applied sciences

MDPI

Article

Enhancing UML Connectors with Behavioral ALF Specifications for Exogenous Coordination of Software Components

Alper Tolga Kocatas [1,2,*] and Ali Hikmet Dogru [2,3]

[1] Aselsan Inc., 06750 Ankara, Turkey
[2] Department of Computer Engineering, Middle East Technical University, 06800 Ankara, Turkey
[3] Department of Computer Science, University of Texas at San Antonio, San Antonio, TX 78249-0667, USA
* Correspondence: tolga.kocatas@metu.edu.tr or kocatas@aselsan.com.tr

Abstract: 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 study, 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 will 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 increases component reusability.

Keywords: ALF; behavior; connector; fUML; model transformation; port; QVT; QVTo; UML

Citation: Kocatas, A.T.; Dogru, A.H. Enhancing UML Connectors with Behavioral ALF Specifications for Exogenous Coordination of Software Components. *Appl. Sci.* **2023**, *13*, 643. https://doi.org/10.3390/app13010643

Academic Editor: Sanjay Misra, Robertas Damaševičius and Bharti Suri

Received: 23 November 2022
Revised: 24 December 2022
Accepted: 29 December 2022
Published: 3 January 2023

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 [1]. 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) [2]. 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 [3]. 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 article 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) [4] for specifying connector behaviors. ALF can be used to declare the structural and behavioral parts of Foundational UML (fUML) [5] 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) [6] 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 in UML (This article follows UML's general convention, which capitalizes the first letter of classifier names such as *Behavior, OpaqueBehavior, Property, ProtocolStateMachine*, etc.).

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 article 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 article is organized as follows: Section 2 presents the related research. Section 3 describes the problem in detail and provides the motivation for the solution of the

problem. Section 4 presents the approach for the specification of connector behaviors using ALF. Section 5 describes the model transformations. Section 6 presents the set of example connectors developed using the proposed approach. Section 7 lists the cases from real-life avionics software projects where we apply enhanced connector concepts. Section 8 presents an analysis of how the approach can be applied to asynchronous coordination. Section 9 includes an analysis of how enhanced connectors are related to ProtocolStateMachines in UML. Section 10 presents our discussions and future work. Finally, Section 11 wraps up our conclusions.

2. Related Work

Connectors were studied before UML was introduced. One of the earlier studies defines architectural connectors as explicit semantic entities [3]. The study specifies the connectors as a collection of protocols that characterize the role of each participant 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) [8]. 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) [9], 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 [10] 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 [11] brings some of the concepts in Real-time Object-Oriented Modeling (ROOM) [12] 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 [13] 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 [14], 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 [15] is one of the mechanisms for coordination. By defining what is called an interaction machine, Reo frames the motivation for coordination [16]. 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 have been presented in [17] using timed data streams.

The study presented in [18] 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 [19] 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 [20] presents a formal framework to support the rigorous design of software architecture 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 [21] 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 [22] uses the π-ADL [23] to provide a formal semantics for the SysADL [24] models. The study maps SysADL architecture descriptions to π-ADL using model transformations defined in the ATLAS Transformation Language (ATL) [25]. 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.

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.

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 increases the reusability of the connected components and simplifies their design since the connectors take on the coordination responsibility. It also allows the reuse of the connector behavior specifications in similar coordination scenarios.

Our motivation is to achieve exogenous coordination by associating behaviors with the connectors. Aside from 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 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.

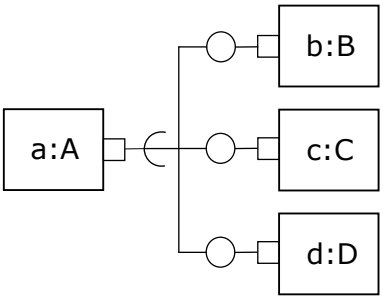


Figure 1. Single requester connected to multiple providers.

3.2. Problem Statement

The previous versions of UML leave the case shown in Figure 1 as a semantic variation point and do not define which providers will receive the request. The current version of UML [2] 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 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. 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. 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 [2].

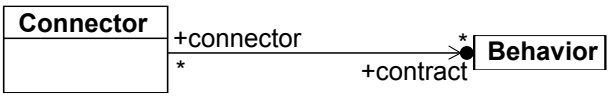


Figure 2. Abstract syntax in UML showing the contract role.

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

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 shown in Figure 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. Additionally, let us assume that the interface used to specify the contract of the ports is *XIfc*. Let us also assume that the interface declares an operation *xOp*, which returns an integer value.

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 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.

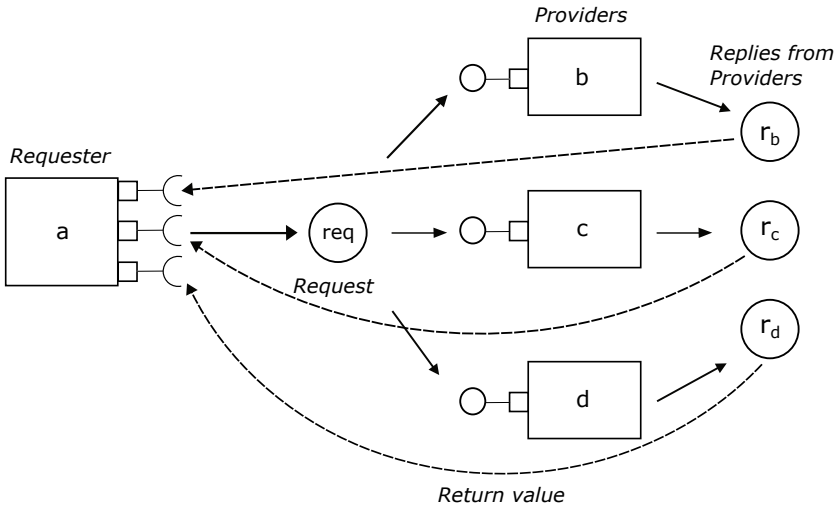


Figure 3. Modifying object *a* in Figure 1 to receive replies from multiple ports.

Without modifying the requester, the connector behavior can perform one of the operations described in Figure 4 to solve the problem caused by multiple replies. In the first strategy shown in Figure 4a, the connector behavior discards replies from all providers and sends a default reply to the requester. In the second strategy shown in Figure 4b, the connector behavior chooses the reply from one of the providers. In the third strategy shown in Figure 4c, the connector behavior uses the replies from all providers to calculate a *merged reply* and sends this merged reply to the requester.

A similar problem can exist for merging requests when there are multiple requesters, as shown in Figure 5. 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.

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.

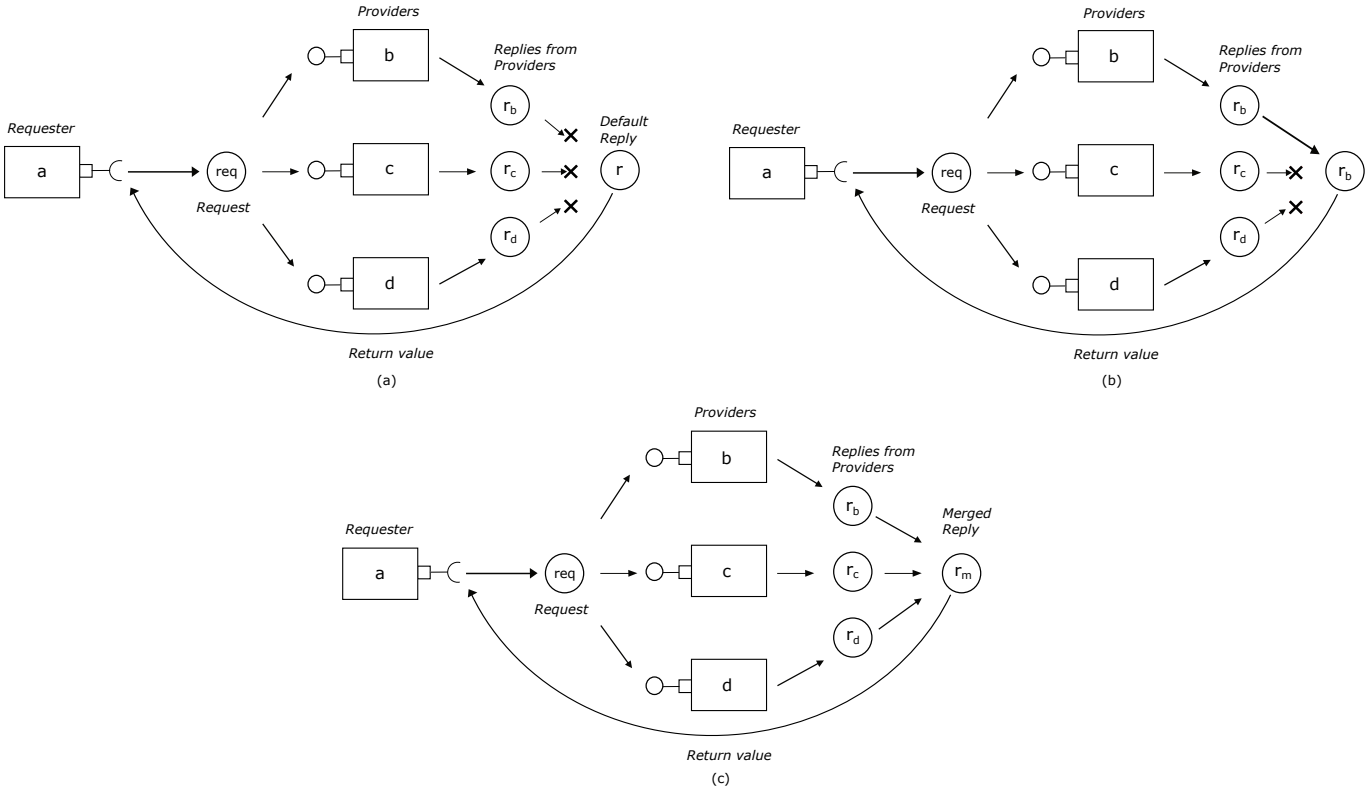


Figure 4. Reply merge strategies for the example case presented in Figure 1: (**a**) Replies from providers are discarded, and a default reply is returned. (**b**) One of the replies is chosen. (**c**) Replies from providers are merged into a single reply.

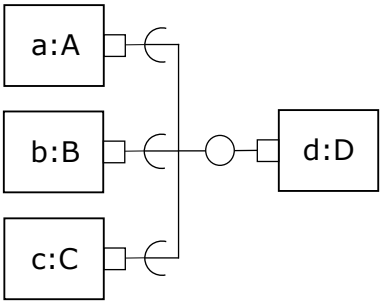


Figure 5. Multiple requesters connected to a single provider.

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.

4.1. Using Buffers for Replies and Requests

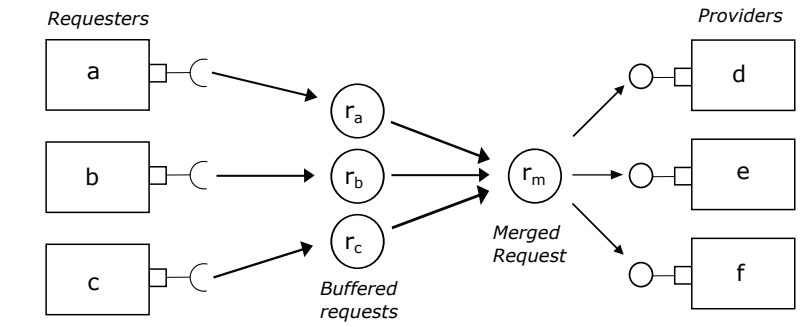
We use request and reply buffers for the specification of merge behaviors. We model the *reply merge* operation required in the "single requester-multiple providers" scenario (see Figure 1) using buffers dedicated to providers. We store the replies from providers in the *reply buffers*. Then, we operate 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 5) 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:

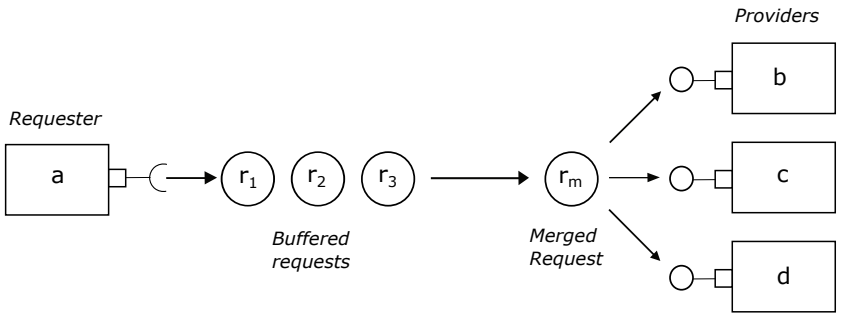
- *Single-copy buffers* are for storing a snapshot copy of requests or replies in which subsequent requests or replies overwrite the previous ones.
- *FIFO buffers* are for storing subsequent requests or replies using first-in, first-out ordering.

For example, in the case presented in Figure 4c, 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 6a 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 6b 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 becomes full, we combine the requests stored in the buffer as a merged request and then send that request to the connected providers.



(a)



(b)

Figure 6. 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.

Examples in Figure 4a,b 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.

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 [4,26]. 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 7, an example included in the ALF specification [4], 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.

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 study, 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 8 shows an example UML model where we define an enhanced UML connector. Listing 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.

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.

```
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)))
  }
}
```

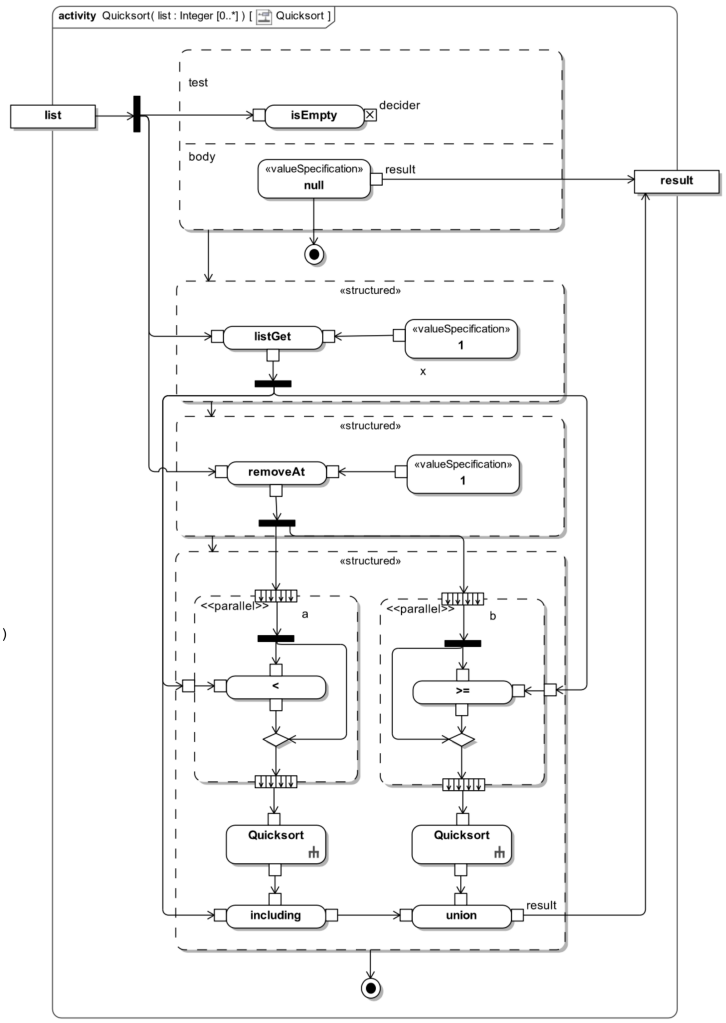


Figure 7. The Quicksort algorithm using ALF and graphical notation.

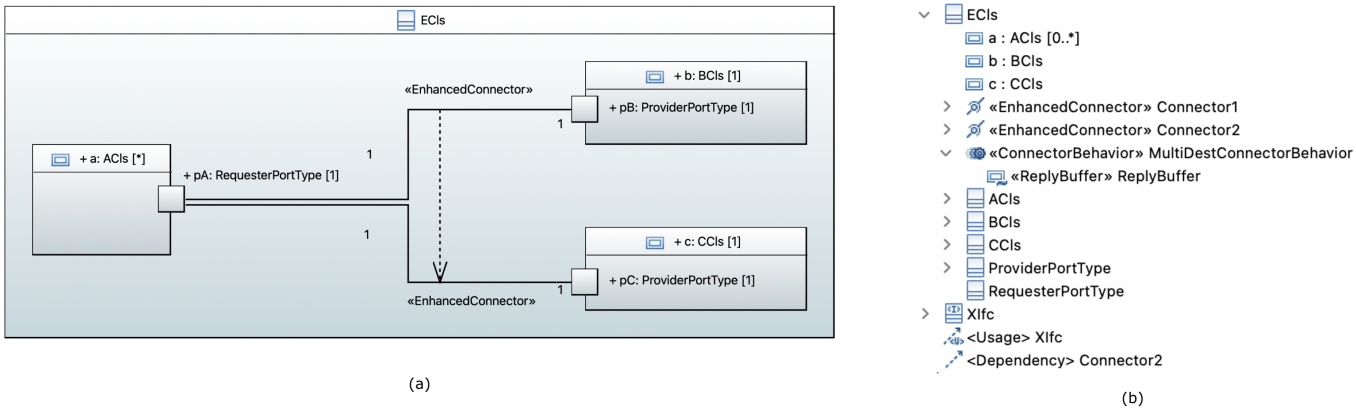


Figure 8. UML diagram and the UML model for the enhanced connector that sends requests to each provider: (**a**) Composite structure diagram showing the requester, providers, and the enhanced connector. (**b**) UML model showing the connector behavior.

Listing 1. Connector behavior specification using ALF for the model in Figure 8.

```
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 <= PRV_CNT) {
  Integer intval = this.ReplyBuffer[i];

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

Table 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 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.

4.3. Enhanced Connector Profile

Using a profile is one of the methods for extending UML. Figure 9 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.

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.

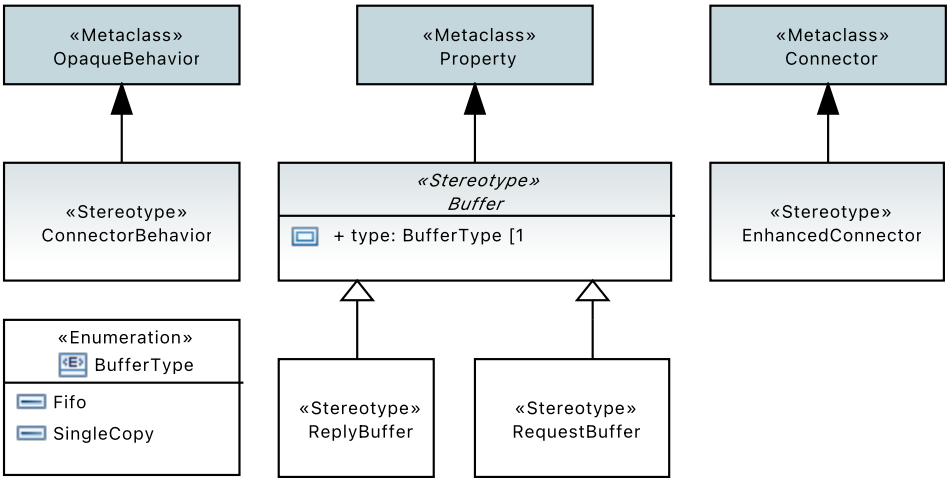


Figure 9. Enhanced connector profile.

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 8a shows one such dependency among the connectors using a dashed line. The contract role of the secondary enhanced connectors is empty.

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 transformations convert the input model into a model in which fUML activities represent the connector behaviors. Figure 10 shows the overall workflow for executing the model transformations. 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.

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 which describes 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 these ALF specifications into fUML activities in the E3 model.

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 [6]. We used Papyrus [27] to develop the UML models, and the QVTo plugin of Eclipse [28] to develop and test model transformations. In the following sections, we describe the model transformations in detail.

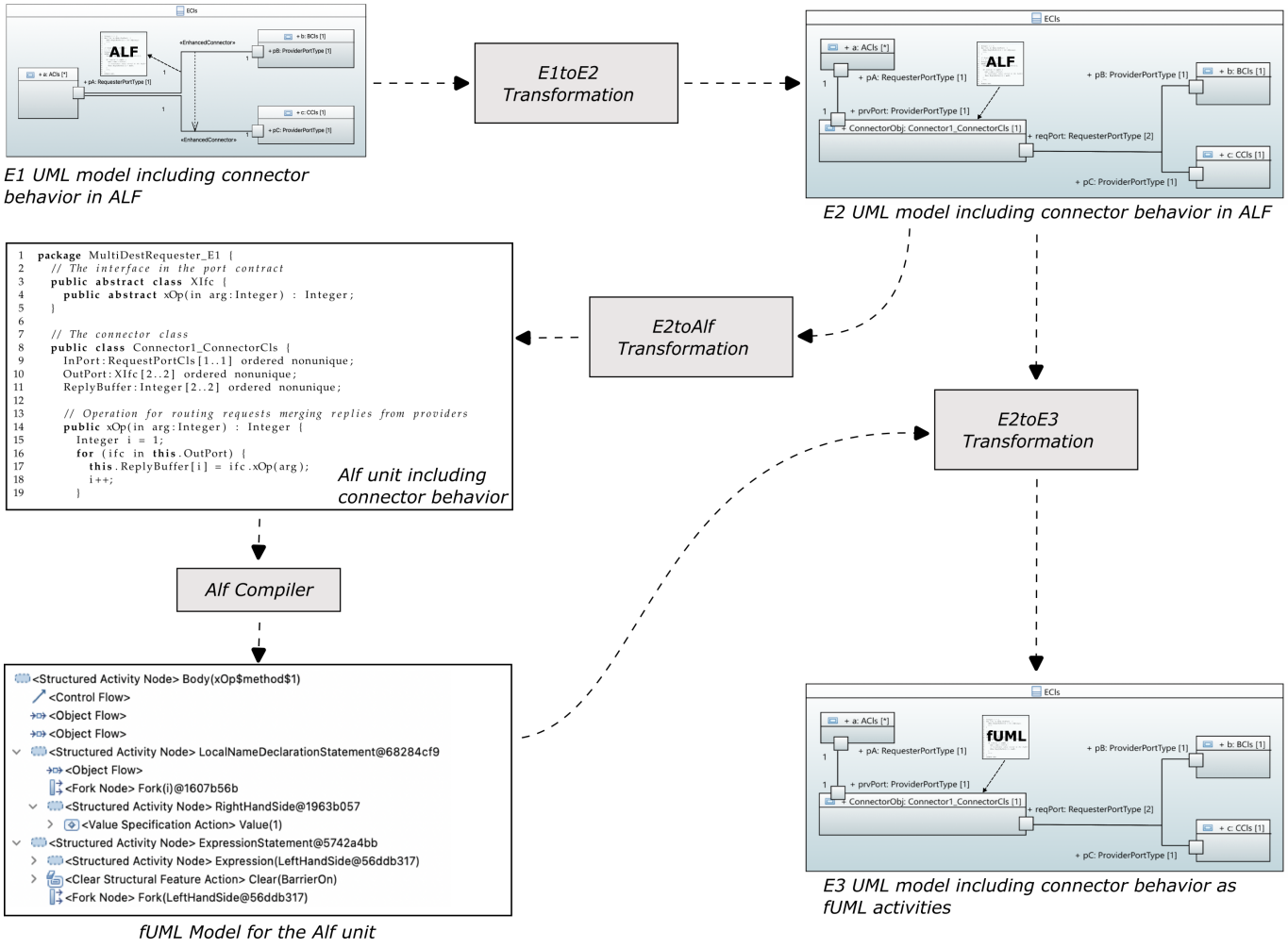


Figure 10. Enhanced connector transformation process.

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 11 shows the E2 model that results from the E1 model shown in Figure 8. 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 Figures 8 and 11). 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.

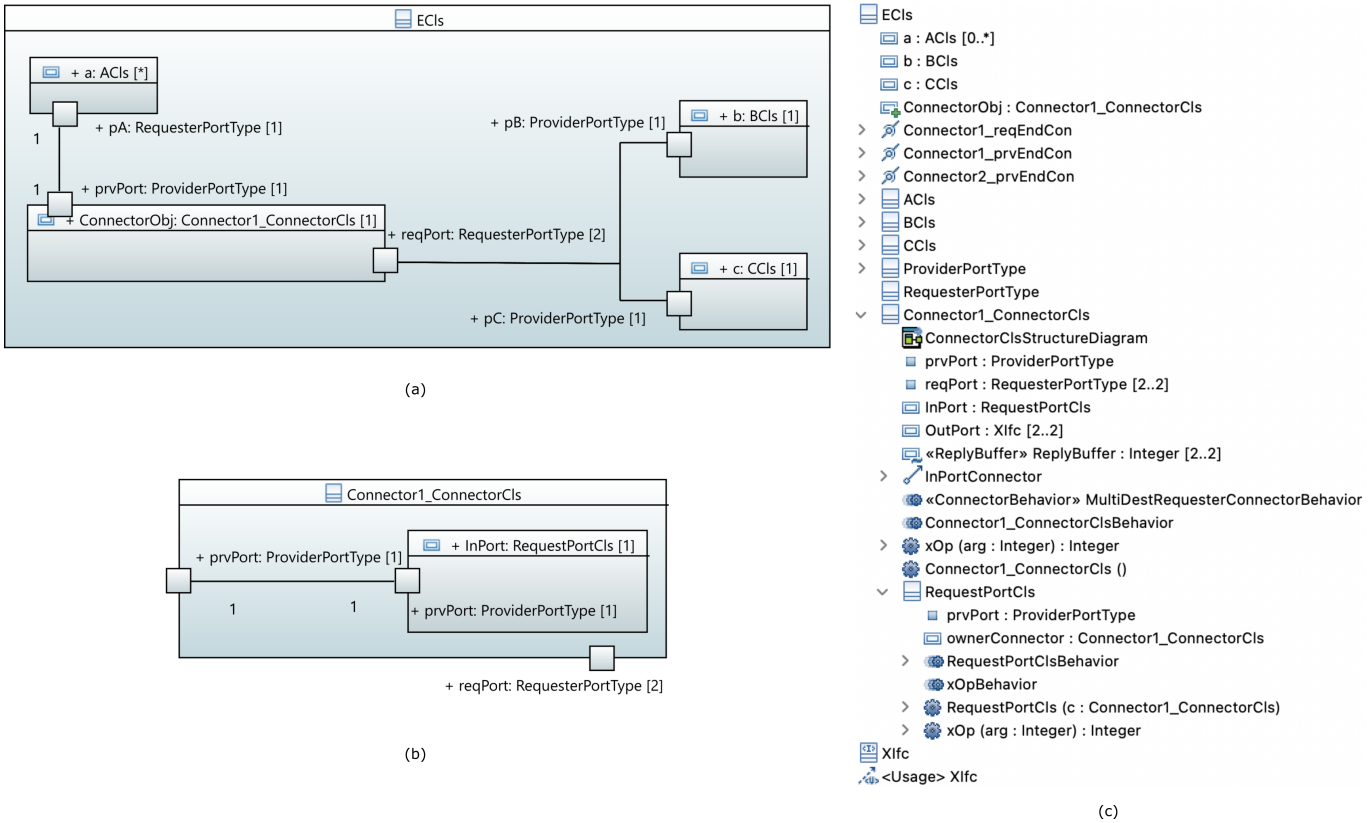


Figure 11. E2 model for the example shown in Figure 8: (**a**) Composite structure diagram that contains the requester, providers, and the object of the connector class. (**b**) Composite structure diagram for the connector class. (**c**) UML model showing the connector behavior and other behaviors generated by the model transformation.

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.

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* [29] 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 shows the ALF unit generated from the E2 model shown in Figure 11 using E2toAlf. We then use the ALF reference implementation to compile this model into an fUML model. 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.

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 10. We have also experimented with ATL [25] to implement the 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 the 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 representations 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.

Listing 2. ALF unit generated from the E2 model shown in Figure 11.

```
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
```

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.

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 8a shows the composite structure diagram for the E1 model. Listing 3 shows the connector behavior specification.

Listing 3. Connector behavior specification for the round-robin requester.

```
1  Integer result = this.OutPort[this.Index].xOp(arg);
2  this.Index = this.Index + 1;
3  this.Index = this.Index % PRV_CNT;
4  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.

A 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.

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 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.

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 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.

The requests are dequeued from the FIFO request buffer until the buffer is empty. Figure 6b 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.

Listing 4. Connector behavior specification for the less frequent sender.

```
1  Integer returnValue = this.DefaultReply;
2  if (this.InPort[1].RequestBuffer.size() == this.MaxRequestCount) {
3    // Merge requests:
4    Integer sum = 0;
5    while (this.InPort[1].RequestBuffer.size() != 0) {
6      Integer request = this.InPort[1].RequestBuffer.removeFirst();
7      if (request != null) {
8        sum = sum + request;
9      }
10   }
11   returnValue = OutPort[1].xOp(sum);
12 }
13 return returnValue;
```

6.4. The Request Barrier

The request barrier connector coordinates multiple requesters and a single provider. Figure 12 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 5 shows the connector behavior specification.

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).

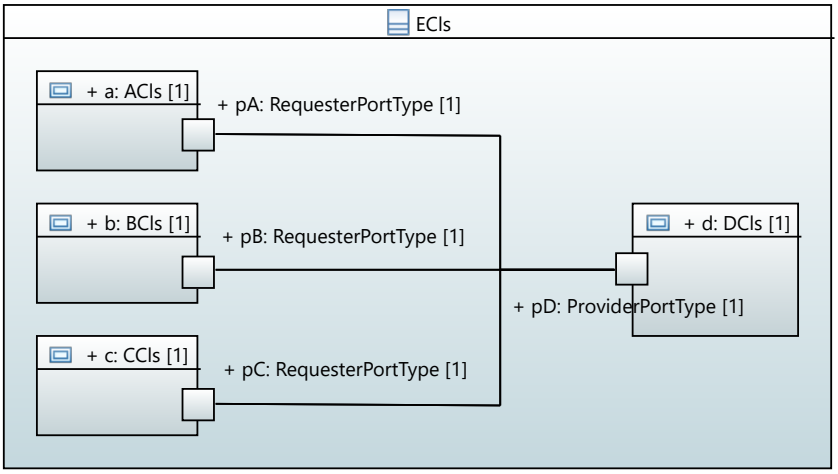


Figure 12. E1 level UML model composite structure diagram for the request barrier connector.

Listing 5. Connector behavior specification for the request barrier.

```
1  Integer i = 1;
2  this.BarrierOn = false;
3  while (i <= REQ_CNT) {
4    Integer request = this.InPort[i].RequestBuffer;
5    if (request == null) {
6      this.BarrierOn = true;
7    }
8    i++;
9  }
10
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 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).

Figure 13a shows the composite structure diagram, and Figure 13c 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 13b shows the internal structure of the connector class, which contains the *InPort* property.

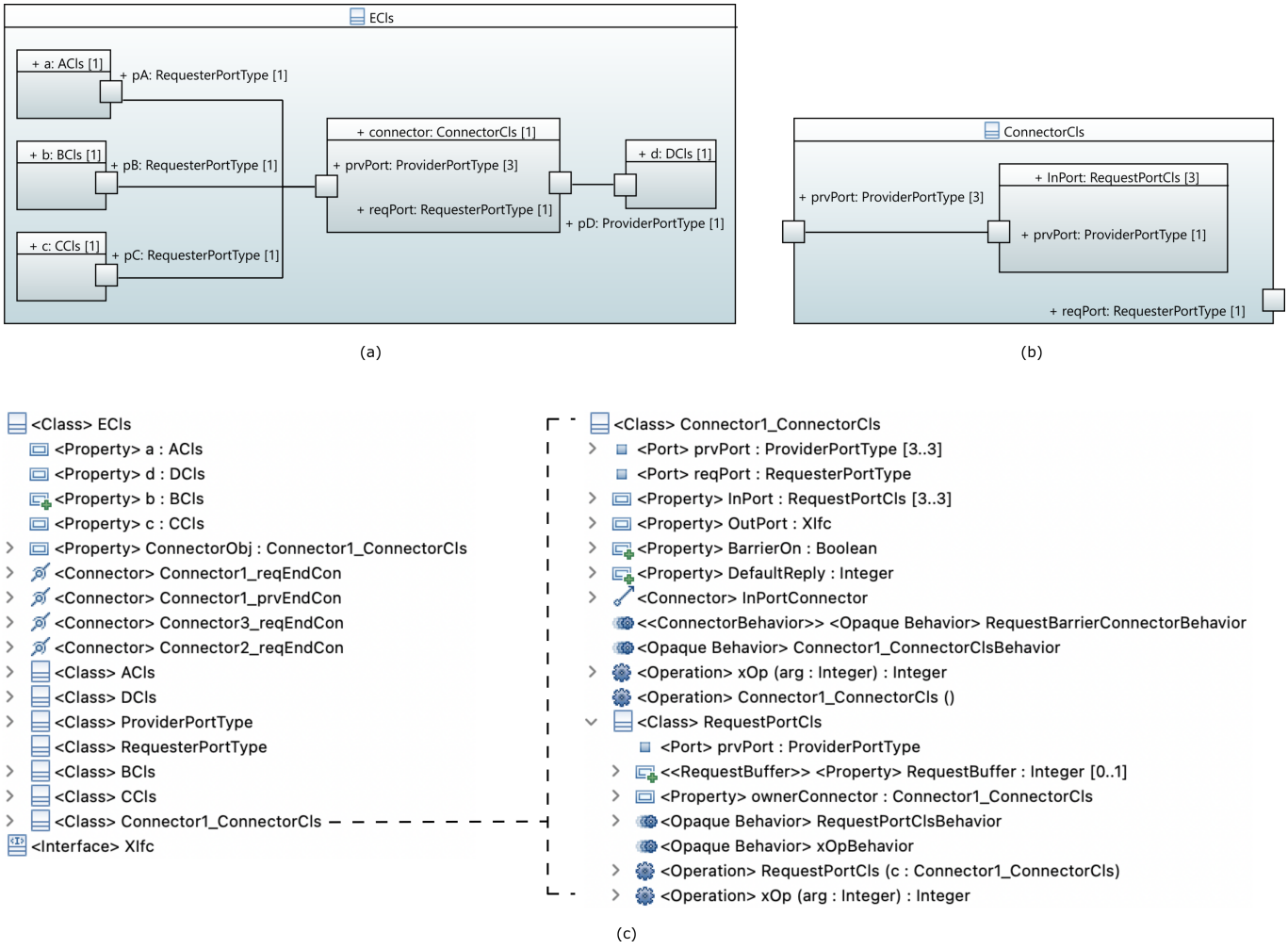


Figure 13. E2 level composite structure diagrams and UML models for the request barrier connector: (**a**) Composite structure diagram for the request barrier connector. (**b**) Composite structure diagram for the connector class. (**c**) E2 level UML model.

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

Figure 14a 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 14b shows the inner model elements for this activity, and Figure 14c shows the internal model elements of the *structured activity node* inside this activity.

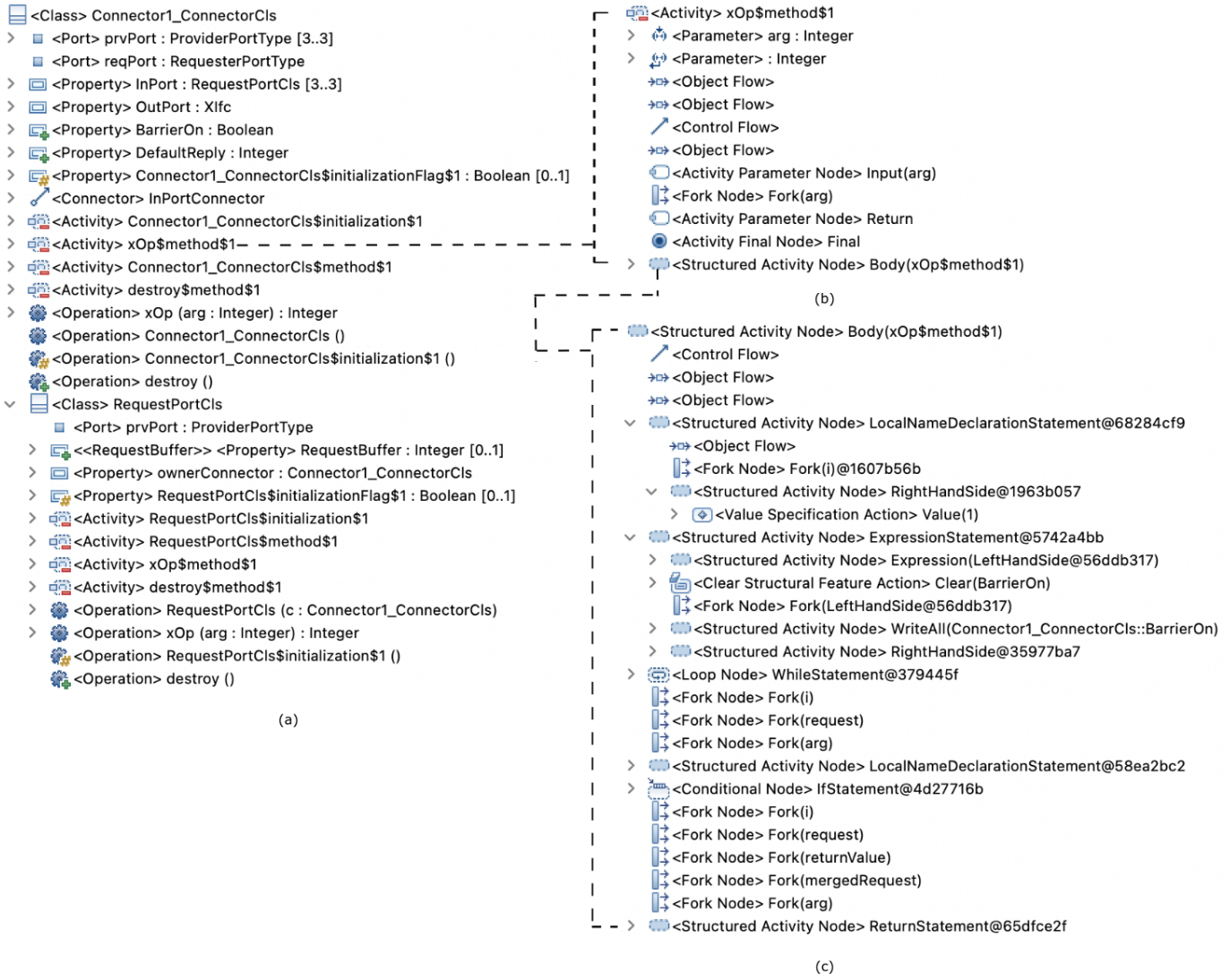


Figure 14. (**a**) E3 UML model for the request barrier connector class. (**b**) E3 UML model for the request barrier connector fUML activity. (**c**) Inner elements of the E3 model request barrier connector fUML activity.

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 15 illustrates the cases searched. We studied the results for both cases to determine if we can use enhanced connectors instead of existing connectors.

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.

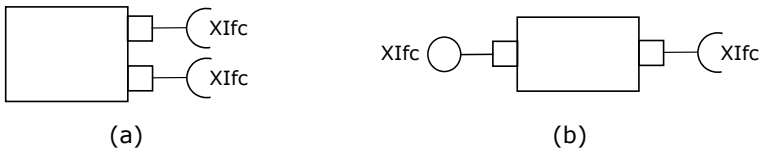


Figure 15. Cases searched in real-life projects: (**a**) Case 1: a class having more than one port that requires the same interface. (**b**) Case 2: a class that provides and requires the same interface at multiple ports.

7.1. Background

The studied projects conform to *DO-178C, Software Considerations in Airborne Systems and Equipment Certification* [30]. 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 [31].

Table 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. Moreover, 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.

Table 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

In addition to DO-178C, the avionics software projects we examined conform to ARINC-653 [32]. 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) [33] 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.

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 overwrite 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 among the 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 16 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 are 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 are sent to four destinations: Cmp_1, Cmp_2, *partition A, and partition B*. The ARINC-653 implementation 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*.

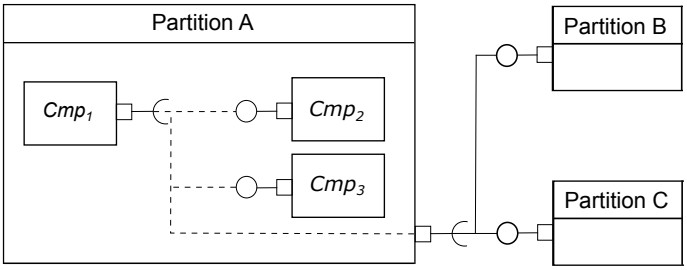


Figure 16. Sampling data produced by a software component sent to multiple destinations. An enhanced connector is used inside Partition A to multicast data.

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 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.

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 17a shows the *Selector* object in the existing design, which forwards the incoming requests to Com_1 or Com_2 device accordingly. Figure 17b 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.

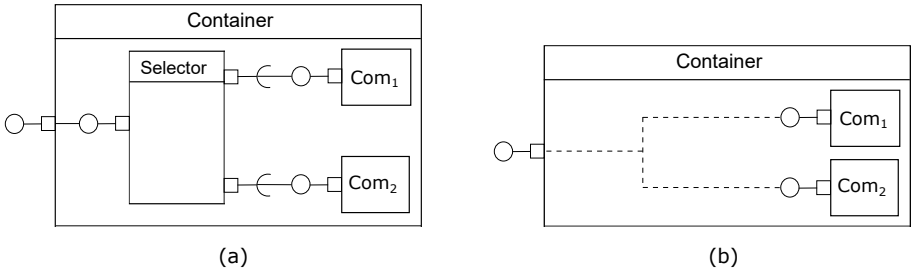


Figure 17. Selector pattern, which forwards requests to different communication devices depending on the project configuration: (**a**) Existing design that uses a dedicated object. (**b**) Updated design that uses an enhanced connector.

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.

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 18a 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_1Src and Dev_2Src). The data from the devices also follow two separate communication channels towards other software components and partitions. Data sent by Dev_1Src and Dev_2Src arrive at device controllers Dev_1Dest and Dev_2Dest inside $Component_2$.

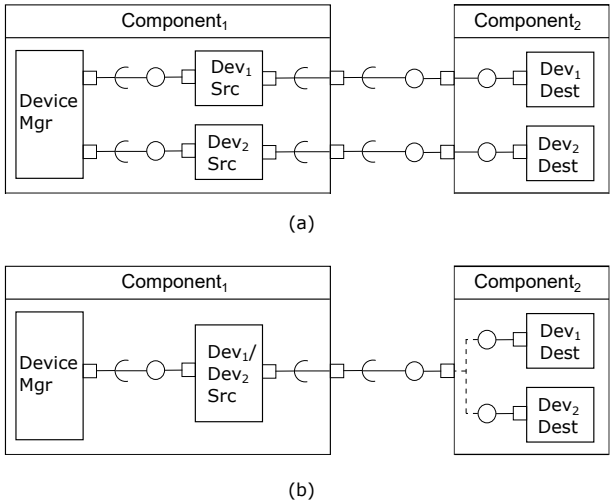


Figure 18. Handling Redundant Devices on the Platform: (**a**) Existing design: Dedicated communication channel for each device. (**b**) Updated design: Single communication channel for multiple devices using an enhanced connector inside $Component_2$.

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 18b shows the updated design. Using the Dev_1/Dev_2Src object, we can obtain 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., $Component_1$, $Component_2$).

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 19a 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.

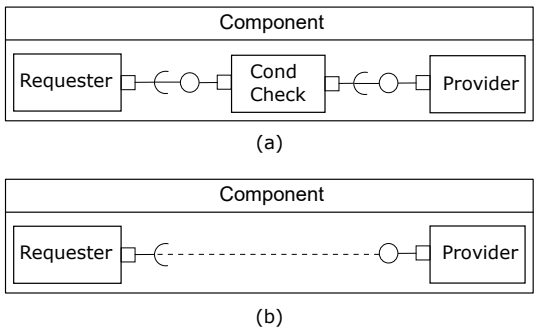


Figure 19. Blocking or forwarding requests depending on the runtime condition: (**a**) Existing design: Using an intermediate object for conditionally blocking the request. (**b**) Updated design: Using an enhanced connector that can conditionally block the request.

Figure 19b 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.

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 20a shows the 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 20b 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.

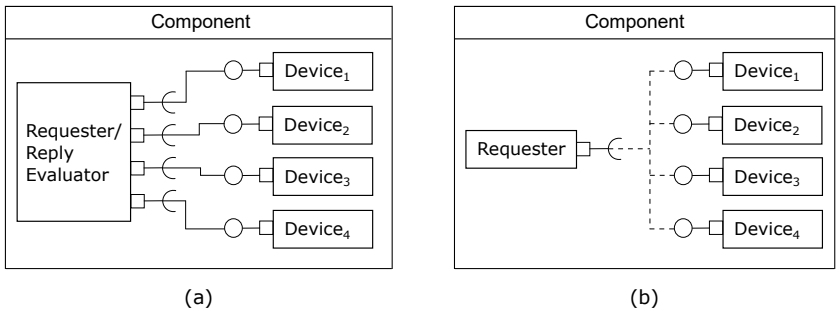


Figure 20. Choosing a single reply among the replies from many providers: (**a**) Existing design: The requester contains the behavior for multicasting the requests and evaluating the replies. (**b**) Updated design: Using an enhanced connector to multicast requests and evaluate the replies.

7.7. Case VI: Discriminating Between Multiple Requesters

Figure 21a 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 set by the checker objects.

Figure 21b 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.

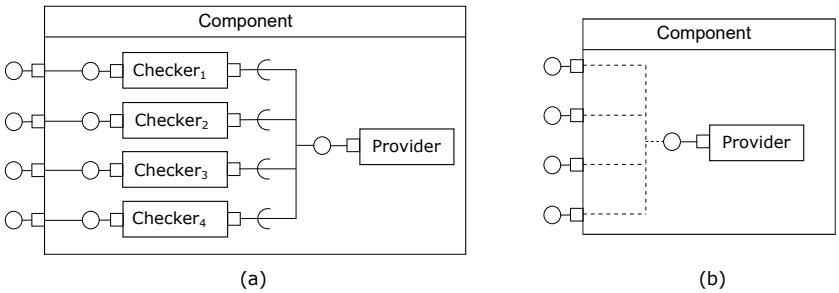


Figure 21. Discriminating between requesters to selectively accept requests from specific sources: (**a**) Existing design: Using dedicated objects to set the source field of incoming requests. (**b**) Updated design: Using an enhanced connector to set the source field of incoming requests.

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 22a shows the existing design. The algorithm for round-robin sending of requests was implemented inside the requesting component, along with other responsibilities.

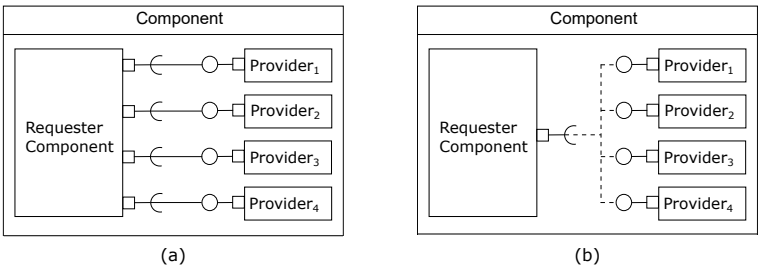


Figure 22. Sending requests to providers in a round-robin order: (**a**) Existing design: Requester component sends requests in round-robin order using multiple ports. (**b**) Updated design: Using an enhanced connector to send requests in round-robin order.

Figure 22b 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 can free the requesting component from this coordination responsibility.

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. Aside from 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 23 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 23a, the requester waits for the reply from the provider before it can send another request. In the asynchronous case shown in Figure 23b, 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 24 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 24a, 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.

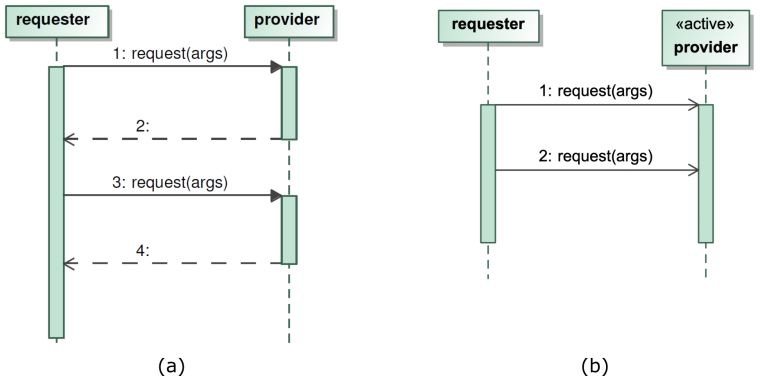


Figure 23. Synchronous and asynchronous handling of requests without using enhanced connectors: (**a**) Synchronous handling of requests. (**b**) Asynchronous handling of requests.

In the asynchronous case shown in Figure 24b, we design the connector as an active object which acts asynchronously. In the figure, filled arrows indicate synchronous messages, and unfilled arrows indicate asynchronous messages. Upon receiving 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.

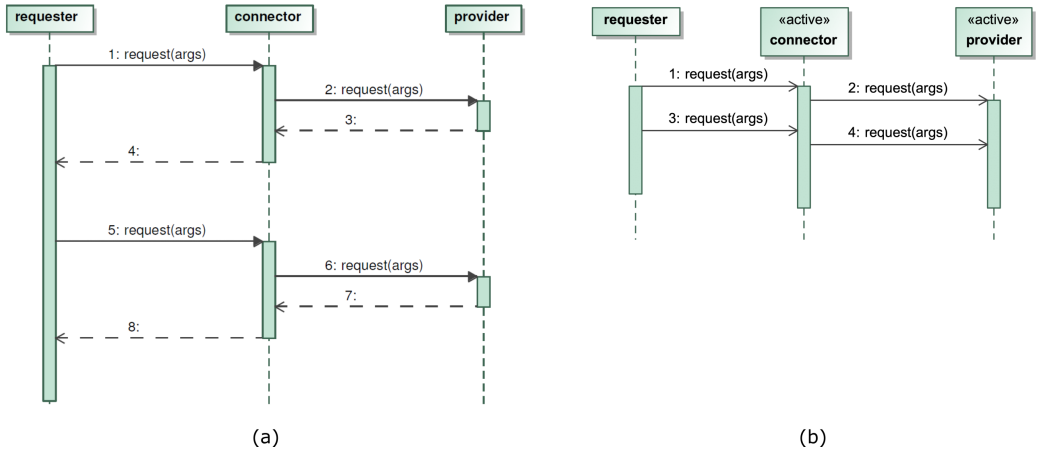


Figure 24. Synchronous and asynchronous handling of requests using enhanced connectors: (**a**) Synchronous handling of requests using enhanced connectors. (**b**) Asynchronous handling of requests using enhanced connectors.

9. Relation with Protocol State Machines

UML defines the concept of *ProtocolStateMachines*, similar to the *protocol* concept in UML-RT [34] and ROOM [12]. 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 25 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.

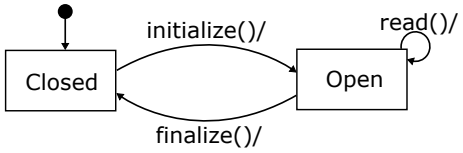


Figure 25. Protocol state machine example.

Let us assume that the ProtocolStateMachine shown in Figure 25 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.

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 [10,15,18] 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 [10] 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 [18] 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 [11] from ROOM [12]. UML further inherited the concept from UML-RT. UML-RT defines the

protocols independent from any specific context; therefore, they are reusable [35]. 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 [2]. 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 [26] and the ALF specification [4] 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 [19] 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 [8]. 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 [6] is suitable and mature enough for developing model transformations. Although QVTo does not support the refining mode of ATL [25] that enables making a few changes on the 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 [36], IBM Rhapsody [37], Papyrus [27], StarUML [38], and IBM Rational Rose [39], do not support them. Some of the tools only have support for n-ary associations. SysML [14] 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 [40–44] 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 [45] 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 [45–47] 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 [15] 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. Since we cannot use connectors as *ConnectorEnds* in UML, we cannot attach a connector to another connector. Therefore, composition support 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.

The cases we demonstrated from the real-life projects showed that the enhanced connector concept is applicable. In [48], 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 study 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 6b, 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) [49] statements to develop a fully functional production-ready system.

11. Conclusions

This article 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 simplifies the design, increases the reusability of the components, and helps 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.

Author Contributions: Conceptualization, A.T.K. and A.H.D.; Data curation, A.T.K.; Formal analysis, A.T.K.; Funding acquisition, A.T.K.; Investigation, A.T.K. and A.H.D.; Methodology, A.T.K. and A.H.D.; Project administration, A.T.K.; Resources, A.T.K.; Software, A.T.K.; Supervision, A.H.D.; Validation, A.T.K. and A.H.D.; Visualization, A.T.K. and A.H.D.; Writing—original draft, A.T.K.; Writing—review and editing, A.T.K. and A.H.D. All authors have read and agreed to the published version of the manuscript.

Funding: The APC was funded by ASELSAN.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: UML models for the example connectors presented in the paper, 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 (accessed on 2 January 2022). Additionally, the Code Ocean capsule at https://codeocean.com/capsule/6929314/tree/v2 (accessed on 2 January 2022) allows a reproducible run and analysis of the results using a web browser.

Acknowledgments: The motivation for research on the enhanced UML connector concepts came to life from working in the avionics software industry, then starting a challenging Ph.D. program in model-driven software engineering. Without the support from a large number of people, its completion would have been even more of a challenge, which is why we want to thank our supporters. First, we want to thank Aselsan Inc. for providing the resources for working on the research problems that emerged from the use of cutting-edge technology. Even with support from industry, we could get lost in a sea of different problems. Therefore we want to give our special thanks to Halit Oğuztüzün and Onur Demirörs for providing helpful directions and feedback that had a significant impact on our research. Finally, we want to thank Ed Seidewitz for his support in the problems we encountered while using the ALF reference implementation.

Conflicts of Interest: Alper Tolga Kocatas is employed by ASELSAN.

Abbreviations

The following abbreviations are used in this manuscript:

ALF	Action Language for Foundational UML
ATL	ATLAS Transformation Language
CSP	Communicating Sequential Processes
FIFO	First-In, First-Out
fUML	Foundational UML
OCL	Object Constraint Language
QVTo	QVT Operational Mappings Language
ROOM	Real-time Object-Oriented Modeling
UML	Unified Modeling Language

References

1. Stahl, T.; Völter, M. *Model-Driven Software Development: Technology, Engineering, Management*, 1st ed.; Wiley: New York, NY, USA, 2006; ISBN 978-0-470-02570-3.
2. Unified Modeling Language, Version 2.5.1, December 2017. Available online: www.omg.org/spec/UML/2.5.1 (accessed on 22 November 2022).
3. Robert, A.; David, G. A formal basis for Architectural Connection. *ACM Trans. Softw. Eng. Methodol.* **1997**, *6*, 213–249. [CrossRef]
4. 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).
5. 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).
6. MOF Query/View/Transformation, Version 1.3, June 2016. Available online: www.omg.org/spec/QVT/1.3 (accessed on 22 November 2022).
7. Kocatas, A.; Dogru, A. Enhancing UML Connectors with Behavioral Specifications. In Proceedings of the 2022 IEEE/ACIS 20th International Conference on Software Engineering Research, Management and Applications (SERA), Las Vegas, NV, USA, 25–27 May 2022; pp. 16–21. [CrossRef]

8. Hoeare, C.A.R. *Communicating Sequential Processes*; Prentice-Hall, Inc.: Upper Saddle River, NJ, USA, 1985; ISBN 978-0-13-153271-7.
9. Allen, R.J. A Formal Approach to Software Architecture. Ph.D. Thesis, Carnegie Mellon University, Cambridge, MA, USA, 1997. Available online: https://www.cs.cmu.edu/~able/paper_abstracts/rallen_thesis.htm (accessed on 22 November 2022).
10. 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: Pittsburgh, PA, USA, 2004. Available online: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7095 (accessed on 22 November 2022).
11. Selic, B. Using UML for Modeling Complex Real-Time Systems. In Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems, Montreal, QC, Canada, 19–20 June 1998; pp. 250–260. [CrossRef]
12. Selic, B.; Gullekson, G.; Ward, P.T. *Real-time Object-Oriented Modeling (ROOM)*; John Wiley & Sons Inc.: New York, NY, USA, 1994; ISBN 978-0-471-59917-3.
13. Object Management Group, "MARTE Tutorial" November 2007, Version 1.1. Available online: www.omg.org/omgmarte/Tutorial.htm (accessed on 22 November 2022).
14. OMG System Modeling Language (SysML), Version 1.6, 2019. Available online: www.omg.org/spec/SysML/1.6 (accessed on 22 November 2022).
15. Arbab, F.; Mavaddat, F. Coordination through channel composition. In Proceedings of the COORDINATION 2020: Coordination Models and Languages, 5th International Conference, York, UK, 15–19 June 2002. [CrossRef]
16. 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).
17. Arbab, F. Reo: A channel-based coordination model for component composition. *Math. Struct. Comp. Sci.* **2004**, *14*, 329–366. [CrossRef]
18. Lau, K.; Ornaghi, M.; Wang, Z. A soft ware Component Model and Its Preliminary Formalization. In *FMCO 2005: Formal Methods for Components and Objects, Lecture Notes in Computer Science (LNCS, Volume 4111)*; de Boer, F.S., Bonsangue, M.M., Graf, S., Roever, W., Eds.; Springer: Berlin/Heidelberg, Germany, 2005; pp. 1–21. [CrossRef]
19. Janisch, S. Behaviour and Refinement of Port-Based Components with Synchronous and Synchronous Communication. Ph.D. Thesis, Universidade Nova de Lisboa, Lisbon, Portugal, 1999. Available online: https://edoc.ub.uni-muenchen.de/12075/1/Janisch_Stephan.pdf (accessed on 22 November 2022).
20. Rouland, Q.; Hamid, B.; Jaskolka, J. Formal specification and verification of reusable communication models for distributed systems architecture. *Future Gener. Comput. Syst.* **2020**, *108*, 178–197. [CrossRef]
21. Jackson, D. *Software Abstractions: Logic, Language, and Analysis*; The MIT Press: Cambridge, MA, USA, 2006.
22. Araújo, C.; Batista, T.; Cavalcante, E.; Oquendo, F. Generating Formal Software Architecture Descriptions from Semi-Formal SysML-Based Models: A Model-Driven Approach. In Proceedings of the International Conference on Computational Science and Its Applications, Cagliari, Italy, 13–16 September 2021; Springer: Cham, Switzerland, 2021. [CrossRef]
23. 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 Softw. Eng. Notes* **2004**, *29*, 1–14. [CrossRef]
24. 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, Switzerland, 2016; ISBN 978-3-319-44339-3.
25. Jouaulta, F.; Allilairea, F.; Bézivina, J.; Kurtevb, I. ATL: A model transformation tool. *Sci. Comput. Program.* **2008**, *72*, 31–39. [CrossRef]
26. Buchmann, T. Prodeling with the Action Language for Foundational UML. In Proceedings of the ENASE 2017 12th International Conference on Evaluation of Novel Approaches to Software Engineering, Porto, Portugal, 28–29 April 2017. [CrossRef]
27. Eclipse Papyrus Modeling Environment. Available online: www.eclipse.org/papyrus (accessed on 22 November 2022).
28. Eclipse IDE, The Eclipse Foundation. Available online: www.eclipse.org/ide (accessed on 22 November 2022).
29. ALF reference implementation, Model Driven Solutions, Inc. 2020. Available online: https://modeldriven.github.io/Alf-Reference-Implementation (accessed on 22 November 2022).
30. *DO-178C*; Software Considerations in Airborne Systems and Equipment Certification. Radio Technical Commission for Aeronautics (RTCA), Inc.: Washington, DC, USA, 2011.
31. Rierson, L. *Developing Safety-Critical Software, A Practical Guide for Aviation Software and DO-178C Compliance*; CRC Press: Boca Raton, FL, USA, 2013.
32. *653P1-3*; Avionics Application Software Standard Interface: ARINC Specification 653P1-3, Required Services. Aeronautical Radio, Inc.: Annapolis, MD, USA, 2015.
33. *DO-297*; Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations. Aeronautical Radio, Inc.: Annapolis, MD, USA, 2005.
34. Posse, E.; Dingel, J. An Executable Formal Semantics for UML- RT. *Softw. Syst. Model.* **2016**, *15*, 179–217. 7/s10270-014-0399-z. [CrossRef]
35. Selic, B. Turning clockwise: Using UML in the real-time domain. *Commun. ACM* **1999**, *42*, 46–54. [CrossRef]
36. MagicDraw Modeling Tool, No Magic Inc. Version 19.0. Available online: www.magicdraw.com (accessed on 22 November 2022).
37. IBM Rhapsody, Developer Edition. Version 8.1.1. Available online: www.ibm.com (accessed on 22 November 2022).
38. StarUML Modeling Tool. MKLabs Co., Ltd. Version: 5.0.2. Available online: https://staruml.io (accessed on 22 November 2022).
39. IBM Rational Rose Enterprise, Version: 7.0.0.4. Available online: www.ibm.com (accessed on 22 November 2022).

40. Proenca, J.; Clarke, D. Data abstraction in coordination constraints. *Commun. Comput. Inf. Sci.* **2013**, *393*, 159–173. [CrossRef]
41. Wermelinger, M.; Lopes, A.; Fiaderio, J.L. Superposing connectors. In Proceedings of the Tenth International Workshop on Software Specification and Design, Grenoble, France, 11–15 September 2000.
42. Kumar, M.B.; Srikant, Y.N. On the use of connector libraries in distributed software architectures. *ACM SIGSOFT Softw. Eng. Notes* **2002**, *27*, 45–52. [CrossRef]
43. Wermelinger, M.; Fiaderio, J.L. Towards an Algebra of Architectural Connectors: A case study on Synchronization for Mobility. In Proceedings of the Ninth International Workshop on Software Specification and Design, Ise-Shima, Japan, 16–18 April 1998. [CrossRef]
44. Wermelinger, M. Specification of Software Architecture Reconfiguration. Ph.D. Thesis, Universidade NOVA de Lisboa, Lisboa, Portugal, 1999. Available online: http://hdl.handle.net/10362/1137 (accessed on 22 November 2022).
45. Genova, G.; Llorens, J.; Martinez, P. The Meaning of multiplicity of n-ary associations in UML. *Softw. Syst. Model.* **2002**, *1*, 86–87. [CrossRef]
46. Dragan, M. *Model-Driven Development with Executable UML*; Wrox: London, UK, 2009; ISBN 978-0-470-48163-9.
47. Brambilla, M.; Fraternali, P. Domain modeling. In *Interaction Flow Modeling Language*; Brambilla, M., Fraternali, P., Eds.; Morgan Kaufmann: Burlington, MA, USA, 2015; pp. 25–50. [CrossRef]
48. Kocatas, A.; Can, M.; Dogru, A. Lightweight realization of UML ports for safety-critical real-time embedded software. In Proceedings of the 4th International Conference on Model Driven Engineering and Software Development (Modelsward), Rome, Italy, 19–21 February 2016. Available online: https://ieeexplore.ieee.org/document/7954368.
49. Object Constraint Language, Version 2.4, February 2014. Available online: www.omg.org/spec/OCL/2.4 (accessed on 22 November 2022).