device functionalities specific to the avionics software application. Responsibilities of this component can change depending on the avionics software application. On the other hand, *Component$_2$* is responsible for the device functionalities only in the scope of the device's capabilities. Responsibilities of *Component$_2$* do not change depending on the application, so it can be reused in avionics software applications that require a dual redundant setup of the same device.

Component$_1$ includes a device manager software component (*DeviceMgr*) which manages a dual redundant device configuration. Additionally, there are two separate objects in *Component$_1$* which perform application-specific device management and data transformation tasks for the individual devices (*Dev$_1$Src* and *Dev$_2$Src*). The data from the devices also follows two separate communication channels towards other software components and partitions. Data sent by *Dev$_1$Src* and *Dev$_2$Src* arrive at device controllers *Dev$_1$Dest* and *Dev$_2$Dest* inside *Component$_2$*.

The existing design is easy to understand and implement, but if we need to add a third copy of the device to the system in the future, we should establish a third communication channel. However, adding a third channel will change the design of all the software component classes on the communication path.

Figure 2.20.b shows the updated design. Using the *Dev$_1$/Dev$_2$Src* object, we can get requests for both devices and forward them into a single port by adding the device identification information to the requests. Then, the enhanced connector inside *Component$_2$* can multiplex the requests to the correct device by reading the device identification information. Compared to the original design, the updated design is more resilient against adding or removing devices. If we need to add a third device, we will not have to change the ports of the classes in the communication path (e.g., *Component$_1$*, *Component$_2$*).

2.7.5 Case IV: Blocking Requests Depending on the Condition

In this case, an intermediate object between two ports is used to allow or block requests. Figure 2.21.a shows the existing design. During runtime, when objects of these classes receive a request through the port that provides the interface, they check a particular condition. They forward the incoming request only if the checked condition holds. Otherwise, they block and drop the request.

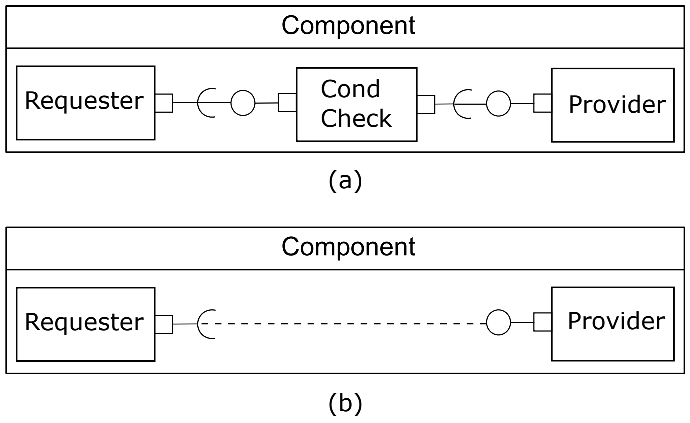


Figure 2.21: Blocking or forwarding requests depending on the runtime condition: (**a**) Existing design: Using an intermediate object for conditionally blocking the request. (**b**) Updated design: Using an enhanced connector that can conditionally block the request.

Figure 2.21.b shows the updated design using an enhanced connector. We implement the required coordination behavior by checking the particular condition inside the connector behavior. We observed that the conditional forwarding and blocking of the requests is used in several places in the projects, and they are implemented as separate classes and objects. Enhanced connectors can replace these objects, simplifying the design.

2.7.6 Case V: Choosing a Single Reply Among Many Providers

In this scenario, multiple identical devices can respond to a request. A software component that acts like the multiple destination sender connector sends the same request to all devices. When all the providers reply, the response from one of the providers is used according to a selection algorithm.

Figure 2.22.a shows existing design, where the requester object is responsible for its own functionality, in addition to the responsibility of multicasting requests to four devices and choosing a reply. Figure 2.22.b shows the updated design using an enhanced connector. The enhanced connector can perform multicasting of the request to each provider and then selection of the reply of a specific provider. We can decouple the requesting side from the selection algorithm and increase its reusability by using the enhanced connector.

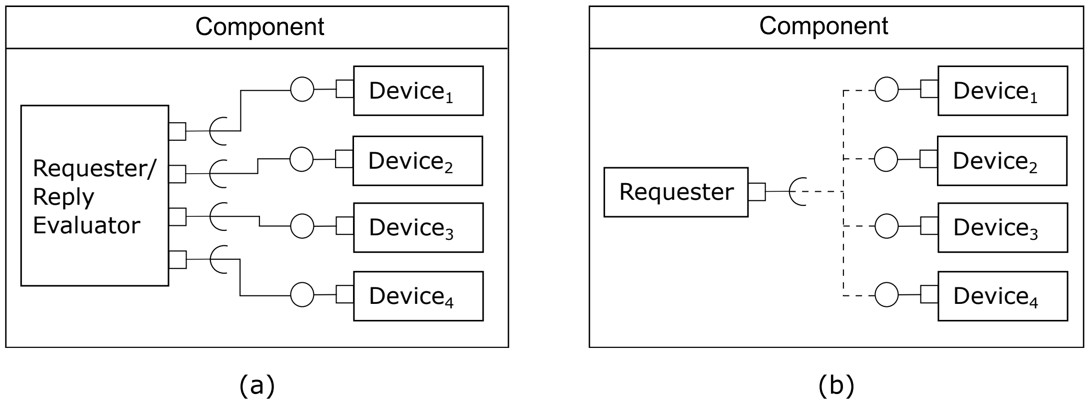


Figure 2.22: Choosing a single reply among the replies from many providers: (**a**) Existing design: The requester contains the behavior for multicasting the requests and evaluating the replies. (**b**) Updated design: Using an enhanced connector to multicast requests and evaluate the replies.

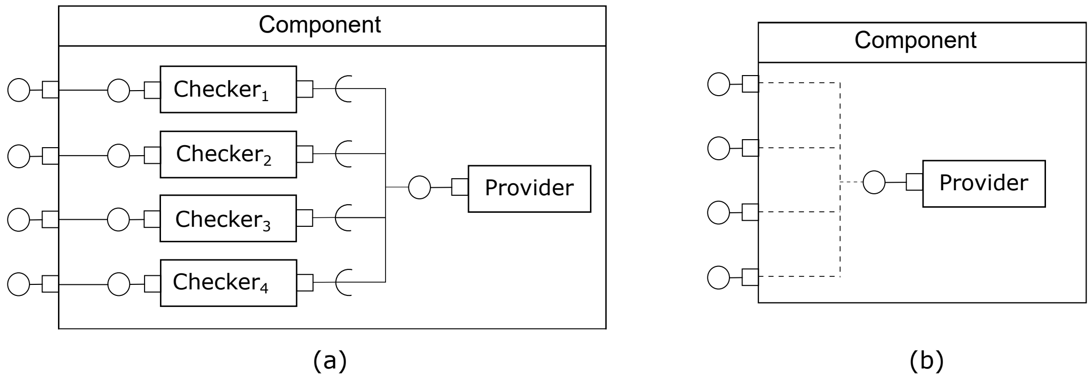


Figure 2.23: Discriminating between requesters to selectively accept requests from specific sources: (**a**) Existing design: Using dedicated objects to set the source field of incoming requests. (**b**) Updated design: Using an enhanced connector to set the source field of incoming requests.

2.7.7 Case VI: Discriminating Between Multiple Requesters

Figure 2.23.a shows the existing design for this case. In this case, a software component receives requests of the same type from multiple relay ports. There are *checker* objects behind each relay port to which incoming requests are forwarded. The checker objects set the *source* field of the requests depending on the relay port the message is received (e.g., *Checker$_1$* sets the source field to *1*). Then, the provider object can drop or accept requests from specific requesters based on the source information.

Figure 2.23.b shows the structure diagram for the updated design using an enhanced connector. The enhanced connector can discriminate between requests received from different ports using the *InPort* property. The InPort property is an ordered collection. Thus, the request stored in a particular index of the InPort property corresponds to a requester at that index. The connector behavior sets the source field of the request according to the requester who sent the request. The connector then forwards the request to the provider.

2.7.8 Case VII: Sending Requests in Round-robin Order

In the projects we have examined, we found multiple objects providing the same interface signaled in round-robin order. Since connecting a single requester to more than one port providing the same interface causes ambiguity when using standard UML ports, individual request ports were used in the design for each provider. Figure 2.24.a shows the existing design. The algorithm for round-robin sending of requests was implemented inside the requesting component, along with other responsibilities.

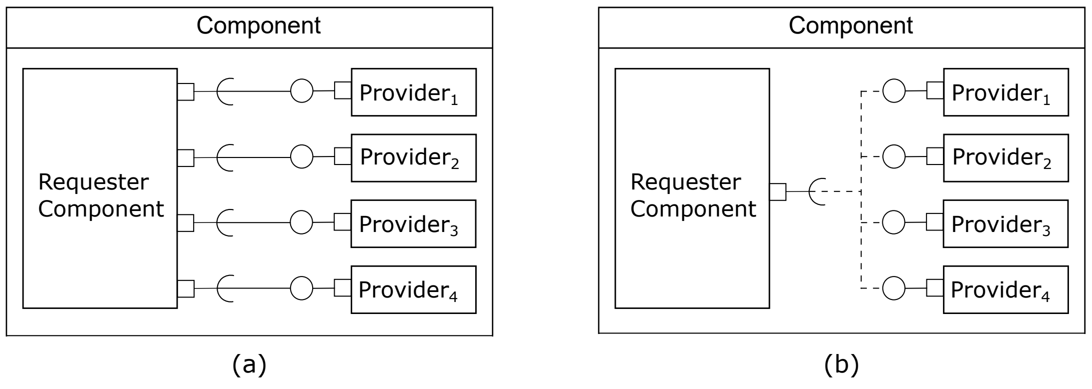


Figure 2.24: Sending requests to providers in a round-robin order: (**a**) Existing design: Requester component sends requests in round-robin order using multiple ports. (**b**) Updated design: Using an enhanced connector to send requests in round-robin order.

Figure 2.24.b shows the updated design using an enhanced connector. In the updated design, we can avoid duplicating the ports for each provider. Furthermore, we take the implementation for round-robin sending of requests outside of the requesting component. As a result, we free the requesting component from this coordination responsibility.

2.8 Handling Asynchronous Coordination

This study presented enhanced connector behavior specifications using synchronous coordination. In this section, we generalize the approach also for asynchronous coordination. When using synchronous coordination, the provider blocks the requester until it completes execution and returns with a reply. When using asynchronous coordination, the requester can continue execution while the provider handles the request.

UML implements synchronous coordination using synchronous operation calls and asynchronous coordination using signals or asynchronous operation calls. Besides the method of operation call and usage of signals, the distinction between synchronous and asynchronous coordination also depends on objects being *active* or *passive*. In UML, a passive object does not have its own thread of control. It only exhibits its behavior when one of its operations is called from the environment. In contrast, an active object has its own thread of control.

Figure 2.25 shows the sequence diagrams for synchronous and asynchronous messaging between a requester and a provider when enhanced connectors are not used. In the synchronous case shown in Figure 2.25.a, the requester waits for the reply from the provider before it can send another request. In the asynchronous case shown in Figure 2.25.b, the requester is not blocked, so it can send a second request before it receives the reply to its first request. In this case, the provider must be an active object since passive objects cannot handle requests asynchronously.

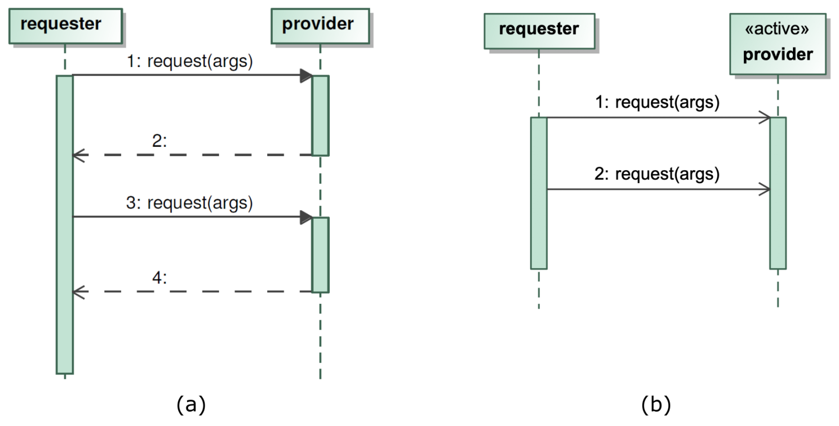


Figure 2.25: Synchronous and asynchronous handling of requests without using enhanced connectors: (**a**) Synchronous handling of requests. (**b**) Asynchronous handling of requests.

Figure 2.26 shows the sequence diagrams for synchronous and asynchronous messaging between a requester and a provider when an enhanced connector is involved. In the synchronous case shown in Figure 2.26.a, the connector acts as another passive object in between, preserving the overall synchronous behavior the requester observes when an enhanced connector is involved or not.

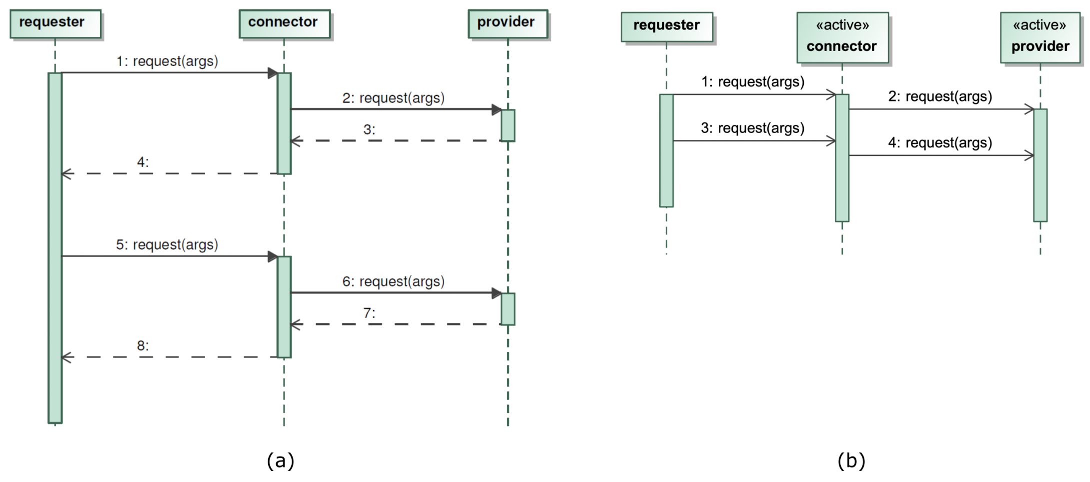


Figure 2.26: Synchronous and asynchronous handling of requests using enhanced connectors: (**a**) Synchronous handling of requests using enhanced connectors. (**b**) Asynchronous handling of requests using enhanced connectors.

In the asynchronous case shown in Figure 2.26.b[5], we design the connector as an active object which acts asynchronously. When it receives an asynchronous message, the connector does not block the requester. When the connector decides on the routing of the message, it also forwards the request to the provider asynchronously. Thus, the connector is not blocked while waiting for a reply from the provider.

As a result of describing the order of messaging in both synchronous and asynchronous cases, we showed that using a passive object as a connector in the synchronous case and using an active object as a connector in the asynchronous case allows preserving the semantics of coordination in both cases.

2.9 Relation with Protocol State Machines

UML defines the concept of *ProtocolStateMachines*, similar to the *protocol* concept in UML-RT [48] and ROOM [19]. ProtocolStateMachines allow the definition of protocols that should be followed when interaction occurs using ports. In this section, we show how enhanced connectors can be used in the presence of ProtocolStateMachines.

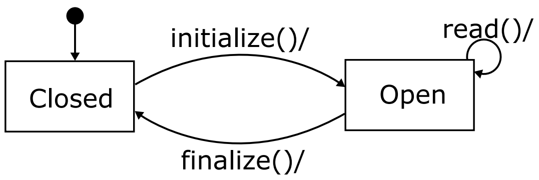


Figure 2.27: Protocol state machine example.

Figure 2.27 shows an example ProtocolStateMachine. This ProtocolStateMachine defines the order of operation calls and state changes for reading a file from the disk. The protocol enforces calling the *initialize* operation first, then calling a series of *read* operations, and then ending the communication with a *finalize* operation. When the operations *initialize*, *read* and *finalize* are declared in an interface that defines the contract of a port, this ProtocolStateMachine can enforce sending the requests in the specified order.

[5] In UML, filled arrows indicate synchronous messages, and unfilled arrows indicate asynchronous messages.

Let us assume that the ProtocolStateMachine shown in Figure 2.27 is used for reading data from a single disk. Let us also assume that we have three disks in the system, and we want to read the same data simultaneously to increase reliability. We can use an enhanced connector between the requester and three disk managers. The enhanced connector can replicate the read requests to three disks and then compare the data, eventually returning with success to the original requester only if all three read requests return the same data. This example shows how enhanced connectors and ProtocolStateMachines can be used together and indicates that the two concepts are orthogonal concepts that complement each other.

However, enhanced connectors and ProtocolStateMachines are not totally independent concepts. In the same scenario using three disks, if the requests are routed in a round-robin style by the enhanced connector, this may cause inconsistencies. We should send the *initialize* request to all disks before we can route the subsequent *read* requests to the three disks in round-robin order. Otherwise, we will send *read* requests to some disks that are not initialized. Likewise, we should replicate the *finalize* request to all three disks.

The above observation indicates that when enhanced connectors are used in the presence of ProtocolStateMachines, the connector behavior specification should satisfy the constraints of the ProtocolStateMachine that bounds the coordination among the connected ports. In this example, we can design the connector to execute different behaviors for the *initialize*, *read*, and *finalize* requests. Specifically, the connector can use *the multiple destination sender* behavior for the *initialize* and *finalize* requests and *the round-robin requester* behavior for the *read* requests. As a result, using a combination of two different connector behaviors, the constraints of the ProtocolStateMachine can be satisfied.

2.10 Discussion

This study aimed to develop an approach for the specification of connector behaviors in UML and test it by using example connectors and cases from real-life avionics software projects. Our results indicate that UML can benefit significantly from the enhanced connectors. By demonstrating the proposed approach using example connectors and the cases from real-life projects, we showed that the *contract* role in UML can be utilized to enable exogenous coordination, increasing the power of UML.

Previous studies [22], [17], [25] also emphasized the importance of exogenous coordination in different contexts and using different approaches. Since earlier versions of the UML did not have ports and connectors, the study in [17] proposed using association classes to define connector behaviors. We preferred using connectors because connectors can relate specific instances of classes. On the other hand, association relations apply to all instances of classes. The study in [25] proposes that connectors possess the whole control flow in the system to decouple components from each other. Consequently, their setup requires each component to act as passive objects defined in UML. In contrast, our findings indicate that objects should possess the control flow in the synchronous coordination case. However, in the asynchronous case, connectors should have their own control flow besides other active objects in the system.

Although we demonstrated the proposed approach using synchronous coordination problems, we also showed how the concept could be applied to asynchronous coordination. Additionally, we included an analysis that evaluates the relation of the approach with the ProtocolStateMachines in UML. ProtocolStateMachines were imported to UML-RT [18] from ROOM [19]. UML further inherited the concept from UML-RT. UML-RT defines the protocols independent from any specific context; therefore, they are reusable [49]. The connector behaviors we propose are also loosely coupled with the coordinated components since they are specified using the interfaces. Therefore, they are also reusable.

ProtocolStateMachines do not affect behavior; however, they impose constraints on the legal state transitions for associated classifiers [1]. On the contrary, enhanced connectors affect the behavior, but as a result of our analysis, we conclude that the coordination behavior specified by enhanced connectors should conform to the protocols. Unlike enhanced connectors, ProtocolStateMachines are limited to the definition of behavior for binary interaction patterns because higher-order protocols have not been addressed in UML, ROOM, or UML-RT. As a result of analyzing the relation of ProtocolStateMachines with the enhanced connectors, we conclude that the two concepts complement each other rather than rendering each other obsolete.

Using ALF to specify connector behaviors proved to be much more potent than using sequence diagrams proposed in our previous study [7]. Besides routing the requests, ALF is also suitable for defining merge behaviors. ALF was introduced as a surface notation for fUML. However, existing research [41], and the ALF specification [12] encourage its usage outside the fUML scope when execution semantics is not required. Our findings, which demonstrate using ALF to specify connector behaviors in UML models, support this use case.

Our use of reply and request buffers agrees with the existing research. The work in [26] proposes FIFO queues on component ports for aiding asynchronous communications, similar to the FIFO request queues we use in behavior specifications. FIFO ordering of messages was also practiced in [15]. Our experience in using FIFO buffers and single-copy buffers for both asynchronous and synchronous coordination supports the finding that the buffers add significant power to the method for defining the connector behaviors. The buffers enable ALF specifications to operate on the entire set of requests or replies for creating merged replies and requests.

As a result of implementing multiple complex model transformations, we conclude that QVTo [14] is suitable and mature enough for developing model transformations. Although QVTo does not support the refining mode of ATL [32] that enables making a few changes on input models, we could eliminate the limitation by programmatically capturing the results of the in-place QVTo model transformations.

Our findings indicate that UML and existing tools need to put more emphasis on n-ary connectors. Although UML includes support for n-ary connectors, most of the tools, such as MagicDraw [50], IBM Rhapsody [3], Papyrus [42], StarUML [51], and IBM Rational Rose [52], do not support them. Some of the tools only have support for n-ary associations. SysML [21] also excludes the support for n-ary connectors.

Although most of the tools exclude them, we observed that n-ary connectors were mentioned as examples in previous studies [53], [54], [55], [56], [57] and used without indication of any specific problems. Therefore, after surveying existing work, we conclude that there are not enough reasons for excluding support for n-ary connectors from the tools. Nevertheless, there are some challenges regarding n-ary associations, especially concerning the meaning of multiplicities. The work in [58] showed that the lower multiplicities are problematic. Therefore, they propose introducing inner and outer multiplicities in UML. Their proposal still needs to be incorporated into the current version of UML.

The studies in [58], [59], and [60] suggest representing n-ary relations by introducing an additional class and *n* binary relations. The fact that an n-ary connector can be represented by an object and *n* binary connectors might be the cause of why the tools do not support the n-ary connectors. We also use an object and *n* binary connectors in the E2 model to represent an n-ary connector. In the E1 model, we use *n-1* binary connectors without an object to represent an n-ary connector. Since developers will not work directly in the E2 model, it is reasonable to represent an enhanced connector in the E2 model using multiple binary connectors and an object. However, representing an n-ary connector with multiple binary connectors in the E1 model means that the behavior specification for the n-ary connector scatters into several connectors. Our solution for this problem was to specify connector behavior only for the *primary enhanced connector* and then establish dependencies from the primary enhanced connector to the other binary connectors, which we call the *secondary enhanced connectors*. Although this solution is sufficient for demonstrating the enhanced connector concepts, we conclude that lacking n-ary connector support in the tools complicates modeling efforts significantly and introduces unnecessary complexity into the models.

The motivation of our study is in parallel with Reo [22] since we specify connectors to coordinate components that are unaware of the coordination. Unlike UML connectors, Reo channels do not support message passing between components using method calls. Additionally, Reo relies on channel composition for expressing different connector behaviors. In UML, we cannot attach a connector to another connector. Therefore, true composition support for the connectors in UML is not possible without modifying the UML metamodel. Consequently, our study relies on behavior specifications to specify complex coordination logic rather than composition.

By the term "true composition support for the connectors," we mean the ability to connect the connectors to each other for building higher-level connectors. Figure

2.28.a shows an example of this kind of composition, where we use a ternary round-robin requester connector, a ternary multiple destination sender, and a binary less-frequent sender connector to build a quaternary connector.

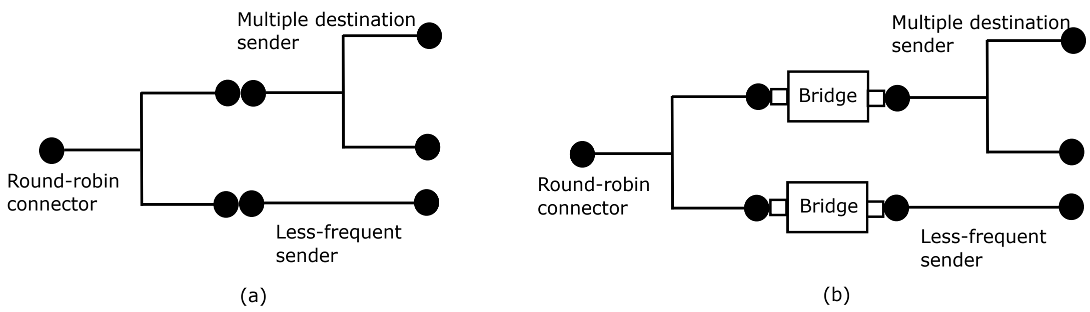


Figure 2.28: Composition of connectors for building higher-level connectors: (**a**) Composition of connectors by connecting the connectors to each other (not supported by UML). (**b**) Possible composition of connectors in UML by using bridge objects in between connectors.

Since a ConnectorEnd in UML is not a ConnectableElement, we cannot achieve the composition of connectors as shown in Figure 2.28.a. However, as shown in Figure 2.28.b, one can use objects acting as bridges that do nothing but forward the incoming requests to the other connector. A stereotype can be defined in the enhanced connector profile, which can be applied to such kinds of bridge objects. These objects can be implemented in the E2 and E3 models to trivially forward incoming requests. We leave this as a possible future extension of our study.

Our method for specifying connector behaviors with ALF covers some capabilities of Wright connectors but not all. Wright [16] connectors are capable of expressing the notion of "providing" and "using" a set of services. Additionally, Wright describes a sequence of events that should occur in the coordination. These are called the glue protocols, and using the protocols formally defined in CSP [15], Wright allows checking of the software architecture analogous to the type checking in programming languages. Using UML ports for providing and requiring a set of services and then using ProtocolStateMachines with state machine formalisms to enforce an ordering of requests can somewhat cover these capabilities of Wright. However, the focus of our approach is not checking the consistency of the architecture but providing means to specify exogenous coordination. Thus, enhanced connectors currently do not provide the required formalisms to perform such checks.

Wright also supports glue specifications which are trace specifications in CSP. A trace specification is a predicate that must hold for every trace of the glue. Since Wright borrows its semantics from CSP, it can use the parallel composition operator of CSP to argue that traces of a parallel composition must satisfy the specifications of its sub-parts. Wright allows reasoning about the behaviors of sub-parts separately. Then it allows proving that the system composed of these parts will continue to respect the properties established about the parts. This level of composability is distinct from composition support for the connectors we mentioned above. It requires a formal notation for the connector and component behaviors, which the use of UML and ALF in our approach does not currently address.

On the other hand, the glue specifications of Wright cannot handle system properties such as the timing behavior of interactions because the semantic model of CSP is not sufficient for these properties. Our method shares the same limitation because we also lack the mechanisms to define and reason about the timing properties of a real-time system. Capabilities of UML-RT [18] can be incorporated to achieve such abilities.

The cases we demonstrated from the real-life projects showed that the enhanced connector concept is applicable. In [9], we addressed the problem of efficient code generation methods for UML ports that results in minimal runtime overhead and compact code size. We can use the same code generation methods to realize the UML ports included in the E3 UML models.

The future research directions for this chapter are as follows: First, introducing composition support for the UML connectors is a valuable future research direction. The composition support can enable the building of higher-level enhanced connectors by using existing connectors. Second, we are motivated to implement the proposed approach in the "IBM Rhapsody Developer UML modeling tool" as a plugin for utilizing the approach in large-scale avionics software projects. However, we still need research on the efficient and safe realization of the fUML activities in object-oriented programming languages to use the approach in real-life DO-178C certifiable projects. Third, in the case given in Figure 2.8.b, to increase the runtime efficiency, we can perform the merge operations while caching subsequent requests. In such cases, we can design the connector behavior to employ multiple threads to perform the merge operations in parallel to caching of the requests. Future research is required to achieve this parallelization using multiple active objects that constitute the connector behavior. Finally, future work is required to define well-formedness rules for the E1 model using Object Constraint Language (OCL) [61] statements to develop a fully functional production-ready system.

2.11 Conclusions

This chapter presented a method for specifying behaviors of UML connectors for coordinating the components exogenously. We have demonstrated the effectiveness of enhancing connectors with behavioral representations as articulated further in this section.

We proposed a solution that uses ALF to specify connector behaviors. We used the *contract* role of the UML connectors for associating behaviors specified in ALF. We then used QVTo transformations to generate UML models in which fUML activities represent the connector behaviors. The resulting models also include binary connectors and classes representing the connectors. They can be transformed further into code, other platform-independent models, or platform-specific models.

We demonstrated the connector specification method and the transformations by presenting example connectors. Additionally, we presented cases from real-life large-scale avionics projects where using enhanced connectors can simplify the design, increase the reusability of the components, and help resolve ambiguities. We showed that the approach applies when using synchronous and asynchronous coordination. We also showed that the enhanced connector concept is consistent with the ProtocolStateMachine concept in UML.[6]

[6] UML models for the example connectors presented in the chapter, QVTo model transformations, the Enhanced Connector Profile, the Java application EcTransformer, and a copy of the ALF reference implementation can be accessed from the GitHub repository `https://github.com/alperkocatas/enhanced-uml-connectors/tree/v1.0.1`. Additionally, the Code Ocean capsule at `https://codeocean.com/capsule/6929314/tree/v2` allows a reproducible run and analysis of the results using a web browser.

CHAPTER 3

LIGHTWEIGHT REALIZATION OF UML PORTS FOR SAFETY-CRITICAL REAL-TIME EMBEDDED SOFTWARE

UML ports are widely used in the modeling of real-time software due to their advantages in flexibility and expressiveness. When realizing UML ports in object-oriented languages, using objects for each port is one option. However, this approach causes runtime overhead and renders significant amount of additionally generated code. To meet the performance constraints of safety-critical real-time embedded software and to decrease the costs of code reviews throughout the development, more efficient approaches are required. In this chapter, we propose an approach that introduces relatively less runtime overhead and results in more compact source code. We transform a structural model defined with UML ports into a model that uses associations instead of objects to implement the UML port semantics with fewer lines of code. We demonstrate the improvements and the validation of the proposed approach with a case study using the design of an existing avionics software project.[1]

3.1 Introduction

Defining interactions of classes using association relations that expose the entire public interface of the classes to their clients makes it hard to observe the data flow in the model. Since ports function as an opening in the encapsulation of classes through which messages are sent either into or out of the class [62], a model designed using ports is easier to understand, more flexible, easier to maintain, and more suitable for communication of design decisions. In UML, a port is a point defined by a classifier

[1] The study described in this chapter was published in Modelsward Conference in 2016 [9].

for conducting interactions between the internals of the classifier and its environment. The contract-based interaction provided by ports allows the classifiers to be defined independently from other classifiers [63].

Software for safety-critical real-time embedded systems is becoming more and more complex [64]. These systems typically require certification. Therefore the runtime overhead caused by any design decision should be justified because of the limited resources. The source code size should also be compact for less costly code reviews. For instance, DO-178C [6] defines several requirements for the certification of airborne systems. In DO-178C, if a development tool (i.e., a code generator) is not *qualified*, its outputs must be manually reviewed for correctness. Since the development tool qualification is the most rigorous type in the scope of DO-178C, often, instead of qualifying the tools, their outputs are manually reviewed. As a result, source code, which is more compact and less complex, is preferred because it is expected to reduce the effort required for manual code reviews.

Ports and connectors do not have direct correspondents in object-oriented programming languages [65], and yet, UML does not put constraints on how ports are realized [66]. The UML standard mentions ports as interaction points that provide *unique references* [1]. According to this definition, the realization of ports using objects seems adequate. However, this approach causes a certain amount of runtime overhead for the final executable [67]. Ideally, ports should have zero overhead for transmitting messages for complex real-time systems [18]. Another problem with this approach is that the source code grows significantly because of the added objects and classes to realize the ports.

In this chapter, we propose an efficient approach for mapping ports to object-oriented languages. The proposed approach enables the generation of relatively compact source code and introduces relatively less runtime overhead. We transform a source model defined using ports and connectors into a target model. We then generate code from the target model. The target model includes association relations and initialization operations, which implement the ports and connectors in the source model. We evaluate the approach using the design model for an avionics software project.

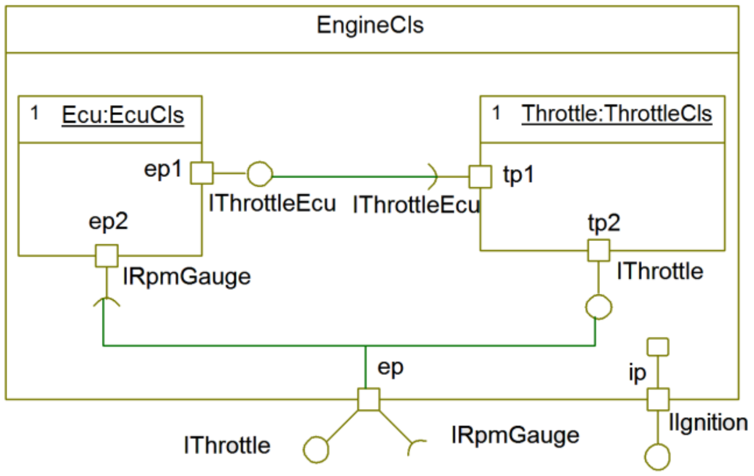


Figure 3.1: UML composite structures and port notation.

3.2 Overview of UML Ports

The abstract syntax and the concrete syntax for composite structures and ports are given in the "Structured Classifiers" and "Encapsulated Classifiers" sections of UML 2.5.1 [1]. Figure 3.1 presents an example composite structure diagram using ports. In the figure, a connector connects the ports *ep1* and *tp1*. The connector semantically indicates that a message sent from the *tp1* port of the *Throttle* object should be sent to the *Ecu* object via its *ep1* port. When sending the message, the source class (*Throttle*) specifies the port name, operation name, and operation arguments. An example expression is presented in statement (3.1) using C++:

$$GetPort(tp1).msg(args); \tag{3.1}$$

`GetPort` is used to obtain a reference to the destination of the message, and it should be translated to an appropriate statement by the port realization approach. In this chapter, connectors are categorized as *cross connectors* and *relay connectors* for better communication of ideas. Cross connectors connect ports of the two objects (i.e., the connector between ports *ep1* and *tp1* in Figure 3.1). In contrast, relay connectors connect a port of an internal part with a port of the owner class (i.e., the connector between ports *tp2* and ep in Figure 3.1).

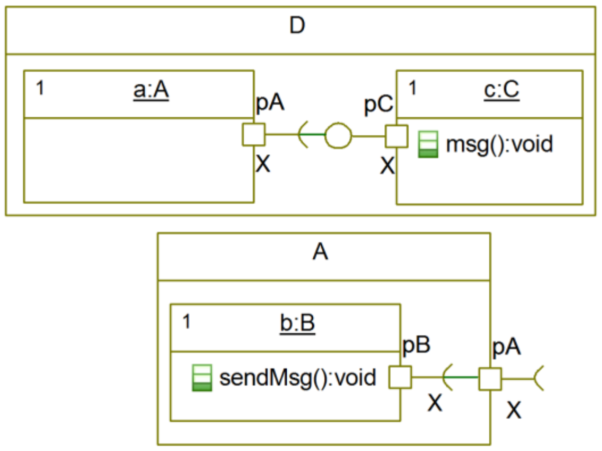


Figure 3.2: Example source model.

3.3 Approaches for the realization of ports

We use the model shown in Figure 3.2 to demonstrate the different realization approaches for ports. In the model, a message is transferred from object *b* to *c* via the path, which is formed by ports *pB, pA* and *pC*. The UML model for Figure 3.1 and the source code generated using the presented approaches are available in a public repository[2]. The C++ language is chosen as the target language, but the approaches are also applicable to other object-oriented languages.

3.3.1 Heavyweight Approach

The specific realization of ports presented in this section is implemented by one of the existing modeling tools [3]. According to this approach, each port is transformed into a class and its corresponding object. Furthermore, to differentiate messages from both directions, provided and required parts of port contracts are realized using *in* and *out* objects, which are instances of additionally generated classes. Using this approach, the example model provided in Figure 3.2 is transformed into the model shown in Figure 3.3. *PA, PB*, and *PC* are the classes generated for ports. Classes *Out* and *In* are generated for the outbound and inbound direction of ports. After the transformation, the constructor for class *C* includes:

[2] `https://github.com/alperkocatas/UmlPortStudy`

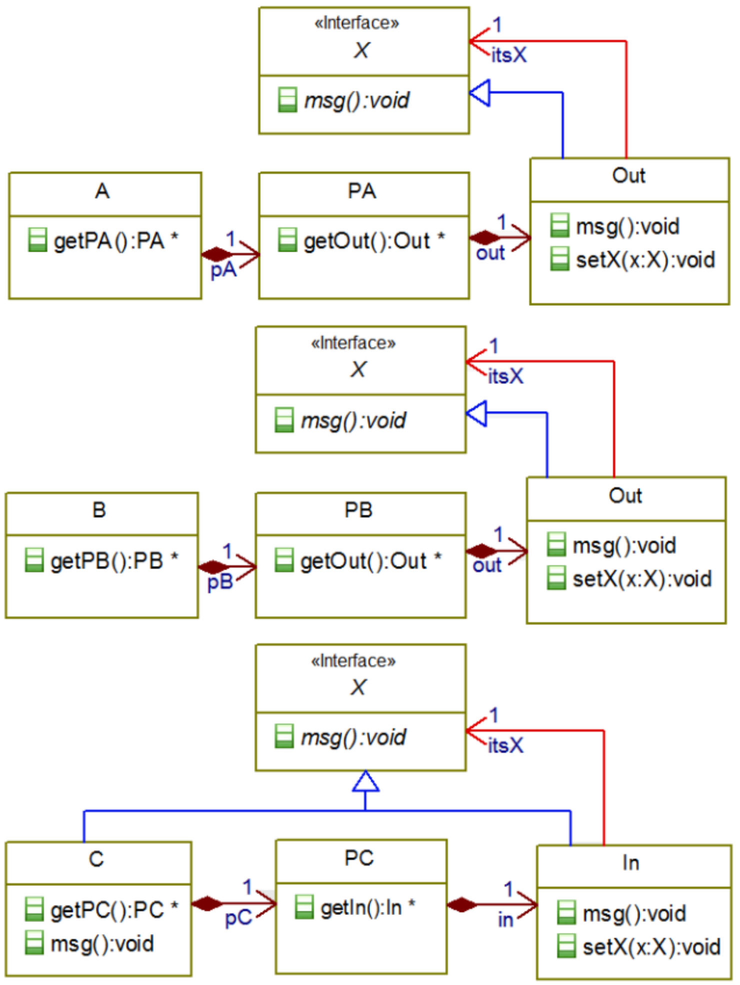


Figure 3.3: Realization of ports with heavyweight approach.

$$pC.getIn().setItsX(this); \tag{3.2}$$

Statement (3.2) is required to connect port *pC* of class *C* to object *c*. Furthermore, the constructor of class *A* is generated as:

$$b.getPB().getOut().setX(getPA().getOut()); \tag{3.3}$$

The method *setX* is a trivial accessor function for the generated association *itsX*. Statement (3.3) is required to connect the relay port *pA* of class *A* to the port *pB* of object *b*. Finally, constructor for class *D* is generated as:

$$a.getPA().getOut().setX(c.getPC().getIn()); \qquad (3.4)$$

Statement (3.4) initializes the *itsX* association of *out* object of port *pA* with the reference of the *in* object of object *c* so that messages sent from port *pA* will be handled by the *in* object of port *pC* of object *c*. `GetPort(PortName)` in statement (3.1) is translated as `getPortName()->getOut()`, which returns a reference to the *out* object of the port. This reference is used to send messages from the port.

3.4 Lightweight approach

In the lightweight realization, no objects are generated for ports. Instead, only associations with required interfaces and the operations to initialize them are generated. Relay connectors are particularly transformed into smart getter and setter methods. The smart getters and setters are used to connect the ports, which are at the ends of a chain of relay connectors. At runtime, after the initialization code generated for cross connectors runs, each object enters a state in which the final destinations of the messages that will be sent from its ports are determined. As a result, when sending messages, relay port chains spanning multiple ports are resolved in one step. Using this approach, the example model in Figure 3.2 is transformed into the model shown in Figure 3.4.

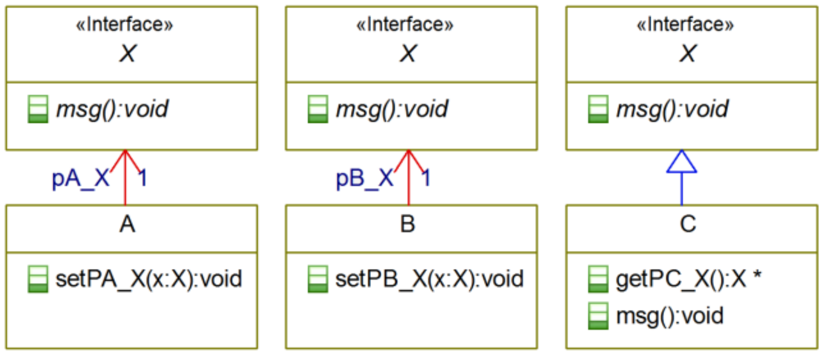


Figure 3.4: Lightweight realization of ports.

In the transformed model, associations *pA_X* and *pB_X* are used by objects *a* and *b* to send messages through their ports *pA* and *pB*. Since none of its ports requires interfaces, class *C* does not have such an association. Constructor of class *D* is generated as:

$$a.setPA\_X(c.getPC_X()); \tag{3.5}$$

Statement (3.5) is generated for the cross-connector in the source model. It initializes the *pA_X* association of object *a* with the value obtained from the *getPC_X()* operation of object *c*. Since *pC* is a behavioral port, generated operation *getPC_X()* returns *this* pointer of object *c*. The body of operation *setPA_X(X& val)*, which is a smart setter, is generated as:

$$pA\_X = val; \tag{3.6}$$

$$b.setPB\_X(val); \tag{3.7}$$

Port *pA* is a relay port that forwards messages from *pB* to *pC*. Statement (3.6) first sets the *pA_X* association of object *a* to *val*, so that object *a* can also send messages using the port *pA*. Statement (3.7) then forwards the parameter *val* to object *b*, which will set association *pB_X* of object *b* to *val*. Since the value of *val* is passed as the *this* pointer of object *c*, statements (3.6) and (3.7) effectively connect the port *pB* of object *b* with object *c*. As a result, after the expression in statement (3.5) is executed, the destination of messages going out from port *pB*, which is the object *c*, is determined. In the lightweight approach, the expression `GetPort(PortName)` in statement (3.1) is translated as `PortName_InterfaceName`, which is the name of the association created for the port and required interface pair. To resolve the interface appended to the port name, an interface that implements the called operation is searched in the required interfaces of the port. This search is performed at the time of port realization.

3.5 Disconnected Ports and Interfaces

If the source model contains disconnected ports, problems may occur at runtime. The following two cases correspond to disconnected ports: First, if a port is not connected to any other port via a connector, the port is considered as *disconnected*. Second, when there is a connector that connects two ports, if the ports at both ends of the connector do not have matching contracts, ports are considered as partially disconnected. In this chapter, the second case is denoted with the term *disconnected interfaces*.

When operations declared by the unmatched interfaces are called, messages cannot be forwarded because there is no provided interface at the opposite side of the connector. Figure 3.5 depicts an example of disconnected interfaces. The port *pA* requires interfaces *X* and *Y*, while the port *pB* provides interfaces *Y* and *Z*. Even if the ports are connected with a connector, object *a* cannot send messages declared by interface *X* to object *b* since the port *pB* does not provide interface *X* in its contract.

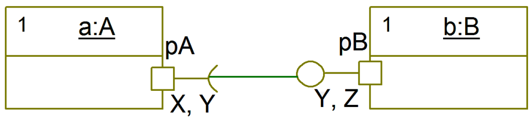


Figure 3.5: An example for disconnected ports/interfaces.

In the lightweight approach, if the source model has disconnected ports or disconnected interfaces, messages which are being sent from the generated associations may cause *null* pointer exceptions. One of the following strategies can be employed for handling disconnected ports and interfaces:

1. The lightweight approach can be used, and disconnected ports and interfaces can be allowed in the model. Then, if there is a call from disconnected ports or interfaces, the software may crash at runtime because of a *null* pointer exception. Alternatively, the model can be checked before the realization of ports to ensure that no messages will be sent using disconnected ports or interfaces during execution.

2. The lightweight approach can be used, but disconnected ports and interfaces are not allowed in the source model. The model can be checked before the realization of ports to ensure that there are no disconnected ports or interfaces.

3. The lightweight approach can be modified so that messages sent through dis-

connected ports and interfaces are checked at runtime. The messages can be ignored, or an exception can be thrown.

When the first option is used without checking the model for disconnected ports, there is a possibility of software crashing at runtime due to a *null* pointer exception. However, if statement coverage for all of the operation calls from ports is achieved while testing, it can be guaranteed that no such crash will occur. For example, DO-178C [6] requires full statement coverage by test cases beyond a certain safety-criticality level. Therefore, the effort required to achieve full statement coverage is already included in the development costs. Alternatively, instead of achieving the statement coverage, the model checking mentioned in the first option can be attempted to be incorporated. However, such model checking is not trivial since predicting the dynamic runtime behavior of software may not be feasible.

The second option is possible to implement. However, it can be argued that disconnected ports and interfaces are part of the flexibility offered by ports. For example, in Figure 3.5, although object a cannot send messages declared by interface *X*, it can be allowed to send messages declared by interface *Y*. Thus, if the operations declared by the interface *X* are not crucial for the expected behavior of object *a*, the model in Figure 3.5, which is invalid according to the second option, can be assumed as a valid one.

The third option can retain the flexibility of ports while providing graceful runtime error handling. Indeed, this option is supported by the presented heavyweight approach. Since a separate object is employed for each port in the heavyweight approach, the objects can check whether the destination of a message is *null* at runtime. As a result, one of the three options can be used. Because it allows more flexibility during modeling, the third option was selected for implementation. The next section presents how to apply the third option to extend the lightweight approach.

3.6 Extending the Lightweight Approach for Disconnected Ports

In this approach, a port with a required interface is implemented using a separate object, which can check if its generated association is *null* at runtime. Ports only having provided interfaces in their contracts are still implemented as they would be in the lightweight approach, without using objects. The generation of smart getters and setters for the relay connectors is also the same as in the lightweight approach. Using the approach, the example model presented in Figure 3.2 is transformed into the model shown in Figure 3.6.

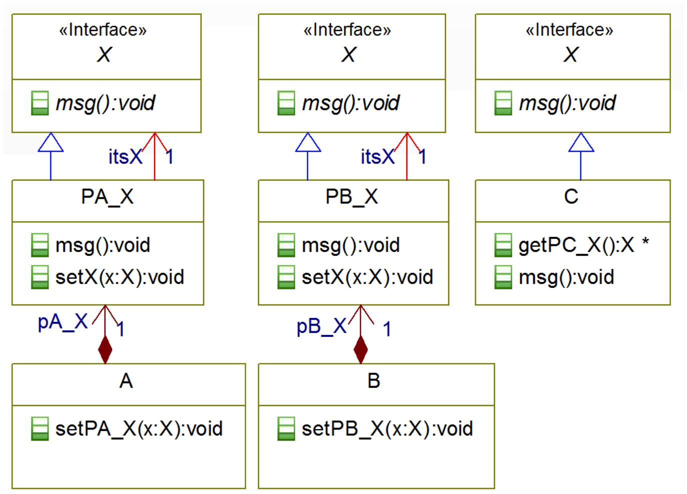


Figure 3.6: Lightweight realization of ports with checks for disconnected ports.

Associations *pA_X* and *pB_X* in the lightweight approach are generated as objects. The objects have associations with interface *X*, called *itsX*. The port connection initialization statements generated in the constructors are identical to the previous approach. However, now the *msg()* operation, which is implemented in the classes *PA_X* and *PA_B* objects, checks if the *itsX* association is *null* at runtime. In this extended version of the lightweight approach, the expression `GetPort(PortName)` in statement (3.1) is translated as `getPortName_InterfaceName()->getOut()`.

3.7 Evaluation

3.7.1 Performance Analysis

One of the critical performance drawbacks of ports originates from messages redirected through multiple ports along a chain of relay ports. The lightweight approach ensures that at runtime, the objects hold a reference to the final destinations of the messages. This approach routes the messages to their final destinations in one step. In the heavyweight approach, messages cannot reach their destinations in one step, but they are redirected multiple times between ports along the chain of relay ports. With the extension of the lightweight approach for checking disconnected ports, we still expect the performance to be better than the heavyweight approach because we only add one level of redirection.

Avoiding unnecessary message redirections may also yield faster performance due to the decrease in the number of instruction pipeline operations. Smaller code size is also expected to improve cache utilization, thus improving the runtime performance. Furthermore, the computation required to create objects during initialization is another potential drawback for the heavyweight approach. We expect the lightweight approach and its extension to perform better since the number of created objects is zero or, at least, fewer.

3.7.2 Code Size Analysis

We expect the lightweight approach to deliver the most compact code among the presented approaches because we do not create any objects and classes for ports. When we add the checks for disconnected ports and interfaces, the code size should still grow less than it would in the heavyweight approach. This is because we only generate objects for the outbound directions of the ports. In the heavyweight approach, the resulting code size is considerably larger due to implementing the operations declared by the provided and required interfaces in every class generated for ports.

3.7.3 Case Study Design

In order to compare the results from both approaches, we used a previously released version of an avionics software project. The software was developed by a team of fifty software engineers within a five-year schedule and is still in progress. The software coordinates over thirty avionics devices and provides the pilot with crucial flight information. IBM Rhapsody [3] is used as the development tool. The project requires DO-178C [6] compliance. Because the code generator of IBM Rhapsody is not a *qualified* development tool in the scope of DO-178C, generated code is reviewed manually.

Software components in the case study run on an ARINC 653 [46] compliant real-time operating system. Software processes run in time and space-isolated partitions. Partitions are scheduled on a fixed, cyclic basis. Partitions perform their initialization tasks and then start executing a periodic running task. When the allocated execution duration finishes for the scheduled partition, the scheduler switches to the next partition in the schedule.

For comparison purposes, we generated the code using different approaches. We collected several metrics during code generation and runtime. AVGRT metric indicates the average time, in milliseconds, required for a partition to finish its periodic execution. The LSLOC metric indicates the logical source lines of code measured using the Unified Code Count tool [68]. LSLOC is not affected by style and formatting decisions. We define the OVRHD metric as the difference between LSLOC measurements for the source code with and without port realization. Thus, the OVRHD metric indicates the increased LSLOC caused by port realization. The TGEN metric indicates the time, in seconds, required to generate code from the model. SBIN metric indicates the size of the final executable binaries in megabytes.

3.7.4 Results and Discussion

Figure 3.7 shows charts corresponding to the measurements for given metrics according to different approaches. The lightweight approach dramatically reduced the LSLOC from 316,195 to 181,390. The 42% decrease in source code size may provide significant cost savings during code reviews.

Similarly, the OVRHD metric is the lowest for the lightweight approach. When the checking for disconnected ports is incorporated, OVRHD is still significantly less than the OVRHD measured for the heavyweight approach.

According to the TGEN measurements, the fastest code generation was achieved with the lightweight approach, followed by the lightweight approach with checks. SBIN metric measurements showed that the resulting binaries were most compact with the

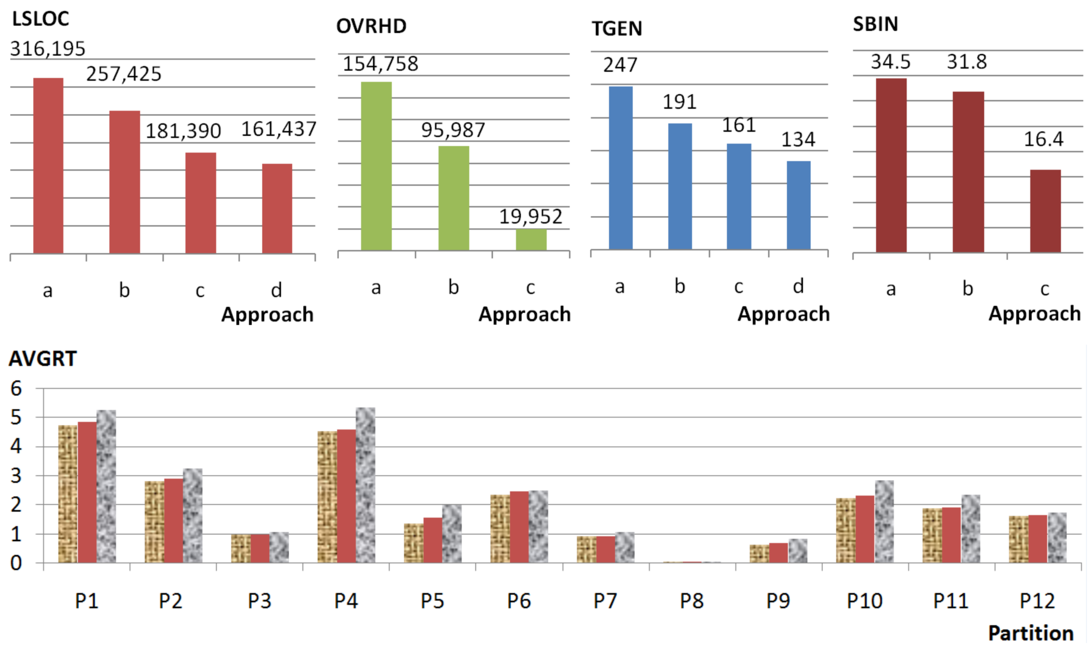


Figure 3.7: LSLOC, OVRHD, TGEN, SBIN and AVGRT metrics - Approaches: (a) Heavyweight approach, (b) Lightweight approach with checks for disconnected ports, (c) Lightweight approach, (d) None (ports are not realized). AVGTR metric - Left bar: Lightweight approach., Middle bar: Lightweight approach with checks., Right bar: Heavyweight approach.

lightweight approach.

After generating and compiling the source code, we executed the resulting binaries. The software ran successfully. We demonstrated the verification of the build by running a subset of the test cases used in the formal software release. We did not observe any *null* pointer exceptions during runtime due to disconnected ports, even with the lightweight approach. This was not surprising since the previous release of this software was already tested, and complete statement coverage was achieved for statements used to send messages from ports.

While collecting the AVGRT metrics, we activated an identical set of functions to generate an identical load on the system. According to the results, we observed up to 32% improvement for the AVGRT metrics. When we averaged the runtime improvements for each partition, we observed 15.7% overall performance improvement with the lightweight approach compared to the heavyweight approach. A similar calculation revealed an average of 10.9% performance improvement by the lightweight approach with disconnected port checking compared to the heavyweight approach. The variation in improvement percentages for each partition is due to the different levels of port usage and composite structure hierarchies. The results showed that using the lightweight approach and its extension yields more compact code and better performance than the heavyweight approach.

3.8 Related Work

The concept of ports is also available in other modeling languages. UML-RT [18] and MARTE [69] are the UML extensions for modeling safety-critical embedded real-time software. In UML-RT, a protocol defines which kind of messages can be received and sent from a port. In UML, the provided and required properties of ports capture the information captured by the protocol concept. On the other hand, MARTE categorizes ports as client/server ports and flow ports. Client/server ports are syntactic sugar over UML ports, enabling a more convenient way to define the port contracts. The flow ports are used to model the data flow between structured components. Flow properties and flow specifications are used to define the messages that can flow through the ports. If flow specifications and properties are mapped to interfaces, flow ports can be realized using the approaches presented in this chapter. Otherwise, they may need different realization approaches. Based on the purpose of port representation and realization, such alternative languages do not offer additional advantages. Consequently, we have exploited UML due to its wider usage and access to vast project material.

UML ports and composite structures are mentioned in previous studies that use ports to model embedded systems. The ports are mapped to target languages such as SystemC [70], [71], VHDL [72] and Simulink [73]. In these studies, one-to-one mappings between UML ports and the target languages could be performed since the target languages provided the constructs that correspond to UML ports.

For mapping ports into object-oriented programming languages, some studies suggest mapping ports using objects [74]. The heavyweight approach presented in this chapter is employed by IBM Rhapsody [3]. To cope with the performance degradation, IBM Rhapsody offers an optimization performed at runtime to find the final destinations of ports [67]. The optimization runs during the initialization of software and uses algorithms to traverse the relay connector chains to find the ultimate targets of the messages. However, the additional computation during initialization is costly, and using particular data structures makes the code even more complex and harder to review.

The possibility of lightweight realization for ports was mentioned previously in [75]. The study argues that we can realize the ports in a lightweight fashion, with no port objects created at runtime. However, the study did not present a specific method for the mentioned lightweight realization of ports.

Another approach for the realization of ports [65] is very similar to the lightweight approach presented in this chapter. However, the approach creates the getter methods using only the name of the ports without utilizing the interface names. This naming scheme cannot cope with cases where a source port requires more than one interface in its contract, and the port is connected to more than one destination port, each providing one of the different interfaces required by the source port. Moreover, the validity of the approach was not demonstrated by a case study or by using any other means.

3.9 Conclusions

This chapter proposed a lightweight approach for mapping UML ports to object-oriented programming language constructs. We first presented a widely used method for realizing UML ports that is prone to performance and code size problems. Afterward, we presented the lightweight approach that enables the use of ports without sacrificing runtime performance and source code size. Additionally, we discussed the problems which disconnected ports in the source model may cause. We then introduced an extension to the lightweight approach for handling disconnected ports.

We compared the presented approaches using metrics collected from a real-life case study. Metrics used for comparison are logical source lines of code, average runtime performance, model transformation duration, and binary size. The case study showed that the proposed lightweight approach results in more efficient and compact code. Additionally, the time required for code generation and the size of executable binaries produced after compilation were lower when using the proposed approaches. Better performance yields more headroom for meeting hard real-time requirements, while smaller and less complex code enables relatively more effortless and more accurate code reviews, potentially improving safety.

CHAPTER 4

CONCLUSIONS

This thesis presented our research and studies on improving the usage of UML ports and connectors. We used safety-critical real-time avionics software projects for our case studies. However, the thesis findings also apply to similar domains where it is required to precisely specify the software design to aid automatic code generation and verification. This chapter includes short summaries of each chapter, remarks, and possible future work.

4.1 Summary

In Chapter 2, we introduced the concept of enhanced UML connectors, which are connectors with associated behaviors. We used ALF for the specification of the connector behaviors. Then we used QVTo to transform the UML models that include the connector behaviors into platform-independent UML models, which include fUML activities representing the connector behaviors. The resulting models can be transformed into other models or code.

We used four example connectors to demonstrate the connector specification method: The round-robin requester connector, the multiple destination sender, the less-frequent sender, and the request barrier. We tested the QVTo transformations on the four example connectors to generate the platform-independent models of connector behaviors describing the required coordination logic. Additionally, we searched existing real-life large-scale avionics software projects to find the cases where the enhanced connector concept can be applied. Our findings show that the concept is applicable and can improve the design significantly. We also showed how to use the approach in synchronous and asynchronous coordination. Finally, we clarified the relation of enhanced connectors with the existing concept of ProtocolStateMachines in UML.

In Chapter 3, we presented a lightweight approach for realizing UML ports in object-oriented programming languages. We first showed the heavyweight approach that existing tools have utilized for realizing UML ports. The heavyweight approach implements the ports with individual classes and their objects. We then introduced the lightweight approach that uses associations instead of dedicated objects for the ports. We finally presented a variation of the lightweight approach that can gracefully handle disconnected ports.

We compared the three approaches for their runtime performance, code size, code generation time, and compiled binary size. The results showed that the lightweight approach and its extension that handles disconnected ports excelled at generating code that runs significantly faster than the code generated by the heavyweight approach. The size of the generated code and the resulting binaries was also considerably smaller. The time required for code generation was also lower than the heavyweight approach.

4.2 Remarks

The purpose for associating behaviors with the connectors is three-fold. First, ambiguities that may arise when n-ary connectors are used in UML can be resolved using the associated connector behaviors that specify the routing of the requests between multiple requesters and providers. Second, since the connectors take on the coordination responsibility, the functions assigned to the connected components are reduced, and therefore, their reusability can increase. Finally, the connector behaviors can be reused in scenarios with similar coordination requirements.

Our findings show that we can successfully resolve the ambiguities in UML that the usage of n-ary connectors can cause. When an n-ary connector connects a single requester to multiple providers, it is unclear in UML which providers receive the requests and in which order. Using ALF, we could specify connector behaviors that precisely describe the destination of the requests in such cases. Aside from the definition of the providers which will receive the requests, we also defined rules for merging the replies from multiple providers and replying to the requester with a single reply. The ability to route the requests from a single requester to multiple providers and then merge the replies from multiple providers enabled exogenous coordination of the requester and multiple providers. The exogenous coordination is provided in the sense that the requester is unaware of the fact that its requests are being routed to more than one destination. Moreover, since the connector also merges multiple replies received as one reply, the requester does not know that it is receiving replies from multiple providers. This decouples the requester from the number of providers, the algorithm that routes the requests to multiple destinations, and the algorithm to evaluate multiple replies.

Additionally, in cases where an n-ary connector connects multiple requesters, we could design connector behaviors in ALF that can collect requests from each requester, merge them and send them to the requesters. Collection of the requests from multiple requesters and sending them as a single request to the provider also provided exogenous coordination in such cases.

We validated the enhanced connector approach with the help of example connectors developed using the proposed method. The four example connectors developed using the proposed approach demonstrated the expressive power of ALF and the buffers used in the definition of connector behaviors. Using QVTo transformations, we transformed the E1 level models that include the connector behavior specifications into the E3 level models, which include fUML activities as connector behaviors.

Since ALF borrows its semantics from fUML, elements outside of the fUML scope, such as ports and connectors, are not allowed. By using ALF only to precisely define the behaviors of the connectors, our study demonstrated using ALF in the context of UML models not limited to the fUML subset. We achieved this with the help of a model transformation that extracts the ALF specifications from the E2 level model. We then compiled the extracted ALF specifications using the ALF reference implementation. Then we merged the fUML models output by the ALF compiler back to the E2 level model to get the final E3 level model.

As a second level of validation, we identified real-life examples from large-scale avionics projects where enhanced connectors can simplify design and increase reusability. Our search inside the projects using a custom-developed tool revealed numerous cases in which we can apply the enhanced connector concept. In the mentioned cases, the designs of the components ended up being simpler when we incorporated enhanced connectors for the coordination tasks. Since their responsibilities decrease, we also expect their reusability to increase. In other cases, using enhanced connectors eliminated existing intermediate objects designed to perform exogenous coordination. In such cases, using enhanced connectors simplified the overall design.

The example connectors and the cases from real-life projects supported the validation for the enhanced connectors. However, to fully validate the approach, we should establish efficient realization approaches for the fUML activities representing the connector behaviors. Then, we can generate executable code from the models built using UML ports, regular connectors, and enhanced connectors with behavior specifications. This can enable field testing of the approach, followed by further improvement due to user feedback.

Using ALF to specify connector behaviors proved to be much more beneficial than using sequence diagrams proposed in our previous study [7]. Using sequence diagrams was adequate for routing the requests. However, the behaviors for merging requests and replies were not possibly expressed using sequence diagrams. Therefore, in our previous study, we used the target programming language (e.g., C++) to define such merge behaviors. That, unfortunately, weakened the approach because the resulting specifications were not platform-independent. Using ALF both for the routing of the requests and the specifications of merge behaviors improved the approach significantly, allowing platform-independent specifications.

As a result of our research on n-ary connectors, we concluded that most modeling tools do not support them, but there need to be more reasons to exclude the support. We completed our study by representing n-ary connectors with *n-1* binary connectors. However, our findings indicate that the lack of support for n-ary connectors complicates the modeling efforts, making the design models unnecessarily complex.

When UML's support for the n-ary connectors is implemented in the tools, the "primary enhanced connector" and the "secondary enhanced connector" concepts that we introduced in Chapter 2 will not be required since the behavior specifications can be associated with a single n-ary connector in that case. The example connectors and the cases from real-life projects support the finding that having the n-ary connector support in the tools and the ability to define behaviors for the connectors in ALF will significantly increase the power of UML. Using a code generation scheme for fUML activities that describe the behaviors of the connectors, the approach can be employed in the design of software systems, not limited to the avionics software.

Our study regarding the lightweight realization of UML ports augments the enhanced UML connectors by providing efficient methods for transforming models that contain connectors and ports into object-oriented programming languages. While the enhanced connector concept is just taking off and lacks a code generation scheme for the connector behaviors at the current time, we have used the lightweight realization of UML ports to generate code for the last six years in most large-scale avionics software projects in Aselsan.

Although it is impossible to figure an exact number, the code generated by the lightweight realization for UML ports has been flying hundreds if not thousands of flight hours in fixed and rotary-wing aircraft. Thanks to the more straightforward and smaller code, it saved tons of effort in manual code reviews. Additionally, the simpler code structure saved a significant amount of effort while performing the structural coverage analyses of the software as a part of verification. Finally, the approach has enabled better utilization of hardware resources since it improved the runtime performance. Consequently, using the lightweight realization approach has significantly reduced the overhead of extensive use of UML ports and connectors.

4.3 Future Work

The future work for the enhanced connector study includes the development of a production-ready environment that we can use in avionics software development. We plan to implement the approach as a plugin for the "IBM Rhapsody Developer UML modeling tool" [3]. However, we need further research for the efficient realization of the fUML activities representing the connector behaviors in object-oriented languages to develop such a system. We also need to establish well-formedness rules using the Object Constraint Language (OCL) [61]. The rules can detect invalid connector configurations and erroneous behavior descriptions.

We implemented the lightweight realization for the UML ports as a plugin for the "IBM Rhapsody Developer UML modeling tool" in Java programing language. As a future work, we can use a standard model transformation language such as QVT[14] to develop the model-to-model transformations, which transform the models that include UML ports and connectors, into models that include associations and links. Doing so will enable the approach to be used in other platforms supporting standard model transformations.

REFERENCES

[1] Unified Modeling Language, Version 2.5.1, December 2017. Available online: `www.omg.org/spec/UML/2.5.1` (accessed on 22 November 2022).

[2] Li, R., Liang, P., Soliman, M., Avgeriou, P., Understanding Software Architecture Erosion: A Systematic Mapping Study, Journal of Software: Evolution and Process, 2022.

[3] IBM Rhapsody, Developer Edition. Version 8.1.1. Available online: `www.ibm.com` (accessed on 22 November 2022).

[4] Kocatas, A., Pala, N. E., Koksal, O., ARINC 653 Uyumlu Aviyonik Sistemlerin Yapılandırma Dosyalarının Model Güdümlü Geliştirilmesi, Ulusal Yazılım Mühendisliği Sempozyumu (UYMS), Ankara, Turkey, September 2011.

[5] Kocatas, A., Durmus, U., Tufaner, M., Senlet, T., Model Based, ARINC 653 Compatible Avionics Software Development and Integration, 1. Avionics and System Integration Symposium, Ankara, Turkey, 2010.

[6] DO-178C, Software Considerations in Airborne Systems and Equipment Certification, RTCA, December 2011.

[7] Kocatas, A., Dogru, A. H., Enhancing UML Connectors with Behavioral Specifications, in IEEE/ACIS 20th International Conference on Software Engineering Research, Management and Applications (SERA), Las Vegas, NV, USA, 2022 pp. 16-21.

[8] Kocatas, A., Dogru, A. H., Enhancing UML Connectors with Behavioral ALF Specifications for Exogenous Coordination of Software Components, Applied Sciences. 2023; 13(1):643.

[9] Kocatas, A., Can, M., Dogru, A. H., Lightweight Realization of UML Ports for Safety-Critical Real-Time Embedded Software, 4th International Confer-

ence on Model Driven Engineering and Software Development (Modelsward), 2016.

[10] Stahl, T., Völter, M., "Model-Driven Software Development: Technology, Engineering, Management", Wiley, 1st edition, ISBN: 978-0-470-02570-3, May 19, 2006.

[11] Allen, R., Garlan, D., A formal basis for Architectural Connection, ACM Transactions on Software Engineering and Methodology, Volume 6, Issue 3, July 1997 pp 213–249.

[12] Action Language for Foundational UML (Alf), Version 1.1, June 2017. Available online: `www.omg.org/spec/ALF/1.1` (accessed on 22 November 2022).

[13] Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.5, June 2021. Available online: `www.omg.org/spec/FUML/1.5/` (accessed on 22 November 2022).

[14] MOF Query/View/Transformation, Version 1.3, June 2016. Available online: `www.omg.org/spec/QVT/1.3` (accessed on 22 November 2022).

[15] Hoeare, C. A. R., "Communicating Sequential Processes", Prentice-Hall, Inc. ISBN: 978-0-13-153271-7, 1985.

[16] Allen R. J., A Formal Approach to Software Architecture, Ph.D. Thesis, Carnegie Mellon University, Technical Report Number: CMU-CS-97-144, May 1997. Available online: `https://www.cs.cmu.edu/~able/paper_abstracts/rallen_thesis.htm` (accessed on 22 November 2022).

[17] Ivers, J., Clements, P., Garlan, D., Nord, R,. Schmerl, B., Silva, O., Documenting Component and connector views with UML 2.0, Technical Report. Carnegie Mellon School of Computer Science, Software Engineering Institute. CMU/SEI-2004-TR-008, 2004. Available online: `http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7095` (accessed on 22 November 2022).

[18] Selic, B., Using UML for Modeling Complex Real-Time Systems, Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems. Springer-Verlag, 1998.pp. 250-260.

[19] Selic, B., Gullekson, G., Ward, P. T., “Real-Time Object-Oriented Modeling (ROOM)”, Wiley Professional Computing, 1994.

[20] Object Management Group, MARTE Tutorial, November 2007, Version 1.1. Available online: `www.omg.org/omgmarte/Tutorial.htm` (accessed on 22 November 2022).

[21] OMG System Modeling Language (SysML), Version 1.6, 2019. Available online: `www.omg.org/spec/SysML/1.6` (accessed on 22 November 2022).

[22] Arbab, F., Mavaddat, F., Coordination through channel composition, in Proceedings of COORDINATION 2020: Coordination Models and Languages, 5th International Conference, YORK, UK, 2002.

[23] Arbab F., What do you mean, coordination?, Bulletin of the Dutch Association for Theoretical Computer Science (NVTI), March 1998. Available online: `https://homepages.cwi.nl/~farhad/Papers/NVTIpaper.pdf` (accessed on 22 November 2022).

[24] Arbab, F., Reo: A Channel-based Coordination Model for Component Composition, Mathematical Structures in Computer Science, 2004, 14(3), 329-366.

[25] Lau K., Ornaghi M., Wang Z., A Software Component Model and its Preliminary Formalization, FMCO 2005: Formal Methods for Components and Objects pp 1-21. Lecture Notes in Computer Science (LNCS, volume 4111), 2005.

[26] Janisch S., Behaviour and Refinement of Port-Based Components with Synchronous and Asynchronous Communication, Phd Thesis, Universidade Nova de Lisboa, 1999. Available online: `https://edoc.ub.uni-muenchen.de/12075/1/Janisch_Stephan.pdf` (accessed on 22 November 2022).

[27] Quentin R., Brahim H., Jason J., Formal specification and verification of reusable communication models for distributed systems architecture, Future Generation Computer Systems, Volume 108, 2020, Pages 178-197.

[28] Jackson D., “Software Abstractions: Logic, Language, and Analysis”, The MIT Press (2006).

[29] Araújo, C., Batista, T., Cavalcante, E., Oquendo, F., Generating Formal Software Architecture Descriptions from Semi-Formal SysML-Based Models: A Model-Driven Approach, International Conference on Computational Science and Its Applications. Springer, Cham, 2021.

[30] Oquendo, F., π -ADL: an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures, ACM SIGSOFT Software Engineering Notes 29(3), 1–14 ,2004.

[31] Oquendo, F., Leite, J., Batista, T. “Software Architecture in Action: Designing and Executing Architectural Models with SysADL Grounded on the OMG SysML Standard”, UTCS, Springer, Cham, 2016, ISBN:978-3-319-44339-3.

[32] Frédéric J., Freddy A., Jean B., Ivan K., ATL: A model transformation tool, Science of Computer Programming, Volume 72, Issues 1–2, 1 June 2008, Pages 31-39.

[33] Dogru A. H., Tanik M. M., A process model for component-oriented software engineering. IEEE software, 20(2):34–41, 2003.

[34] Kaya, M. C., Cetinkaya, A., Dogru, A. H., Off-the-Shelf Connectors for Interdisciplinary Components. Transactions of the SDPS: Journal of Integrated Design and Process Science, 22 (3), 2018, 35–53.

[35] Kaya, M. C., Saeedi N. M., Suloglu, S., Tekinerdogan, B., Dogru, A. H., Managing Heterogeneous Communication Challenges in the Internet of Things Using Connector Variability, in Connected Environments for the Internet of Things: Challenges and Solutions, Mahmood, Zaigham Springer, International Publishing, 2017, pages 127–149.

[36] Cetinkaya, A., Kaya, M. C., Dogru, A. H., Enhancing XCOSEML with Connector Variability for Component-Oriented Development, Proceedings of the

SDPS 21st International Conference on Emerging Trends and Technologies in Designing Healthcare Systems, Orlando, FL, USA, 2016, pages 120–125.

[37] Kaya, M. C., Suloglu, S, Dogru, A. H., Variability Modeling in Component Oriented System Engineering. In The 19th International Conference on Transformative Science and Engineering, Business and Social Innovation, Kuching, Sarawak, Malaysia 2014.

[38] Kaya, M. C., Cetinkaya, A., Dogru, A. H., Composition Capability of Component-Oriented Development, 5th International Symposium on Innovative Technologies in Engineering and Science (ISITES), Baku, Azerbaijan, 2017, pages 1299–1307.

[39] Cetinkaya, A., Kaya, M. C., Dogru, A. H., Component Integration Through Connector Supported Process Models, Proceedings of the SDPS 22nd International Conference on Emerging Trends and Technologies in Convergence Solutions, Birmingham, AL, USA, 2017, pages 31–37.

[40] Çetinkaya, A., Karamanlıoğlu, A., Kaya, M. C., Dogru, A. H., Eşgüdümlü Bileşenler ile Birleştirme Yaklaşımı, Ulusal Yazılım Mühendisliği Sempozyumu, UYMS 2017.

[41] Buchmann T., Prodeling with the Action Language for Foundational UML, ENASE 2017.

[42] Eclipse Papyrus Modeling Environment, Available online: `www.eclipse.org/papyrus` (accessed on 22 November 2022).

[43] Eclipse IDE, The Eclipse Foundation, Available online: `www.eclipse.org/ide` (accessed on 22 November 2022).

[44] ALF Reference Implementation, Model Driven Solutions, Inc. 2020. Available online: `https://modeldriven.github.io/Alf-Reference-Implementation` (accessed on 22 November 2022).

[45] Rierson, L., "Developing Safety-Critical Software, A Practical Guide for Aviation Software and DO-178C Compliance", CRC Press, 2013.

[46] Avionics Application Software Standard Interface: ARINC Specification 653P1-3, Required Services. Aeronautical Radio, Inc. 2015.

[47] DO-297, Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations, Aeronautical Radio, Inc., November 2005.

[48] Posse. E., Dingel, J., An Executable Formal Semantics for UML- RT, Software & Systems Modeling, vol. 15, no. 1, pp. 179–217, February 2016.

[49] Selic, B., Turning Clockwise: Using UML in the Real-Time Domain, Communications of the ACM, October 1999.

[50] MagicDraw Modeling Tool, No Magic Inc. Version 19.0. Available online: `www.magicdraw.com` (accessed on 22 November 2022).

[51] StarUML Modeling Tool. MKLabs Co., Ltd. Version: 5.0.2. Available online: `https://staruml.io` (accessed on 22 November 2022).

[52] IBM Rational Rose Enterprise, Version: 7.0.0.4. Available online: `www.ibm.com` (accessed on 22 November 2022).

[53] Proenca J., Clarke D., Data Abstraction in Coordination Constraints, Communications in the Computer and Information Science 393:159-173, European Conference on Service-Oriented and Cloud Computing, 2013.

[54] Wermelinger M., Lopes A., Fiaderio J. L., Superposing Connectors, Tenth International Workshop On Software Specification and Design, 2000.

[55] Kumar, B., M., Srikant, Y. N., Lakshminarayanan R., On the Use of Connector Libraries in Distributed Software Architectures, ACM SIGSOFT Software Engineering Notes, 2002.

[56] Wermelinger M., Fiaderio J. L., Towards an Algebra of Architectural Connectors: A Case Study on Synchronization for Mobility, Ninth International Workshop on Software Specification and Design, 1998.

[57] Wermelinger M., Specification of Software Architecture Reconfiguration, (Phd Thesis), 1999. Available online: `http://hdl.handle.net/10362/1137` (accessed on 22 November 2022).

[58] Genova G., Llorens J., Martinez P., The meaning of multiplicity of n-ary associations in UML, Software and Systems Modeling 1(2):86-87, 2002.

[59] Dragan M., "Model-Driven Development with Executable UML", ISBN:978-0-470-48163-9, Wrox; 2009.

[60] Brambilla M., Fraternali P., Domain Modeling, In: Interaction Flow Modeling Language, December 2015.

[61] Object Constraint Language, Version 2.4, February 2014. Available online: `www.omg.org/spec/OCL/2.4` (accessed on 22 November 2022).

[62] Bjerkander, M., Kobryn, C. 2003. Architecting Systems with UML 2.0, Software, IEEE Volume:20 , Issue: 4, pp. 57 - 61.

[63] Selic, B., Architectural Patterns for Real-Time Systems, In: Lavagno, L., Martin, G., Selic., B. ed. UML For Real. Kluwer Academic Publishers, Norwell, MA, 2003, USA 171-188.

[64] McDermid, J., Kelly, T., Software in Safety Critical Systems: Achievement and Prediction, Journal of The British Nuclear Energy Society, 2006, Vol. 02, pp. 140-146.

[65] Mraidha, C., Radermacher, A., Gerard, S., Model-Based Deployment and Code Generation. In: Hugues, J., Canals, A., Dohet, A., Kordon., F. ed. Embedded Systems: Analysis and Modeling with SysML, UML and AADL, Wiley, 2013.

[66] France, R., B., Ghosh, S., Dinh-Trong, T., Solberg, A. 2006. Model-Driven Development Using UML 2.0: Promises and Pitfalls, Computer, Vol. 39, pp 59-66.

[67] Douglass, B., P., Systems Architecture, In: Real Time UML Workshop for Embedded Systems, ISBN: 978-0-7506-7906-0. Newnes, 2007, pp:90-92.

[68] Nguyen, V., Deeds-Rubin, S., Tan, T., Boehm, B., A SLOC Counting Standard, COCOMO II Forum.

[69] UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.1, OMG, 2011. Available online: `http://www.omg.org/spec/MARTE/1.1` (accessed on 22 November 2022).

[70] Andersson, P., UML and SystemC – A Comparison and Mapping Rules for Automatic Code Generation. In:Villar, E. Ed. Embedded Systems Specification and Design Languages. Springer Netherlands, 2008.

[71] Xi, C., Hua, L., J., Zucheng, Z., YaoHui, S., Modeling SystemC design in UML and automatic code generation, Proceedings of the 2005 Asia and South Pacific Design Automation Conference, pp 932-935.

[72] J. Vidal, F., Lamotte, G., Gogniat, P. Soulard, Diguet, J. P., A co-design approach for embedded system modeling and code generation with UML and MARTE, Design, Automation & Test in Europe Conference & Exhibition, 2009.

[73] Brisolara, L., B., Oliveira, M., F., S., Redin, R., Lamb, L., C., Wagner, F., Using UML as Front-end for Heterogeneous Software Code Generation Strategies, Design, Automation and Test in Europe, 2008, pp.504,509.

[74] Willersrud, A., User-defined Code generation from UML 2.0. (M.Sc. Thesis). 2006. Permanent link: `http://urn.nb.no/URN:NBN:no-12371`.

[75] Bock, C., UML 2 Composition Model. Journal of Object Technology, Vol. 3, No. 10, 2004.

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Kocataş, Alper Tolga

EDUCATION

Degree	Institution	Year of Graduation
M.Sc.	Computer Engineering, Koç University	2005
B.Sc.	Computer Science, Bilkent University	2002
High School	Samsun Anadolu Lisesi	1998

PROFESSIONAL EXPERIENCE

Compant/Institution	Title	Year
Aselsan	Senior Lead Software Engineer, Avionics Software	2008-2022
Meteksan Sistem	Senior Software Engineer	2007-2008
Gurmen Group	Software Engineer	2006-2007
Argela Technologies	Software Engineer	2005-2006
Koç University	Research and Teaching Assistant	2002-2005

PUBLICATIONS

- Kocatas. A. Dogru, A. H., Enhancing UML Connectors with Behavioral ALF Specifications for Exogenous Coordination of Software Components, Applied Sciences. 2023; 13(1):643.
- Kocatas A., Dogru, A.H., Enhancing UML Connectors with Behavioral Specifications, in 2022 IEEE/ACIS 20th International Conference on Software Engineering Research, Management and Applications (SERA), Las Vegas, NV, USA, 2022 pp. 16-21.
- Kocatas A., Can, M., Dogru, A. H., Lightweight Realization of UML Ports for Safety Critical Real Time Embedded Software, 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), 2016.
- Ralph, P., Exman, I., Ng, P., Johnson, P., Goedicke, M., Kocatas, A., Yan, K. L., How to Develop a General Theory of Software Engineering: Report on the GTSE 2014 Workshop. ACM SIGSOFT Software Engineering Notes. Volume 39, Issue 6, November 2014 pp 23–25.
- Kocatas, A., Erbas, C., Extending essence kernel to enact practices at the level of software modules. GTSE 2014: Proceedings of the 3rd SEMAT Workshop on General Theories of Software Engineering. ICSE '14: 36th International Conference on Software Engineering. June 2014 Pages 32–35.
- Kocatas, A., Pala Er, N., Koksal, O., ARINC 653 Uyumlu Aviyonik Sistemlerin Yapılandırma Dosyalarının Model Güdümlü Gelistirilmesi", Ulusal Yazılım Muhendisligi Sempozyumu, Ankara, Turkey, 2011.
- Kocatas, A., Durmus, U., Tufaner, M., Senlet, T., Model Based, ARINC 653 Compatible Avionics Software Development and Integration, 1. Avionics and System Integration Symposium, Ankara, Turkey, 2010.
- Kocatas, A., Gursoy, A., Atalay R. C., Application of Data Mining Techniques Into Protein-Protein Interaction Prediction, LNCS Springer-Verlag Heidelberg, Vol. 2869/2003, 316-323.