INTERACTION PATTERNS AS COMPOSITE CONNECTORS IN COMPONENT-BASED SOFTWARE DEVELOPMENT

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2014

By
Petr Štěpán
School of Computer Science

Contents

Word Count: 68549

List of Tables

List of Figures

Abstract

Interaction Patterns as Composite Connectors in Component-based Software Development
Petr Štěpán
A thesis submitted to the University of Manchester for the degree of Doctor of Philosophy, 2014

In current component models, interaction aspects of system behaviour are mostly specified by components, rather than by composition mechanisms, due to the fact that most composition mechanisms cannot express complex interactions. Consequently current component models do not enjoy the benefits that arise from separating the specification of computation from the specification of interaction in software architecture. This thesis investigates the possibility of representing recurring patterns of interaction as composition mechanisms (and other associated component model entities), as distinct from components that define computation; these composition mechanisms would appear as first-class entities in architectures, and can be stored in and reused from repositories. To this end, we have defined a novel, control-driven and data-driven component model that strictly separates computation from interaction. To represent interaction patterns in this model, we have defined composite connectors that can encapsulate control flow and data flow and can be reused via repositories in different contexts. We have also developed a prototype implementation of the component model, and carried out a case study from the reactive control systems domain in order to evaluate the feasibility of our approach. Comparison with related work shows that our composite connectors improve the state of the art in component-based interaction modelling (i) by specifying control flow and data flow explicitly and separately in software architecture; and (ii) by increasing the reuse potential of interaction patterns compared to patterns that are represented by components only.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to thank my supervisor, Kung-Kiu Lau, for his advice, encouragement and support throughout this project and throughout my stay in Manchester.

I am grateful to Michel Chaudron and Graham Riley, who examined my thesis, for interesting discussion during the oral examination and for their suggestions that improved the quality of this thesis.

Thanks are also due to my colleagues in the Component-based Software Development research group for fruitful discussions during our workshops and a friendly atmosphere in our office. I am particularly indebted to Lily and Cuong, with whom we collaborated on the control-driven and data-driven component model.

Finally, I thank my family for their sustaining support and love.

This work was funded by the Doctoral Training Award from Engineering and Physical Sciences Research Council.

Chapter 1

Introduction

The idea of software components has been present since the very beginnings of software engineering [82]. It was inspired by other areas of engineering, such as electronics, in which systems are assembled from pre-existing, standardised parts – components. Component-based engineering brings scalability since systems are defined as compositions of components that may in turn be further composed to form hierarchical system descriptions. Furthermore, components can be reused across many systems to minimise development efforts, thereby reducing project schedules and budgets.

However, component-based software engineering formed as a discipline much later: in the late 1990s [23], following the popularity of some industrial component frameworks [22, 113] and advances in software architecture [105]. Since then, the discipline has established some key terminology: the concept of a universally-defined component [114] has been superseded with the concept of a component model [56, 75], which defines what components are and how they are composed together to form systems. Different component models thus provide different component definitions that are suitable for different domains and purposes.

Although the component model definition puts equal emphasis on components and their composition, the research in the latter has been given less attention. Every existing component model allows developers to create their own components and deploy them into system architectures – components are first-class entities. On the other hand, composition mechanisms in many component models are only defined implicitly and lack any explicit architectural representation. Typically, they are just thin wrappers over underlying communication mechanisms, such as (remote) procedure calls or some messaging bus, provided by a particular programming language or by middleware libraries. Developers thus cannot define their own composition mechanisms in these models.

As a result, system behaviour in the models with such primitive composition mechanisms is defined entirely by means of components. A composition mechanism typically only links behaviour required by a certain component with the matching behaviour provided by another component (e.g., a caller is linked to a callee in the case the composition mechanism represents procedure call).

The behaviour of a component-based software system can be decomposed into two aspects: computation and interaction (or coordination). Computation corresponds to expression or function (in the mathematical sense) evaluation; interaction defines the order in which computations are carried out (control flow) and the passing of outputs of some computations as inputs to other computations (data flow). Because of their primitive composition mechanisms, current component models mostly mix computation and interaction in component definitions.

However, this violates the general principle of separation of concerns and consequently results in unnecessary coupling, decreased reuse potential of components, and decreased comprehensibility of software architectures comprised of such components. Carriero and Gelernter made an analogous observation in the context of generic programming languages [47]. They proposed to split programming languages into languages that would specify coordination [4, 93] and languages that would specify computation in order to model these concerns separately.

Motivated by the promises of the separate specification of computation and interaction, our research focuses on component models supporting complex component composition mechanisms that would be able to define component interaction separately from component computation.

Another motivator for the research presented in this thesis is the concept of *design patterns*. A design pattern is a written record that captures design knowledge in a particular form which facilitates the communication and reuse of that knowledge. The concept originated in the architecture of buildings [3] and was introduced to software engineering by the object-oriented community [15, 44]; it then spread to other areas of software design [28] due to its universality. A design pattern gives a 'good' solution to a recurring design problem; additionally, it describes the context of the problem as well as various trade-offs and consequences of using the pattern. The solution defines participating entities (their role and expected behaviour) and – crucially – their interaction that leads to solving the given problem.

The key insight that links design patterns with component-based software development is that the interaction among participants defined by solutions of certain design patterns may be viewed as a high-level composition mechanism. In the component-based setting, pattern participants would correspond to components, and their mutual interaction would correspond to a composition mechanism (represented in software architecture by connectors). Lau et al. [70] noticed this correspondence between composite connectors in the X-MAN component model and some behavioural design patterns [44]. The corollary of such correspondence is that design patterns provide a rich pool of potential high-level composition mechanisms.

Our research hypothesis directs our research on component composition mechanisms at design patterns as their source:

> In component-based software development, it is possible to define the solution part of certain design patterns (*interaction patterns*) as composite connectors that (i) are first-class entities in software architecture, (ii) can be composed to create more complex patterns, and (iii) can be reused via repositories in different contexts.

The hypothesis acknowledges that not every design pattern may be translated to a composition mechanism by coining a name for the class of patterns for which the translation is possible: interaction patterns. It also emphasises that the composition mechanism defining interaction patterns should be represented by explicit architectural entities of the same significance as components – including their ability to be composed and reused – but with different semantics due to their sole focus on interaction.

Although pattern-inspired composition mechanisms form the main focus of our research, they cannot be explored in isolation from components. The two concepts are intrinsically linked: since a composition mechanism facilitates communication between components, the definition of components and their interface affects the definition of component composition mechanisms, and vice versa. Therefore, the goals of our research need to be formulated in terms of component models, abstractions that take into account both components and their composition mechanism.

In particular, our research aims

- to identify suitable component model characteristics (i.e., characteristics of components and composition mechanisms) for expressing interaction patterns separately from computation,
- to survey existing component models in order to establish their conformance with such characteristics, and
- to either adapt an existing component model or to develop a new component model with such characteristics so that interaction patterns can be represented as composite connectors – explicit in software architecture and reusable across many systems – in that component model.

Such a component model would have the following benefits:

Firstly, it would promote the separation of computation and interaction, in accordance with the proposals of Carriero and Gelernter [47]. Consequently, it would lead to more comprehensible software architectures and to greater reuse of both components and connectors. Furthermore, the reuse of composite connectors representing interaction patterns would be strengthened by the reusable nature of interaction patterns themselves (if a mechanism was in place that would allow connectors to vary among different contexts of their usage, similarly to patterns).

Secondly, interaction patterns represented as composite connectors would improve the current practice of pattern usage. Currently, designers have to read design patterns first and apply the knowledge embodied in patterns manually during the creation of designs complying with those patterns. With patterns represented as composite connectors, developers would instantiate patterns from repositories into the context of their particular system. Not only would patterns facilitate knowledge reuse but also the reuse of design and implementation artefacts.

1.1 Thesis Outline

This section gives a brief overview of the chapters constituting this thesis.

Chapter 2 explains the research problem addressed in this thesis and gives an overview of the results of our research; it is a good starting point for the reader. Chapter 3 motivates and defines the notion of interaction patterns. Chapter 4 provides the necessary background on component models and their properties relevant in this thesis. The reader familiar with these concepts can skip this chapter.

Chapter 5 enumerates the component model characteristics suitable for an interaction pattern representation in component-based software development, on which our work is based. It also surveys the existing component models with respect to these characteristics and shows the poor support for separate modelling of interaction patterns in current component models, and thus motivates the development of a new component model suitable for separate interaction modelling.

The next three chapters form the technical core of the thesis. Chapter 6 defines the basic elements of the new component model that separates computation and interaction modelling, and shows the characteristics of system architectures constructed in the model. The structural and behavioural description of the model is given as well as the formalisation of its execution using Coloured Petri Nets. Chapter 7 defines composite connectors: their interface, composition structure and run-time behaviour. Chapter 8 focuses on reuse of composite connectors; it defines connector templates for repository-based reuse and presents a number of mechanisms that increase connectors' reuse potential.

Chapter 9 gives the reader an overview of the prototype tool that we implemented to support our approach. The tool is used in the case study in Chapter 10, in which we demonstrate the feasibility of our approach on the mode-switching interaction pattern from the domain of reactive control systems. Chapter 11 discusses related work. We evaluate our approach by comparison with closely related component models, coordination models and approaches to improving usage of design patterns. The chapter also contains the overall evaluation of our approach.

Chapter 12 concludes the thesis by re-iterating the presented research contributions and their limitations, and it discusses promising strands of the future work.

Chapter 2

Overview of Our Research

In this chapter, we give an overview of the research presented in this thesis: Section 2.1 explains our research problem, Section 2.2 discusses the limitations of existing component-based methods, exemplified by UML 2.0, in addressing the problem and Section 2.3 highlights the results of our research. For illustration, we use an example of a photo storage system throughout the chapter. The chapter is a good starting point for the reader as it introduces the main technical concepts of our research and helps the reader navigate through the thesis.

2.1 Research Problem

In this section, we explain our research problem. Motivated by the benefits of separate computation and interaction modelling in system design, we aim to represent recurring patterns of interaction using suitable abstractions, different from those representing computation. To represent them explicitly in software architecture and reuse them both in design and implementation phases of system life cycle, we want to define these abstractions in component-based software engineering as component model entities.

The separation of computation and interaction in software design and implementation is an instance of the general principle of separation of concerns and has been acknowledged to bring a number of benefits: increased modularity, increased comprehensibility of software architectures, increased reuse potential and potentially decreased complexity of system verification [47, 69, 101].

Many coordination languages [47, 93] and architectural patterns [77, 101, 55] exist that promote separating computation and interaction. For instance, Selic [101] calls this the principle of 'separation of control from function' and defines the Recursive Control Pattern. The idea of the pattern is to decompose systems into components responsible for control (interaction) and components responsible for computation. Figure 2.1 depicts such a decomposition. Control components coordinate other components through control interfaces, whereas functional components represent computation. As its name suggests, the pattern is recursive: a functional component may be recursively composed of a number of control subcomponents and functional subcomponents.

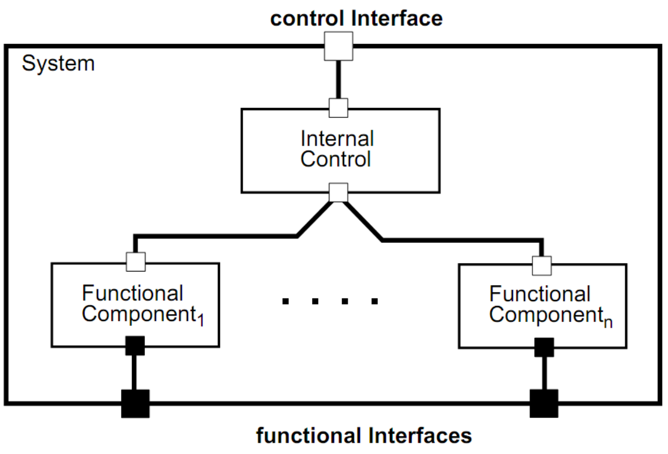


Figure 2.1: Recursive Control Pattern [101]

In order for the benefits of the separate specification of computation and interaction to materialise, system designers need to identify units of computation and units of interaction suitable for being used and reused on their own. System designers are used to identifying units of computation because functional abstractions are abundant in current design and implementation techniques; however, identifying units of interaction is more challenging.

In this thesis, we call these recurring units of interaction *interaction patterns* (Chapter 3). An interaction pattern defines an interaction, i.e., an exchange of data and control signals between a set of computations, but it is not dependent on particular computations. It corresponds to a coordinator that enforces a particular protocol among the participating computations. Like design patterns, interaction patterns should recur in system designs. However, design patterns are more general: they structure design knowledge by providing solutions to recurring design problems, and their purpose is not primarily to separate computation and interaction. Still, because design pattern solutions are typically specified as collaborations of participating entities, the collaboration part of some design patterns can be viewed as an interaction pattern.

For example, the Observer design pattern [44], a well-known design pattern in the object-oriented programming community, solves the problem of notifying a group of objects if the state of an object in which they have registered their interest changes. The solution part of the pattern specifies two kinds of participating entities (roles): Subject, representing the object being observed, and Listener, corresponding to an object listening for changes in the Subject. The solution specifies the interaction between the Subject and a set of Listeners that solves the given problem. This design pattern's solution can be used to extract the Producer-Consumers interaction pattern. We distinguish two kinds of participating computations: Producer (formerly Subject) and Consumers (formerly Listeners). The interaction pattern enforces that Consumers receive and process the datum that the Producer computes when it is invoked.

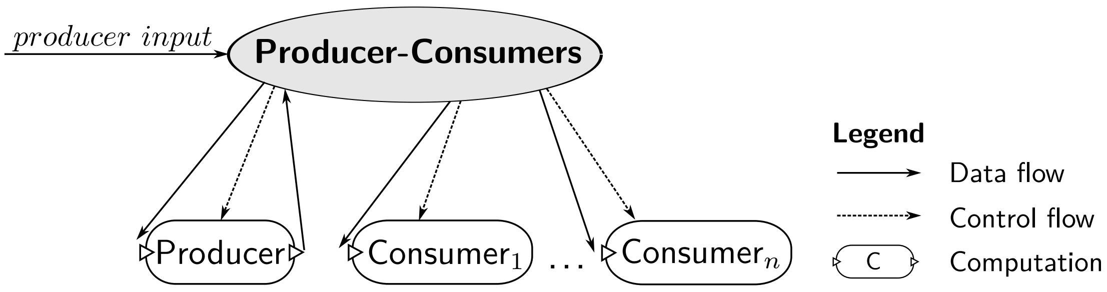


Figure 2.2: Producer-Consumers Interaction Pattern

Figure 2.2 shows the pattern schematically. The interaction pattern defines the interaction (not defined fully in this figure) among participating computations, in terms of control flow and data flow. The pattern does not fix particular computations, it only requires different roles to exhibit some minimal behaviour, such as the ability to produce or consume a datum, to be able to participate in the interaction. In addition, role multiplicities may not be fixed (one producer and at least one consumer). As a result, interaction patterns can be instantiated in software design in many possible ways, like other design patterns.

Interaction patterns identify recurring units of interactions, which helps promote modelling computation and interaction separately in software designs. Like design patterns, they enable reuse of design knowledge and facilitate the communication of software designs by increasing the level of abstraction. However, informal specification of interaction patterns would mean they would inherit the shortcomings of current usage of design patterns. Their instances would be defined each time from scratch, diluted in low-level abstractions provided by existing design and implementation methods. This would preclude them from being first-class entities in design and implementation and would thus thwart their reuse in these phases of software construction.

Our research aims to find a suitable abstraction for interaction patterns that would enable us to define them as explicit entities in software architecture and would make them reusable in both design and implementation. To achieve this goal, we define interaction patterns as first-class entities in component-based software engineering. Component-based system construction is inherently compositional: a system is the result of composition of reusable building blocks. By having separate building blocks for computation and interaction, we can separate these concerns in system architecture. The building blocks of component-based development are defined in so-called component models, which formalise the syntax and semantics (including their run-time behaviour) of abstractions that are used to compose system architecture. Furthermore, component-based software engineering is focused on reuse: there exist separate life cycles for developing reusable component model entities and developing systems out of these entities by reusing them via repositories. By defining interaction patterns as component model entities, not only can we achieve the reuse of their underlying design knowledge, but we can also achieve their full reuse potential as both reusable architectural abstractions and reusable implementation artefacts.

2.2 Shortcomings of Current Approaches

This section explains the shortcomings of most current component models, related to their ability to model computation and interaction separately in software architecture (it draws from the component model survey in Chapter 5 and comparison with related work in Chapter 11). Due to their composition mechanisms that cannot express complex interactions, these models rely on a single abstraction – components – to specify both computation and interaction. Consequently, computation and interaction cannot be distinguished in software architecture and cannot be defined and reused separately.

We illustrate this problem on an example of a photo storage system. We show an architecture of this system in the UML 2.0 component model and explain

(i) how such a representation fails to distinguish computation and interaction in software architecture, and (ii) how this negatively affects the ability to define and reuse interaction patterns.

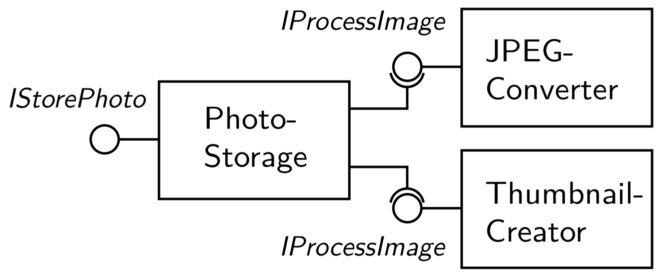
In our comparisons, we limit ourselves to UML 2.0 component diagrams, the subset of UML 2.0 that captures the component-and-connector view of system architecture [33, 68].

Photo Storage System The photo storage system allows users to store photos captured in the raw image format by their cameras. The system uploads the photos to some on-line photo repository; for simplicity, we ignore other functionality of the repository, such as browsing and searching through stored photos. When executed, the system performs the following steps:

1. It converts a given raw image to an RGB image (a 2-D matrix of pixels, each of which consists of three values representing red, green and blue components).
2. It encodes the RGB image as a JPEG image and stores it into the repository.
3. It generates a small preview of the RGB image, which is also stored in the repository.

Mixing Computation and Interaction In Components

Figure 2.3 shows the architecture of the photo storage system in UML 2.0. Before we show how computation and interaction aspects of system behaviour are mixed in UML 2.0 components, we need to give a brief overview of UML 2.0 components and connectors (Chapter 4 contains more background material on component models).



```
interface IStorePhoto {
    storePhoto(rawData: byte [])
}

interface IProcessImage {
    process(image: RGBImage)
}
```

Figure 2.3: Photo Storage System Architecture in UML 2.0

UML 2.0 components are units of functionality with specified interfaces. Component interfaces consist of provided and required ports. A component's provided ports expose the component's functionality to other components; conversely, a component's required ports specify the functionality of other components that is needed by this component to carry out its provided services. The 'functionality' is specified through abstract data types, which comprise sets of operations. UML 2.0 components are first-class component model entities: new components can be defined, and components have an explicit architectural representation (as boxes).

The architecture of the photo storage system in Figure 2.3 comprises three components: PhotoStorage, JPEGConverter and ThumbnailCreator. The PhotoStorage component has a single provided port (depicted as a socket), associated with the IStorePhoto type. It provides the overall functionality of the system: the storePhoto operation uploads a photo in the raw image format to the repository. The functionality is defined by the code (not shown in the figure) that first converts a given raw image (the rawData parameter) to its RGB representation (Step 1 above) and then invokes Steps 2 and 3 by calling the operation process on its two required ports (depicted as receptacles), associated with the IProcessImage type. The components JPEGConverter and ThumbnailConverter define the functionality to perform Step 2 and 3, respectively.

UML 2.0 components are composed by connecting provided and required ports of different components, associated with the same abstract data type. The underlying composition mechanism is method delegation. A component calls operations (methods) defined by the abstract data type associated with its required port; these calls are delegated to the same operations exposed by the connected provided port of another component. The composition mechanism is represented in architecture by connectors. Unlike components, connector types are fixed and new connectors cannot be defined. In architecture, they are represented as links between component ports.

Figure 2.3 depicts two port connections between required ports of PhotoStorage and provided ports of JPEGConverter and ThumbnailCreator. As a result, the process operation calls in the implementation of PhotoStorage will be delegated to the operations defined by the connected components.

Let us now examine computation and interaction in our example system. Each of the three steps defining the behaviour of the photo storage system above represents a computation: converting a raw image into an RGB image, converting an RGB image into a JPEG image and generating a thumbnail are all data transformations. To classify behaviour as inter-component interaction, we need to consider the actual architecture of the system. Figure 2.3 illustrates that each step has been defined as a component. Any exchange of control flow and data flow between components is an interaction. In our example, interaction corresponds to the sequence of two invocations of the process operation of JPEGConverter and ThumbnailCreator and to passing of an RGB image to these components. On the other hand, storing a file in the repository (mentioned in Steps 2 and 3) cannot be considered as inter-component interaction, because the repository is not modelled as a component in our architecture; it is thus viewed as internal computation within JPEGConverter and ThumbnailCreator.

Having identified computation and interaction in our example system, we can observe how they are represented in the architecture in Figure 2.3. As already mentioned, each of the three components defines a computation corresponding to one of the three steps describing the behaviour of the photo storage system above. The interaction that coordinates the invocations of JPEGConverter and ThumbnailCreator is also defined within a component, in PhotoStorage in particular. The code implementing PhotoStorage contains calls to the process operation of its required ports; the control flow and data flow defined by the code determine the order of the two invocations and the data that are being passed. Therefore, the PhotoStorage component represents both computation and interaction aspects of the photo storage system's behaviour; the two aspects are mixed and indistinguishable in the architecture.

Notice the role that connectors are playing here: they only link interface elements of composed components. Although they signify that there is a control flow and data flow exchange between linked components, they are incapable of defining such an exchange, which must be therefore defined within components.

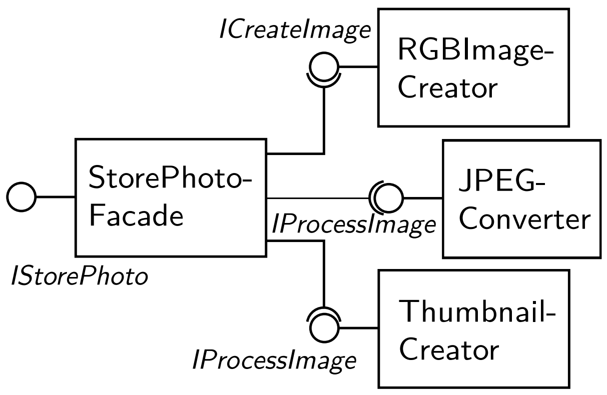
Shortcomings of Component Representation of Interaction

We have shown that standard UML 2.0 component diagrams do not distinguish between interaction and computation by means of different architectural entities since their components model both these aspects of system behaviour.[1] However, this does not imply that every component has to mix computation and

[1]This also holds for similar existing component models.

interaction; it is possible to design systems so that some components represent computation while other components represent interaction. Here, we show how such a component representation of interaction negatively impacts the ability to define and reuse recurring interaction patterns. Again, we use the photo storage system as an illustrative example.

Figure 2.4 shows the architecture of the photo storage system re-designed so that some components represent computation only and other components represent interaction only. We can see that the PhotoStorage component from the original architecture in Figure 2.3 has been split into two components: RGBImageCreator and StorePhotoFacade. The former represents the computation aspect only as it converts a given raw image to a corresponding RGB image. The latter represents the interaction aspect of the original component only.



```
interface IStorePhoto {
    storePhoto(rawData: byte [])
}

interface IProcessImage {
    process(image: RGBImage)
}

interface ICreateImage {
    create(raw: byte []): RGBImage
}
```

Figure 2.4: Redesigned Architecture of Photo Storage System in UML 2.0

The StorePhotoFacade component coordinates the execution of other components by invoking them in sequence and by passing data between them. First, it passes the raw image to RGBImageCreator, invokes the create operation and gets the corresponding RGB image back. Then, it sends the RGB image to JPEGConverter by invoking its process operation. Finally, it invokes the operation of the same name of ThumbnailCreator, passing the RGB image as its input at the same time.

The architecture in Figure 2.4 represents an instance of the interaction pattern Producer-Consumers, mentioned in Section 2.1. In our example, ThumbnailCreator and JPEGConverter are Consumers; they are sent an RGB image whenever RGBImageCreator (the Producer) creates a new one. The StorePhotoFacade component then represents the interaction pattern itself since it defines the interaction among the participating components.

We argue that components, such as StorePhotoFacade in Figure 2.4, are not suitable abstractions for representing and reusing interaction patterns. Firstly, this interaction-defining component is semantically indistinguishable from other components in software architecture. Secondly, since the interaction is defined by the code implementing StorePhotoFacade, it is not explicit in software architecture. Thirdly, StorePhotoFacade does not model control flow and data flow explicitly; instead, these flows are hidden and inseparable. Fourthly, its reuse potential is decreased by dependence on the abstract data types associated with StorePhotoFacade's required ports: unlike the interaction pattern it represents, the component depends on the operation names and exact types of exchanged data. Finally, its reuse potential is further decreased by fixing the number of participants of each role. The underlying interaction pattern may vary the number of Consumers, unlike StorePhotoFacade, which is constrained to two participants of this role.

2.3 Our Approach

Our research addresses the problem of finding an abstraction defined within a component model that is suitable for representing and reusing interaction patterns in component-based software development (see Section 2.1). We need to overcome the failure of most existing component models to distinguish computation and interaction in software architecture, which results in poor representations of interaction patterns by components (see Section 2.2). To this end, we have developed a new component model that defines computation and interaction separately; the model also provides reusable abstractions suitable for representing interaction patterns.

In this section, we illustrate these properties of the new component model with the example of the photo storage system we have used in the previous section. We first show how computation and interaction are distinguished in the system's architecture. Further, we demonstrate the ability of composite connectors in our model to form high-level composition mechanisms that embody interaction patterns.

Separate Computation and Interaction Modelling

We have designed the new component model to enforce separation of computation and interaction modelling (presented in Chapter 6). Different component model entities define different aspects of system behaviour: components in our model can only define computation; connectors can only define interaction. This is possible by having coordination as the underlying composition mechanism. Connectors act as coordinators that trigger component execution and control data exchange between components. Unlike UML 2.0, both components and connectors are first-class entities in our component model. The model allows the definition of new components and new connectors, and both entities have an architectural representation.

The interaction in our component model is defined in terms of control flow and data flow to increase the expressiveness of interaction modelling. Both flows are explicit in software architecture as they are represented by different kinds of connectors. There are several pre-defined connector types: control connectors for basic control flow structures, such as sequencing and branching; data connectors for point-to-point data flow and data routing. These basic connectors can be composed to define more complex interaction. To allow separate modelling of control flow and data flow, we have proposed novel control-driven and data-driven execution semantics.

Figure 2.5 illustrates how these component model characteristics affect the architecture of the photo storage system. Notice that the component model

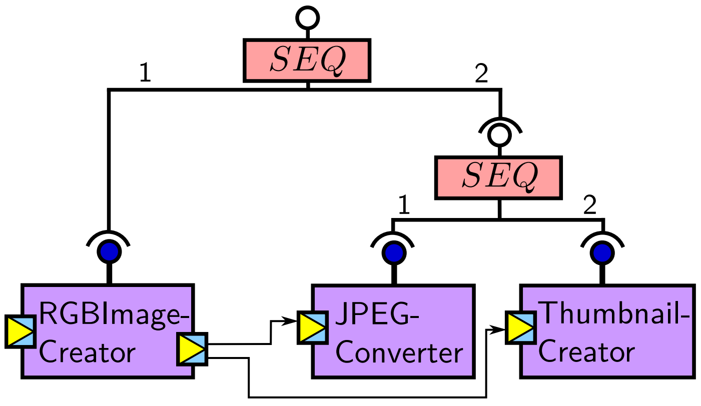


Figure 2.5: Architecture of Photo Storage System in Our Component Model

enforces the design more similar to Figure 2.4 than to Figure 2.3. Indeed, components cannot mix computation and interaction in our model; they can only represent computation. The components RGBImageCreator, JPEGConverter and ThumbnailCreator represent the same computation as the corresponding components in Figure 2.4. Nevertheless, component interfaces differ from the corresponding UML 2.0 components: they consist of data ports (depicted as little squares located at component sides) and control ports (the dark sockets on top of components).

The StorePhotoFacade component, which represented the interaction pattern in UML 2.0 architecture in Figure 2.4, has no corresponding component in this figure. This is because interaction in our model is defined by means of connectors. The interaction in this example is defined by two control connectors and two data connectors. Both control connectors in Figure 2.5 are sequencers. They route control flow from their control port (depicted as an empty socket) to their parameters (receptacles) in sequence (the numbers adjacent to sequencer parameters denote the order of sequencing). All data flow connectors in this example are also of the same type: FIFO data channels (arrows in the figure), which move data from an output port of a connected component to an input port of another connected component. The resulting interaction is identical to the one defined by StorePhotoFacade.

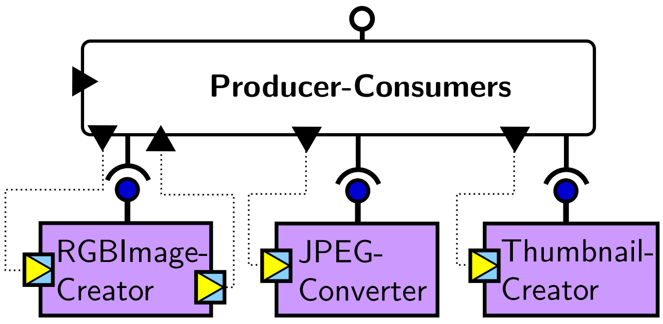
Interaction Patterns As Composite Connectors

Our component model allows the computation and interaction aspects of system behaviour to be modelled separately. Furthermore, new connectors can be defined out of existing ones by hierarchical composition. In this section, we show how these composite connectors can model complex interaction represented in software architecture explicitly by abstractions distinguishable from components. We illustrate with the example of the photo storage system that composite connectors in our approach provide abstractions for representing interaction patterns with a number of benefits over the component representation in UML 2.0.

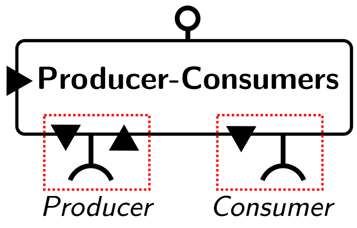
To represent complex interactions by a single architectural entity, we have defined a mechanism of connector composition (in Chapter 7). Existing data and control connectors can be composed to form new connector types, so-called composite connectors. Composite connectors compose control flow and data flow defined by their constituent connectors to form more complex interactions. Therefore, semantically they are still connectors; unlike components, they do not define any computation. Composite connectors are first-class entities in our component model: they have well-defined interface, semantics and an architectural representation; they can be composed further to form new composite connectors. Composite connectors enable our component model – unlike most existing component models – to have an extensible pool of high-level component composition mechanisms.

Furthermore, composite connectors are designed to be reusable. They are meant to be stored in a repository and be instantiated in a number of different systems. To increase their reusability, we have defined the concept of a connector template, which is a connector representation stored in a repository (see Chapter 8). Connector templates parametrise some aspects of connector interfaces and their internal structure. This allows a connector template to be instantiated as a number of different connectors, obtained by parametrising the template differently.

The ability to define complex interactions in a reusable manner makes composite connectors suitable abstractions for representing interaction patterns. We illustrate this with the example of the photo storage system.



(a) Photo Storage System



(b) Producer-Consumers Connector Template

Figure 2.6: Redesigned Architecture of Photo Storage System in Our Approach

Figure 2.6a shows the architecture of the photo storage system in our component model with the interaction pattern represented as the Producer-Consumers composite connector. The computation part of the architecture, represented by the three components, does not differ from the architecture in Figure 2.5. The interaction part of the architecture is represented solely by the composite connector. The connector routes data among the component ports connected to its required data ports (black triangles); it also routes control flow from its control port (an empty socket) to the components connected to its parameters (empty half-ovals, which we call receptacles). The behaviour of the composite connector is equivalent to the behaviour defined by the four basic connectors in Figure 2.5 since we want to enforce the same interaction pattern as before.

Figure 2.6b shows the interface of the connector template of Producer-Consumer, stored in a repository.[2] The template defines the interface elements and composition structure of its instances. A template's interface elements are grouped into roles, which correspond to the roles defined by the interaction pattern. In the figure, we see two roles for this interaction pattern: Producer and Consumer. Roles allow instances of a connector template to have different interfaces and inner composition structure by having a different number of participants for each role. The number of participants assigned to a role (role multiplicity) determines how many times the interface elements grouped in that role will be multiplied in the interface of an instance. Role multiplicities are specified during instantiation to suit a particular context. In our photo storage example, the Producer and Consumer roles have been assigned the multiplicities of one and two, respectively. Connector templates support this and other mechanisms to increase the reusability of interaction patterns they represent.

In this thesis, we argue that composite connectors are an abstraction suitable for representing interaction patterns in component-based systems. Compared to the representation of interaction patterns by components in most current component models, composite connectors in our approach bring the following benefits. Firstly, interaction-defining composite connectors are semantically distinguishable from computation-defining components in software architecture. Secondly, since the composite connectors are built up hierarchically by composing simple connectors, their composition structure defines their interaction behaviour in architecturally explicit way, unlike components, such as StorePhotoFacade, whose behaviour is defined by code that is invisible in software architecture. Thirdly, interaction in our approach is modelled by means of control flow and data flow; some component models, such as UML 2.0, cannot model these flows explicitly in software architecture. Fourthly, due to their port-based interfaces, composite connectors have greater reuse potential compared to interaction-defining components in component models with operation-based interfaces, since they do not depend on operation names or the order of parameters. Finally, the reuse potential of composite connectors is further increased by the variability mechanisms supported by connector templates, particularly their unique role-based structural

[2]For simplicity, we have not shown the inner composition structure of the template.

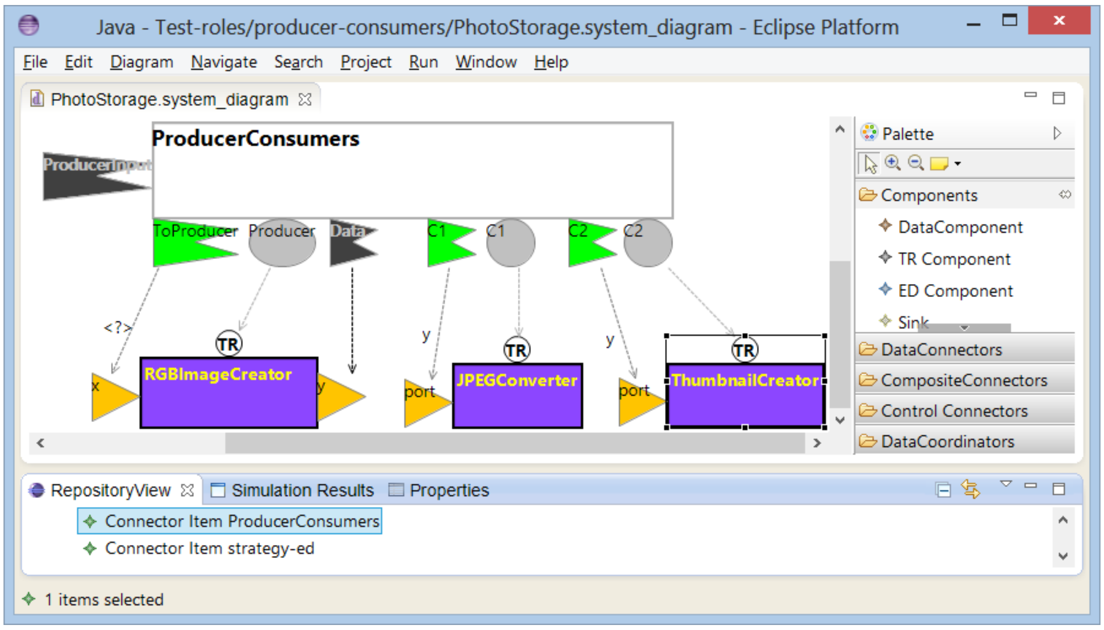


Figure 2.7: System Architecture Editor in our Prototype Tool

variability.

To support our approach, we have implemented a prototype tool for designing systems using our component model and their simulation (see Chapter 9). For example, Figure 2.7 shows the photo storage system in our tool's architecture editor. The tool is used in Chapter 10, in which we demonstrate the feasibility of our approach on a mode-switching interaction pattern for reactive control systems.

In the next chapter, we start the exposition of our research by defining interaction patterns.

Chapter 3

Interaction Patterns

In this chapter, we define a class of design patterns that define recurring units of interaction. These patterns, which we call *interaction patterns*, help software designers model computation and interaction separately. We use the domain of reactive control systems throughout the chapter to illustrate the discussed concepts; the domain also serves as an evaluation domain in the case study in Chapter 10.

Section 3.1 explains the concept of computation and interaction separation in software design, illustrates the concept with partitioning of reactive control software and also defines the notions of control flow and data flow. Section 3.2 gives some background on design patterns as a form of capturing software design knowledge. Interaction patterns are defined in Section 3.3. We conclude by presenting the Mode Switching interaction pattern in Section 3.4.

3.1 Separation of Computation and Interaction

The distinction between computation and interaction[1] in software specifications has been first emphasised by Gelernter and Carriero [47]. In this view, software systems can be defined as a mixture of computation and interaction by the following equation:

$$\text{Software System} = \text{Computation} + \text{Interaction}.$$

Computation corresponds to sequential expression evaluation, such as computing the result of a mathematical function with given input values. Interaction defines

[1]We use the terms coordination and interaction synonymously in this context.

the order in which computations are carried out and data exchanges between computations.

Gelernter and Carriero consider computation and interaction as orthogonal concerns that should be specified separately by means of different languages. Computation languages should focus on specifying computational activities; coordination languages should manage interaction between computations [47].

Such separation brings a number of benefits to system designs. The modularity of system designs increases. This helps to make designs more comprehensible. It also increases the reuse potential of separate computation and interaction modules because they can be reused independently of each other. It also potentially decreases the complexity of system verification.

These benefits have been acknowledged by a number of architectural patterns [69, 77, 101, 55, 105, 121] that promote the separation of interaction and computation in software architecture. We show examples of some architectural patterns for the domain of reactive control systems in Section 3.1.1.

3.1.1 Reactive Control Systems

Reactive systems [54] are systems that continuously react to their environment. Often, they control the operation of the physical devices they are embedded in, hence *reactive control systems*. Figure 3.1 shows a schema of such a system. It is embedded in an environment that it controls via actuators (also called effectors), such as robotic arms, and monitors through sensors, such as cameras or microphones. Examples of such systems vary from cruise control systems in cars to control systems preventing core meltdown in nuclear power plants.

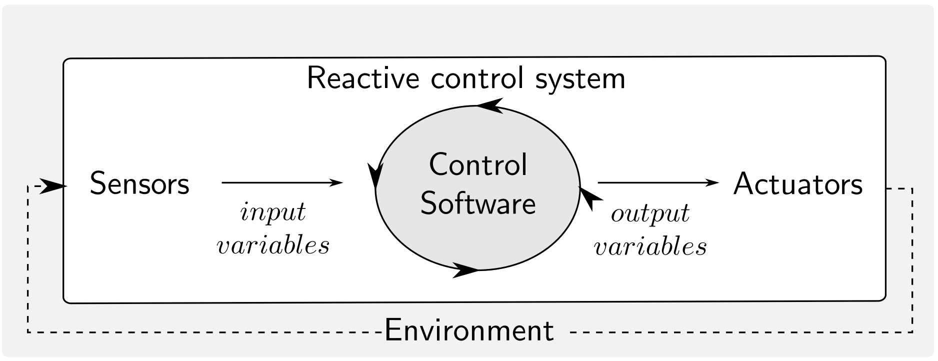


Figure 3.1: Schema of a Reactive Control System

The subsystem defining a system's reactions to data coming from sensors is often implemented in software (the grey circle in Figure 3.1). In terms of behaviour, control software corresponds to an infinite loop comprising reading and processing inputs, computing a reaction to the inputs and issuing outputs to actuators.

Separating computation and interaction in the architecture of control software amounts to partitioning the architecture into the part that comprises computation (data transformation, expression evaluation) and the part that describes interaction between computations. The former is called processing part and the latter is called controller, as shown in Figure 3.2.

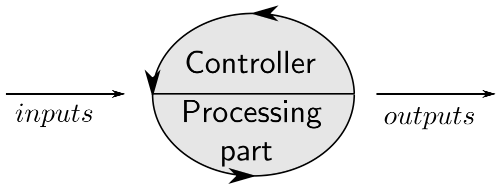


Figure 3.2: Control Software Structure

A number of architectural patterns in the domain recommend exactly such separation. For instance, Selic's Recursive Control Pattern [101], described in Section 2.1 and shown in Figure 2.1 on page 25, proposes such separation. The pattern defines two kinds of components that comprise control software: components responsible for computation (processing part) and components responsible for interaction between computation components (controller). Labbani et al. show such separation in Scade [69]. Again, some components in their architecture only deal with computation, while others only encapsulate interaction. Shaw, inspired by classic control theory, distinguishes controllers and processes in the process control architectural style (and its feedback and feedforward control variants) [105, 121]. Lea differentiates between functional and controlling components in avionic control systems [77]. Haslum identifies 'mode switchers' and 'higher-order task procedures' patterns for reactive programs [55].

3.1.2 Interaction Modelling

To specify computation and interaction separately in software design, we need different modelling abstractions for computation and interaction. In this section, we define control flow and data flow, two abstractions for interaction modelling.

Interaction between computations defines the order of execution of these computations as well as the exchange of data (inputs and outputs) between them.

Control flow represents the former aspect of interaction, data flow represents the latter one:

Control flow defines the ordering of some computation steps, such as machine instructions or function calls. It defines the ordering by describing the trajectory of the movement of control signals between computations in time: the presence of a control signal at a computation at a time signifies that the given computation is being executed at the time.

The origin of the control flow abstraction can be traced to Von Neumann's computational model [123]. The computer's central processing unit executes the instruction whose memory address is stored in its special register. The memory address corresponds to the location of the control signal. Imperative programming languages have adopted this notion of control; the basic building blocks of control flow thus correspond to sequencing, branching and looping known from these languages.

Control flow can be modelled in software design by diagrams used in various engineering disciplines since 1950s, whose variants are still present in the current modelling notations. For instance, UML activity diagrams show sequences (possibly alternative or iterated) of activities whose order of execution is determined by control flow. Figure 3.3 shows[2] the control flow of the request-response interaction between Client A and Server B: computations are drawn as rounded boxes and control flow induced ordering as solid arrows (the leftmost and rightmost circles represent the start and end of the interaction, respectively).

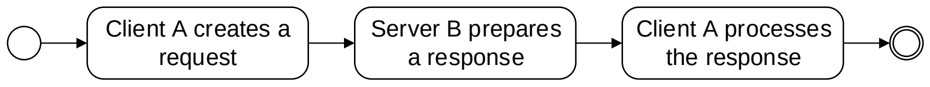

Client A creates a request
Server B prepares a response
Client A processes the response

Figure 3.3: Control Flow in the Request-Response Interaction

Data flow captures the topology of information exchange between several computations. It specifies which computations send their outputs to inputs of which computations.

[2]The depicted diagrams serve for illustration of the concepts of control flow and data flow only; they are not used for modelling interaction patterns in our approach.

Data flow diagrams have been introduced in 1970s in structured design methods [125] to depict data exchange between functions in hierarchical functional system decompositions and are still in use, especially for modelling information flow between business processes. Figure 3.4 shows the data flow diagram depicting data flow in the request-response interaction: bubbles correspond to functions, arrows represent data flow and their labels describe transferred data.

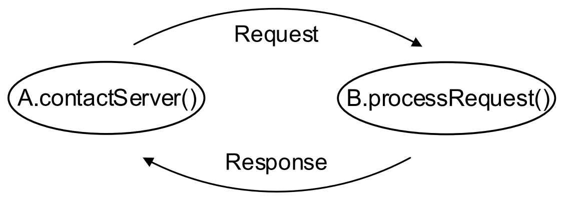


Figure 3.4: Data Flow in the Request-Response Interaction

3.1.3 Identifying Reusable Units of Interaction

Separate specification languages for interaction and computation enable separate modelling of these concerns in software designs. This allows partitioning software designs to parts specifying computation and parts specifying interaction, exemplified for the domain of reactive control systems in Figure 3.2. Consequently, the modularity of such designs increases, as does their comprehensibility. However, it does not mean that separate interaction modules can be reused in multiple contexts. To fully achieve the benefits of the separate specification of computation and interaction (including reuse), designers have to identify suitable units of interaction.

This is a challenging task since software designers are rather used to identifying units of computation, such as functions, or modules that mix both computation and interaction than separate interaction modules. We believe that, as with many challenging tasks in software design, the role of a software designer's experience is crucial. Experienced designers can distil recurring units of interactions from their knowledge of many existing system designs.

One of the most successful forms of capturing software design knowledge are design patterns [44, 100]. Many design patterns specify such recurring units of interaction. In our thesis, we focus on representing this class of design patterns as reusable entities in component-based development.

3.2 Design Patterns

In this section, we describe what design patterns are (Section 3.2.1), how they are used (Section 3.2.2) in current software design practice and what their benefits and shortcomings are (Sections 3.2.3 and 3.2.4). This section contains background material needed for the definition of interaction patterns in Section 3.3; the reader familiar with design patterns can skip this section.

Design patterns [44, 29, 28] are a means for capturing expertise of software designers; they prescribe the form in which the expertise should be captured. The pattern form, originally conceived by Alexander [3] to capture design knowledge in architecture of buildings, has been introduced to software engineering at the end of 1980s in the object-oriented community [15]. Following the publication of the seminal book Design Patterns by Gamma et al. [44], design patterns have become a well-known software engineering concept, as witnessed by dozens of published books, many conferences on patterns and their inclusion in software engineering curricula.

3.2.1 Pattern Form

A design pattern describes a recurring design problem and a well-proven solution to the problem. Additionally, a pattern discusses consequences of using the proposed solution and possible choices in the solution's adaptation and implementation. Solutions should be generic and independent of implementation details of a particular platform to make the pattern applicable in more contexts. Pattern solutions thus have to be specialised to fit into the context of their application.

Although there exists a variety of design pattern forms, the structure of most of them can be mapped to the following main constituents [28]:

Name A pattern's unique and apt identification.

Context A specification of the situations in which the design problem solved by a pattern occurs.

Problem A description of the problem that a pattern solves.

Solution Template The solution template identifies a set of entities, called participants, and defines their collaboration whose aim is to solve the problem addressed by the pattern. Participants play different roles in the collaboration and are defined in terms of their structure and expected behaviour with respect to their role. That is, a role states the minimum structural and behavioural requirements on the role's participants to solve the pattern's problem; participants usually have extra context-specific behaviour, apart from the one defined by their role. Therefore, a pattern solution is not a single fixed design; instead, due to its generality and variability, it defines an infinite set of complying designs. Each time a design pattern is used, the template is instantiated to fit its instantiation context.

Consequences A discussion of trade-offs of the pattern's solution, of different choices in the solution's adaptation and of various implementation hints.

3.2.2 Pattern Life Cycle

In this section, we describe the current practice in using design patterns in software development. We follow the life cycle of a design pattern from the initial design knowledge of the pattern's author to the instantiation of the pattern in a software design (see Figure 3.5).

In the beginning, a software designer accumulates a critical mass of experience in solving a recurring design problem; he/she observes that a certain solution (or rather a family of related solutions) has repeatedly satisfactorily solved the problem in practice and thus has the potential to be useful to other developers. The designer then embodies this design knowledge as a design pattern description in a particular form (e.g., the form used by Gamma et al. [44]). Before the pattern is published, it is usually reviewed by the pattern community at events, such as pattern writers' workshops, and improved based on their feedback.

Eventually, the design pattern is published. To be easier to find, the pattern may be included in some pattern catalogue, be it a book [44, 29, 97] or an electronic pattern library [94, 124].

Having been published, the pattern can be found by software developers who are interested in solving the pattern's problem. They read the pattern's description to understand the design knowledge embedded in the pattern by its author.

Equipped with such knowledge, they can use it when creating software designs. They need to match their particular design situation against the problem described by the pattern and to apply the solution template given by the pattern

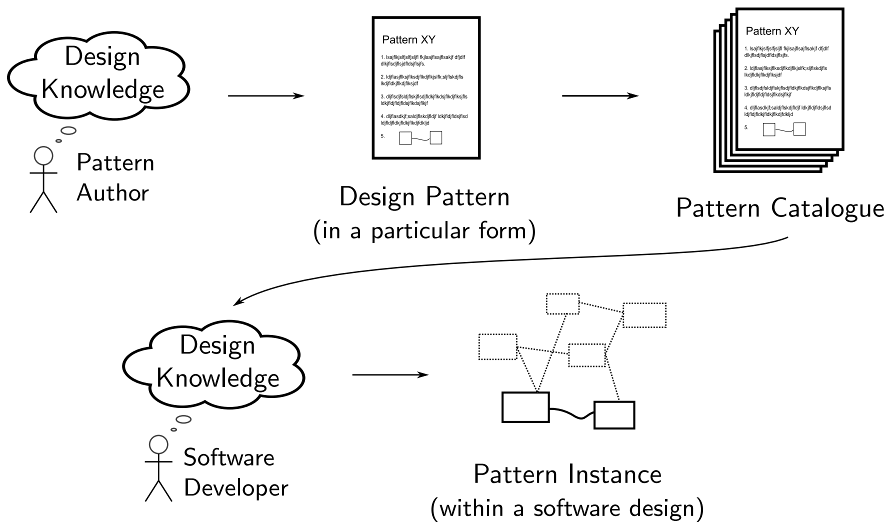


Figure 3.5: Design Pattern Life Cycle

to construct design artefacts (their structure and behaviour) corresponding to individual participants in the solution template.

Figure 3.5 illustrates that design patterns facilitate the transfer of design expertise from pattern authors to pattern readers, which results in reusing that expertise.

3.2.3 Benefits of Design Patterns

Design patterns bring a number of benefits to software developers, software designs and even to the software designed with patterns in mind [28, 98, 99].

Design patterns capture software design expertise and thus enable its reuse. The structured pattern form allows this expertise to be accumulated, categorised and searched in printed pattern catalogues and on-line repositories. Patterns thus form the ever-growing reusable design knowledge base.

Design patterns form the vocabulary of the language used by software developers to communicate design ideas concisely. This leads to raising the level of abstraction in communicating software designs.

Using design patterns in software development affects the qualities of the software. Because design patterns convey well-proven solutions to design problems, they embody best design practices. Using design patterns therefore promotes these good practices, which in turn helps to develop systems with some desirable qualities. There exists some empirical evidence supporting this pattern benefit:

e.g., stories of successfully applying patterns and achieving flexible [64] or less complex and maintainable [100] software systems. Of course, using patterns does not automatically guarantee those qualities; ultimately, it is the designer's responsibility to come up with good software designs. In addition, different patterns contribute to different, and possibly opposing, qualities; for example, flexibility and lower complexity are often in opposition and are therefore achieved by different patterns.

3.2.4 Shortcomings of Design Patterns

Since design patterns are defined informally in some natural language, they are prone to ambiguity and vagueness. And whilst it is in most cases sufficient for people to understand the core idea behind a pattern, their informal definition limits their usage. It precludes using formal methods, such as reasoning about patterns and their relations, and advanced tool support that would go beyond displaying pattern descriptions.

Another negative consequence of patterns' informal nature is the lack of their first-class representation in mainstream software design (such as UML) and programming notations (such as Java or C#). Although design patterns raise the level of abstraction in communicating software design, pattern instances are specified each time from scratch using low-level design entities, such as classes and methods, which developers then translate to low-level implementation constructs. Therefore, patterns lack explicit representation in the actual software design and implementation artefacts (e.g., UML models or source code of Java classes): the high-level abstraction of design patterns is lost, dissolved in low-level notations.

As a result, design patterns achieve reuse of design knowledge only; there is no direct reuse of design and implementation artefacts in traditional software development with patterns. Instead, developers need to incorporate pattern solutions into software designs and subsequently implement the designs manually.

3.3 Interaction Patterns Definition

In this section, we introduce interaction patterns as a class of design patterns that define recurring units of interaction in software designs. Interaction patterns are a novel concept, conceived for the purposes of this thesis, to formulate pattern characteristics that help patterns to be represented in component models as connectors. This section states the specifics of interaction patterns compared to generic design patterns.

We have identified design patterns as a potential source for reusable units of interaction for several reasons. Firstly, they provide reusable solutions to recurring design problems, distilled from design experience. This makes the solutions reusable. Secondly, their solution templates are generally formulated as collaborations between several kinds of participating entities. The focus of pattern solutions on defining collaborations – interactions between pattern participants – is in accordance with our goal of finding suitable units of interaction to reuse.

However, not every design pattern defines such a reusable interaction. Design patterns are a general means for structuring design knowledge and, although they follow the same form, they solve a variety of different problems, some of which focus on other aspects than the interaction between participants. For example, the Singleton design pattern [44] solves the problem of ensuring that a certain class is only instantiated once. It provides a solution defining a single participant and focuses on the structural definition of that participant.

To narrow down the scope of design patterns of our interest, we define a class of design patterns, whose solution templates focus on defining interaction between a set of participants. They primarily prescribe data flow and control flow between participants. Participants correspond to computational entities conforming to the requirements of the role they play in the pattern. We call such design patterns *interaction patterns*. Interaction patterns identify recurring units of interaction and thus facilitate separate computation and interaction modelling in software designs.

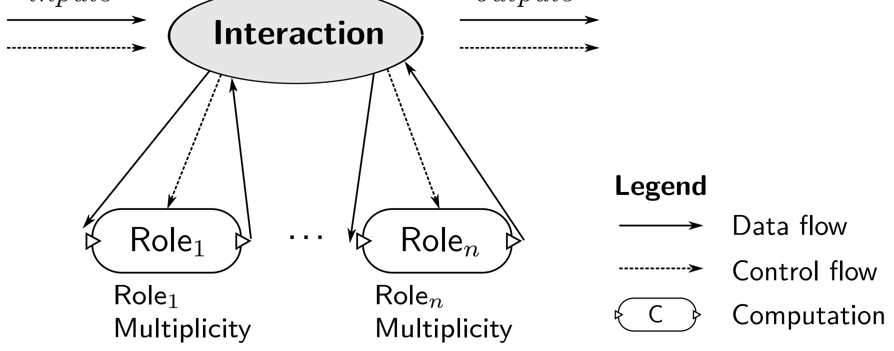


Figure 3.6: Schema of an Interaction Pattern's Solution Template

Figure 3.6 shows the schema of an interaction pattern's solution template. Interaction and computation can be clearly distinguished: the interaction is defined in terms of control flow and data flow; participating computations are defined in terms of roles. A role specifies requirements on entities that can participate in the interaction defined by the pattern. These requirements are defined in terms of control flow and data flow that a participant exchanges with other participants within the pattern. In addition, a role can prescribe some minimal behaviour that enables participants to join the interaction in the given role. Each role also has its multiplicity, which determines how many participants can play that role in any pattern instance.

A simple example of an interaction pattern can be seen in Figure 2.2 on page 26 in Section 2.1. In the next section, we present a real-world example of an interaction pattern from the domain of reactive control systems.

3.4 Mode Switching Interaction Pattern

In this section, we present an interaction pattern that defines a reusable interaction for reactive control systems with multiple modes. We adapted the pattern from the case study on which Labbani et al. [69] demonstrated the benefits of separating interaction and computation in Scade. We represent the pattern as a composite connector using our approach in the case study in Chapter 10.

Section 3.4.1 briefly describes what multi-mode systems are, presenting the context for the pattern. An analysis of the functionality of a controller in such a system, which explains the problem and gives the core idea of the mode switching pattern's solution, follows in Section 3.4.2. In Section 3.4.3, we formulate the pattern's solution template conforming to the interaction pattern definition from Section 3.3.

3.4.1 Systems with Modes

Systems with modes, also known as multi-mode systems, are systems whose behaviour can change during their execution. That is, the same inputs to such systems may result in different results at different times of their execution. The behaviour changes of these systems are restricted at design-time: a system with modes can exhibit one of finitely many predefined behaviours. The active behaviour is determined by a system's *mode*, a special system state. The overall behaviour of a system with modes can be represented using a state transition diagram determining the current mode.

In embedded control systems, the modes of control software often mirror the modes of controlled devices in which the software is embedded. For example, a simple burglar alarm can operate in five modes: active, ringing, inactive, PE1 (password entering) and PE2. The UML statechart diagram in Figure 3.7 captures the overall behaviour of the alarm system. In the active mode, the system

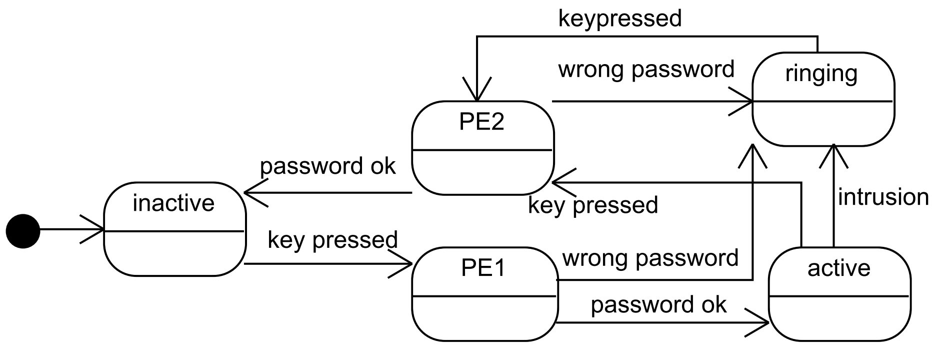


Figure 3.7: Modes of a Burglar Alarm System

uses data from its sensors to detect intrusion and to switch to the ringing mode, in which the alarm bell is ringing. In the inactive mode, the system simply ignores any sensor inputs and the alarm bell is not ringing. A user can switch between inactive and active modes by keying in the password, which happens in PE1 or PE2, depending on the subsequently active mode.

3.4.2 Mode Switching Controller

Reactive control systems with modes can be partitioned into controllers and processing parts, as depicted in Figure 3.2 in Section 3.1.1. Controllers – encapsulating interaction aspect of systems – are responsible for mode switching: they activate computations required in the current mode, deactivate computations active in the previous mode, feed data to those computations and collect their outputs. Processing parts consist of computation components realising the behaviour of systems in all their modes.

In general, every mode has an associated non-empty set of computation components realising the system behaviour in that mode, activated by the controller when the mode becomes active. For simplicity, we make an additional assumption that the associated set contains exactly one computation component different for every mode. Although the assumption rules out some interesting cases (such as components shared by several modes), it allows us to build a reusable mode switching controller with a straightforward behaviour. Table 3.1 summarises the responsibilities of controllers and processing parts in mode switching systems.

Controller (Interaction)	**Processing part (Computation)**
• activation of the current mode's computation component • deactivation of the previous mode's computation component • feeding inputs to the current mode's computation component • collecting outputs from the current mode's computation component	• a computation component for each mode • computation of the next active mode

Table 3.1: Responsibilities of a Controller and Processing Part

Such controller design is independent of (i) processing part and (ii) the mode state transition diagram of a particular system. It takes into account only the minimum set of concepts – current mode, (de)activation of a computation component, routing data inputs and outputs – common to all systems with modes. As a result, it has the potential to be reusable.

3.4.3 The Pattern's Solution Template

The partitioning of a multi-mode system into a controller and processing part, discussed in the previous section, addresses a common design problem and yields a reusable unit of interaction (the controller); it can thus be formulated as an interaction pattern. In this section, we present the solution template of the pattern.

The solution template of the Mode Switching pattern, conforming to the schema in Figure 3.6[3], is shown in Figure 3.8: the reusable interaction defined by the pattern is represented by the grey oval; participants (entities the pattern interacts with) are drawn as rounded rectangles; solid and dotted lines denote data flow and control flow, respectively. The template defines four roles:

[3]The only difference is that the figure here depicts participants rather than roles to indicate role multiplicities graphically.

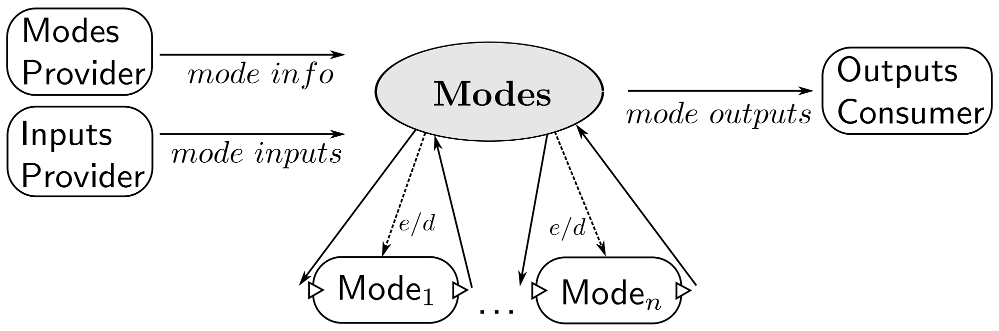


Figure 3.8: Schema of the Modes Interaction Pattern

Modes Provider provides the data (the current mode and, if the pattern is state-less, previous mode) based on which the interaction pattern switches the active Mode computation component.

Inputs Provider provides input data to be processed by the currently active Mode component.

Outputs Consumer consumes the outputs of the currently active Mode component.

Mode is a computation component that implements the system behaviour in a particular mode; the multiplicity of this role is $[2, \infty)$.

The interaction defined by the pattern is summarised in the Controller column in Table 3.1. Overall, the pattern maintains the invariant of having one active Mode component at a time (it therefore requires one Mode component to be enabled initially). Each Mode component has its inputs supplied and its outputs collected by the pattern. The pattern thus defines the data flow from Inputs Provider to the currently active Mode component, and it directs the component's outputs to Outputs Consumer. In terms of control flow, the pattern sends a control signal to activate or deactivate any connected Mode component, based on the data coming from Modes Provider. When the current mode changes, the pattern

1. disables the Mode component corresponding to the old mode,
2. starts redirecting input data to the newly activated Mode component,
3. enables the newly activated Mode component.

3.5 Towards Component-based Abstraction for Interaction Patterns

The main motivation for defining interaction patterns is to identify reusable units of interaction that help achieve the benefits of the separation of computation and interaction in software designs (see Section 3.1). As a class of design patterns, interaction patterns inherit the benefits and shortcomings of design patterns.

They positively influence software development practice (see Section 3.2.3) by enabling reuse of design knowledge, they facilitate the communication of software designs by increasing the level abstraction, and they have the potential to increase some qualities of the software systems developed using patterns.

On the other hand, the informal nature of the design pattern form makes interaction patterns deficient in a number of ways from a software engineering point of view (see Section 3.2.4). The ambiguity of their form precludes patterns from being reasoned about and having advanced tool support. Furthermore, patterns lack an explicit representation in current design and implementation notations, which causes their instances to be defined each time from scratch in low-level abstractions. As a result, design patterns can only achieve reuse of knowledge, but they fail to achieve the reuse of corresponding design and implementation artefacts.

However, interaction patterns form a narrow class of design patterns focused on separation of computation and interaction. Because they define their solutions in terms of a limited number of concepts – control flow, data flow and a set of computations – relevant in software design, their formalisation and first-class representation in software development seems more feasible.

In this thesis, we aim to find a suitable abstraction that would represent interaction patterns as explicit entities in software architecture and would be reusable in both design and implementation. To achieve this goal, we propose to define interaction patterns as first-class entities in component-based software engineering.

Chapter 4

Component Models

This chapter provides the reader with the background on component-based software development needed in the remainder of this thesis. It focuses on the key abstraction of a component model.

Before we define component models in Section 4.2, we give an overview of the basic aims and characteristics of component-based software engineering in Section 4.1. Sections 4.3 and 4.4 provide more details on existing types of components and composition mechanisms, respectively. Section 4.5 discusses the run-time aspects of component models and introduces related terminology. Finally, Section 4.6 characterises several existing component models, with which we compare our approach further in the thesis.

4.1 Component-based Software Engineering

Component-based software engineering (CBSE) [56, 114] is an area of software engineering concerned with building software from components. It aims to tackle the long-standing challenges of building software systems: to cope with their ever-growing complexity and to develop them in a systematic, scalable and predictable manner in order to minimise development times and budgets.

In CBSE, software systems are composed out of *pre-existing* components, reusable across many systems via repositories, rather than built from scratch. The component reuse promises to decrease system development effort (and thus cost and time).

The resulting structure of component composition defines the behaviour of a system and forms its architecture. Component-based software engineering attempts to make system construction compositional; i.e., the properties of composed components determine the properties of the system. This principle has the potential to assure system functional and quality (or non-functional) properties in a scalable manner.

An important characteristic of components is that they form building blocks of both design and implementation since they have both design time and implementation time manifestations. At design time, the structure of component composition forms the system's architecture (or at least its structural part). At implementation time, components are still useful abstractions as they have executable code that realises their functionality at run-time.

The focus of component-based development on reuse is manifested in its development process, which comprises two distinct development processes: a component development process and a system development process [35, 75, 73]. The former is an instance of development for reuse: it aims to develop components and store them in a repository. The latter is an instance of development with reuse: it aims to create systems by composing existing components stored in repositories. The existence of two separate development processes ensures that life cycles of components and systems are separate. This acknowledges that components are not system-specific; on the contrary, they are reusable across a number of systems through component repositories.

4.2 Component Model Definition

The exact definition of a software component is the central question of CBSE. McIlroy [82] was the first to formulate the idea of reusable software components. He envisaged reusable routines as software components. However, the idea has not gained much attention until three decades later when CBSE emerged as a branch of software engineering. The most cited definition comes from Szyperski [114]:

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party."

Szyperski considers the specification of a component's functionality and of its dependencies as a prerequisite for its reuse. He also stresses the independence of component and system development. However, such a universal definition has its limitations due to its generality. Additionally, it disregards component composition.

As a result, the concept of a *component model* has been conceived (i) to allow for more specialised definitions of components and (ii) to define the standards of component composition (and possibly of other aspects). A component model defines the design and implementation abstractions of component-based systems and how they are composed. Heineman and Councill [56] gave the following definition of software components using the concept of a component model:

> "A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard."

Several definitions of a component model exist. Table 4.1 illustrates the general consensus that a component model is a set of standards/rules that define (i) what components are, and (ii) how components can be composed to build systems (component composition mechanism).

Authors	**Definition**
Heineman and Councill [56]	"A component model defines a set of standards for component implementation, naming, interoperability, customization, composition, evolution, and deployment."
Lau and Wang [75]	"A software component model is a definition of (i) the semantics of components, (ii) the syntax of components, and (iii) the composition of components."
Crnkovic et al. [36]	"A component model defines standards for (i) properties that individual components must satisfy and (ii) methods for composing components."

Table 4.1: Definitions of a Component Model

Firstly, a component model specifies the semantics and syntax of components. The former defines the meaning of components – what they are (e.g., objects or services); how they behave at run-time (e.g., active or passive entities); what comprises component interfaces (e.g., sets of ports or methods); what comprises component realisation (code or sets of subcomponents). The latter defines particular specification languages and formats in which components should be developed and distributed (e.g., a particular programming language or interface description language).

Secondly, a component model specifies a component composition mechanism. Such a mechanism composes component behaviour by establishing interaction between composed components. Some component models have separate component model entities defining interaction between components – connectors, while other component models compose components via implicitly defined composition mechanisms.

Standards set by component models are necessary for components to be reusable without modification (adaptation). On the other hand, different application domains may have different requirements on what constitutes components and how they are composed. For instance, a dynamic composition mechanism binding components based on some constraints may not be suitable in resource- or performance-constrained environments. Component models thus allow the existence of several specialised sets of standards that do not have to conform to some universal component definition.

4.3 Components

In this section, we identify basic component characteristics in existing component models. First, we define the notion of a generic component, which embodies the component characteristics common to existing component models, in Section 4.3.1. Further, we present three main types of components, specialising the generic component, identified by Lau and Wang [75] by analysing existing component models. We use these component types in our component model survey in Chapter 5.

4.3.1 Generic Component

In general, components are units of composition providing some functionality. A component comprises an interface and realisation. A component's interface specifies the functionality the component provides to other components, and it may also state dependencies on other components or environment. The realisation implements the behaviour specified by the interface.

The interface of a generic component comprises provided services and required services. The former specifies the provided functionality, and it is a compulsory part of any component interface. The latter specifies the functionality, provided by other components, that this component needs to carry out its provided functionality. Some component types lack explicit declaration of their required services in their component interfaces. Figure 4.1 shows a schema of a generic component. Provided services are depicted as sockets; required services are depicted as receptacles.

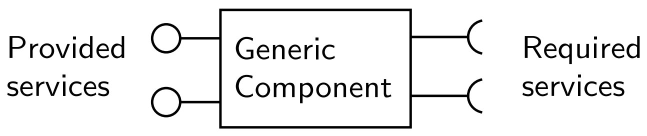


Figure 4.1: Schema of a Generic Component

There exist two main types of interface specification in existing component models: operation-based and port-based. Operation-based interfaces specify services as sets of operations, similar to functions and procedures in imperative programming. Operations have names, an ordered list of input and output parameters, which are possibly associated with their data types. Port-based interfaces comprise control or data ports. Data ports represent places through which data enter or leave a component; control ports are places through which a component receives or sends out control signals.

A component realisation implements the behaviour of provided services specified in the component's interface. Components may be either directly implemented in some programming language or, in some component models, composed of other components. Components of the latter type are called composite components or hierarchical components.

A component's interface and realisation are specified separately. This allows for a component's realisation to be hidden and promotes so-called black-box component reuse – reuse of components based on their interface specification only.

4.3.2 Component Types

Lau and Wang [75] have classified existing component models according to their component semantics into three categories: objects, architectural units and encapsulated components. In this section, we present these three component types as specialisations of the generic component. Example components of the following types can be found in Section 4.6.

Objects This component type corresponds to objects from object-oriented programming. They have operation-based interfaces, comprising sets of methods. Unlike the generic component, objects only declare provided methods in their interfaces. Dependencies on the methods of other objects are not part of their interfaces. Figure 4.2 depicts the interface of the object component type schematically.

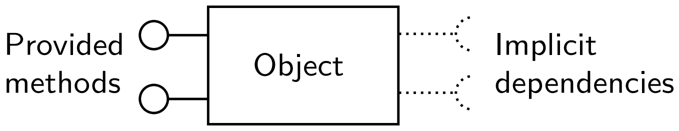
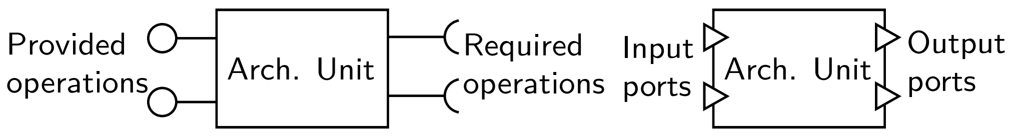


Figure 4.2: The Object Component Type

Objects are implemented by the code realising their interface. They are composed by method invocation: an object can call in its implementation code a method provided by another object. Existing component models based on objects include JavaBeans [112], EJB [113], OSGi [120] and MS COM [22].

Architectural Units Architectural units are units of functionality that – unlike objects – have a first-class architectural representation and strictly separate their interface and realisation. They correspond to basic elements of architecture description languages [105]. Their interfaces comprise both provided and required services. There exist two variants of this component type that differ in how they specify their interfaces: operation-based architectural units (see Figure 4.3a) and port-based architectural units (see Figure 4.3b).

Architectural units can be directly implemented by the code realising their interface (so-called atomic components), but many component models in this category also support composite components. Components are composed by connecting a provided and a required service of the same type (be



(a) Operation-based (b) Port-based

Figure 4.3: The Architectural Unit Component Type

it a port's data type or the same operation signature). Existing component models based on operation-based architectural units include UML 2.0 (Section 4.6.1), SOFA 2.0 [26] and Fractal [24]; component models with port-based architectural units are, e.g., ProCom (Section 4.6.2), Scade (Section 4.6.3) and Simulink (Section 4.6.4).

Encapsulated Components Components of this type are units of computation that do not depend on the services of other components to provide their own services. They are encapsulated in the sense that any incoming control signal is not redirected by a component to another component. Their interface therefore comprises provided services only (see Figure 4.4), but they do not have implicit dependencies like objects.

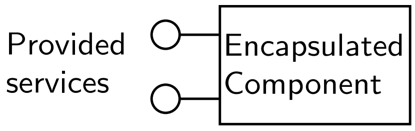


Figure 4.4: The Encapsulated Component Type

Like architectural units, they have a first-class architectural representation, and they separate their interface specification and realisation. They can be implemented in code (atomic components) or by composing other components (composite components). Encapsulated components are composed by means of external coordinators that invoke services of composed components and move data between them. Existing component models with encapsulated components include X-MAN (Section 4.6.5) and orchestrated web services (Section 4.6.6).

4.4 Composition Mechanisms

A composition mechanism is inherent part of the definition of a component model. It determines the way in which (at least two) components are composed. It defines the structure and behaviour of the resulting composition. In terms of structure, the result of composing a set of at least two components can be either another component of the same type or an entity of a different type, such as an assembly of the composed components with their interfaces linked in some way. The result of composition exhibits the behaviour that is derived from the behaviours of composed components. A composition mechanism is thus responsible for constructing complex behaviours out of simpler behaviours of composed components in a bottom-up fashion. The composed behaviour is generally the result of establishing interactions between composed components. To this end, composition mechanisms provide basic inter-component communication primitives, out of which more complex interactions can be built. Components can interact only after they have been composed.

In this section, we first specialise the notions of control flow and data flow to the context of component-based software engineering in Section 4.4.1. We then give examples of several composition mechanisms from existing component models in Section 4.4.2. Section 4.4.3 briefly describes connectors – component model entities representing composition mechanisms. Finally, Section 4.4.4 defines the class of exogenous composition mechanisms, which allow the separation of computation and interaction in software architecture.

4.4.1 Control Flow and Data Flow in Component Models

Inter-component interaction is defined as an exchange of data (inputs and outputs) and control signals (activation or triggering signals) between interacting components. The general interaction modelling concepts of control flow and data flow (see Section 3.1.2) are therefore also applicable in this context. They correspond to possible trajectories of data and control signals between component services (see Section 4.3.1). Control flow defines the ordering of execution of component services. Data flow specifies which component services send their outputs to inputs of which component services; data flow modelling also involves data stores (or buffers).

4.4.2 Existing Composition Mechanisms

Software composition mechanisms can be defined at different levels of abstraction [30]: from composition of programming language constructs, through composition of modules during system development, to composition of distributed systems over a network by various middleware mechanisms. Because components are abstractions spanning a more abstract architectural level and a more concrete implementation level, component composition mechanisms can be described at both of these levels of abstraction.

At the architectural level, Lau and Rana [71] identified two groups of composition mechanisms used in existing component models: connection and coordination.

Connection Composition mechanisms in this category establish interaction between two components by linking one component to an interface of another component.

Object-based component models use *method call* (or method delegation): an object can call a method provided via the interface of another object. Component models with architectural units use *port connection* (see Figure 4.5): a provided port of one component is connected to a required port

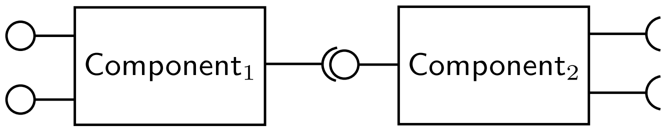


Figure 4.5: Composition of Architectural Units by Port Connection

of the same type of another component; the realisation of the latter component ($\mathsf{Component}_1$ in the figure) can indirectly invoke services of the former component ($\mathsf{Component}_2$ in the figure) through the connected required port.

Coordination Composition mechanisms in this category compose a set of at least two components by means of a coordinator that mediates the interaction by routing control signals and data between the components. Coordinators are semantically different from components. Control coordination of encapsulated components belongs to this category (see Figure 4.6).

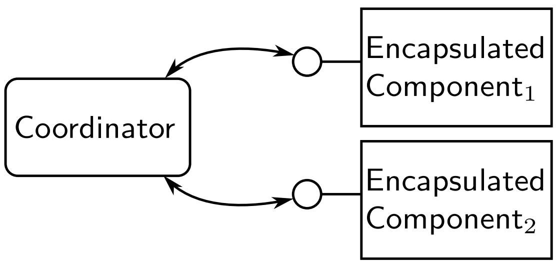


Figure 4.6: Composition of Encapsulated Components by Coordination

At the implementation level, composition mechanisms define the composition of component realisations. These include a number of different mechanisms [104, 84], ranging from method call[1] in object-oriented programming languages to various middleware mechanisms, such as remote procedure calls, message passing and data streaming.

In this thesis, we are not interested in particular composition mechanisms at the implementation level; however, we are interested in different ways in which they allow control and data to be exchanged between components. To abstract from implementation details of individual mechanisms, we use the concept of an interaction style [30, 36]. Interaction styles correspond to simple patterns of interaction known from software architecture [105]. Throughout the thesis, we use the following interaction styles:

Request-response This is a two-step, two-way interaction between two components, which comprises the exchange of control signals and data (see Figure 4.7). In the first step, the component that initiates the interaction sends the control signal to trigger a service of another component and also passes some input data needed by the service. The second step follows the completion of the called service; the called component returns the output data of its service, together with the control signal, back to the component that has initiated the interaction.

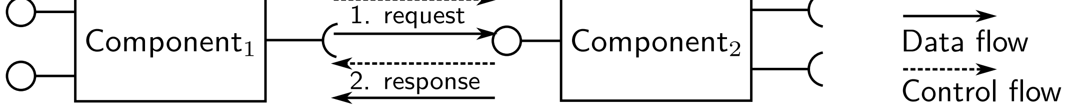


Figure 4.7: The Request-Response Interaction Style

[1]Method call appears at both levels since objects are both architectural and implementation abstractions.

Method delegation and most port connection mechanisms between architectural units with operation-based interfaces follow the request-response interaction style.

Pipe-and-filter This interaction style is based on a single-step, one-way interaction between two components (see Figure 4.8). Furthermore, this interaction style does not mix control flow and data flow since only one of them is being transferred between the two interacting components, depending on the types of composed interface elements.

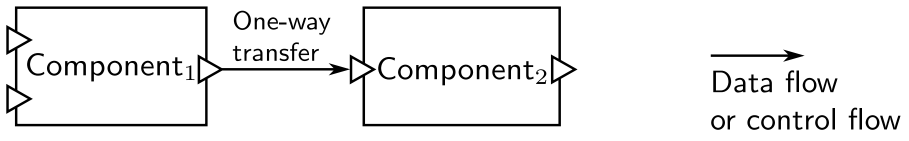


Figure 4.8: The Pipe-and-Filter Interaction Style

This interaction style is typical for composition mechanisms in component models based on architectural units with port-based interfaces, such as ProCom (see Section 4.6.2) or Scade (see Section 4.6.3).

4.4.3 Connectors

In the discipline of software architecture, two basic architectural entities have been traditionally distinguished: components and connectors [105]. Components deal with computation; connectors deal with interaction between components. These two notions form the basis of first-generation architecture description languages [83]. Although the purely structural definition of architecture has since been expanded to “the set of principal design decisions made about the system” [117], the component-and-connector view of a system remains part of the multi-view software architecture descriptions, proposed by Kruchten [68] and standardised in ISO/IEC/IEEE 42010:2011.

Component-based software engineering has largely adopted the notions of components and connectors from software architecture. Connectors are entities defined by a component model; they represent composition mechanisms in software architecture. However, the support for connectors is not universal among existing component models; indeed, the definition of a component model (see Section 4.2) does not make connectors – unlike components – compulsory component model entities.

As a result, some existing component models, many of which are based on objects, do not define connectors. Their composition mechanisms thus lack an explicit architectural representation. Most component models based on architectural units and encapsulated components define connectors. Furthermore, some component models, such as X-MAN [74] or SOFA 2.0 [26], even define the concept of connector composition, which allows them to create new connectors (so-called composite connectors) by composing existing connectors to represent more complex composition mechanisms.

4.4.4 Exogenous Composition Mechanisms

Composition mechanisms establish interaction between components. Different composition mechanisms vary in the extent to which they separate the specification of computation in components and the specification of interaction in connectors. In this section, we define two classes of composition mechanisms – endogenous and exogenous composition mechanisms – based on how they separate computation and interaction specification. We base our definition on the definition of exogenous and endogenous coordination languages by Arbab [4].

Arbab categorises coordination languages according to how the coordination[2] specification written in those languages is combined with the specification of computation [4]. He differentiates between endogenous and exogenous languages.

Endogenous languages mix coordination primitives with the specification of computation: the specifications of computation units *contain* coordination primitives. Exogenous models specify coordination of computation entities *from outside*: coordination and computation specifications are separate. We extend the definition of endogenous and exogenous coordination languages to the context of CBSE – where computation units correspond to components and coordination corresponds to composition – and define endogenous and exogenous composition.

Figure 4.9 illustrates the difference between the two types of composition mechanisms. It shows the specifications of two interacting components. In the case of endogenous composition, the specifications of both entities comprise a mixture of computation (white rectangles) and interaction (grey rectangles). In the case of exogenous coordination, the computation specifications of the interacting components are separate from the specification of their mutual interaction. The interaction aspect can thus be encapsulated and exist on its own.

[2]We consider the terms ‘interaction’ and ‘coordination’ indistinguishable in this context.

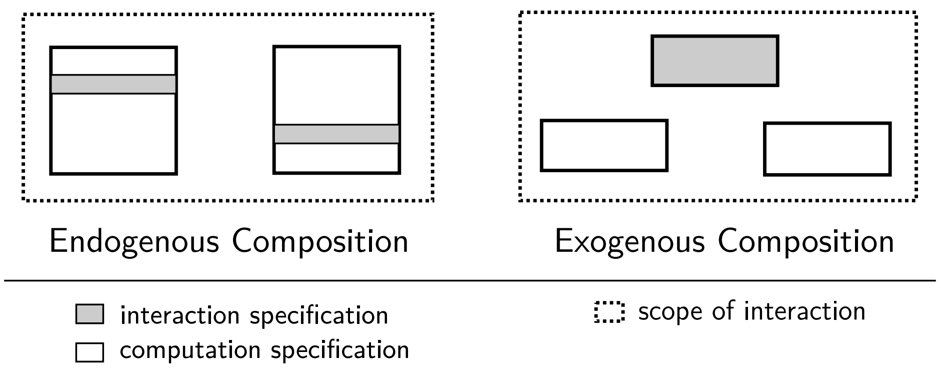


Figure 4.9: Endogenous vs. Exogenous Composition Mechanisms

For example, method delegation in object-based component models is an endogenous composition mechanism because it allows objects to mix in their implementation code the specification of interaction (with other objects) and the specification of their own computation. On the other hand, control coordination of encapsulated components in X-MAN [74] is exogenous, because components cannot define any interaction in their specification; the interaction is defined by connectors, which cannot define any computation.

4.5 Execution Semantics

Components conforming to a particular component model can be composed together to form executable software systems. A component model thus has to specify how components are executed at run-time and how component composition yields the behaviour of a system out of the behaviours of its components. These rules for component and system execution are called execution semantics. In this section, we define several terms, used further in the thesis, related to execution semantics. In particular, we distinguish between active and passive components, and we explain control-driven and data-driven system execution and their influence on interaction modelling in component-based systems.

4.5.1 Active and Passive Components

An important characteristic of execution semantics is how the execution of components is triggered. Two main alternatives exist: either components are continuously computing throughout the whole system execution or they are waiting for some external stimulus to trigger their computation. The components of the former type are called active; the components of the latter type are called passive.

Active components have their own thread of control throughout the whole of system execution, and they do not need any external triggering to perform computation. Each component defines its own internal control flow, which conceptually corresponds to an infinite loop with nested control flow structures that coordinate internal component computations. Therefore, there is no need for an exchange of control signals between active components; they communicate only by exchanging data.

Passive components are by default inactive. They need an external stimulus to start their computation, and they become inactive again once the computation finishes. Passive components typically define short computations, such as function execution. Because they rely on external stimuli from other components to determine when they should be triggered and what data they should process, both control flow and data flow may be meaningful in modelling interactions between passive components. Whether control flow modelling is meaningful depends on whether system execution is control-driven or data-driven.

4.5.2 Control-driven and Data-driven Execution

Another characteristic of execution semantics is what kind of stimulus triggers the execution of components. The execution can be either triggered by a control signal or simply by arrival of data. Accordingly, we distinguish control-driven and data-driven component execution.

Control-driven components are executed upon an arrival of a control signal. Component models with port-based component interfaces, such as ProCom [103], need two kinds of ports, control ports and data ports, and two kinds of port connections, for control flow and data flow. Component models with operation-based interfaces, such as UML 2.0, that compose component services by means of method calls cannot distinguish between the two flows. Only passive components can be control-driven.

The execution trigger of data-driven components is the presence of all their data inputs. For example, Scade [18], an industrial system modelling tool for the avionics domain, has data-driven components: their interfaces comprise input and output data ports and they are composed by port connection. As a result, architectures of such systems completely lack control. Both active and passive components can be data-driven. In models with passive data-driven components, an implicit system scheduler triggers a component's execution after it registers that the component's inputs arrived. Active components do not need such a scheduler as they possess their own thread of control. The relationship between active components and passive data-driven components is asymmetric: active components can mimic the functionality of data-driven components; the opposite is impossible in general (active components can perform some computation even in the absence of their data inputs).

Both kinds of computation triggering influence interaction modelling in system architecture. Data-driven execution renders control flow modelling redundant; control flow is thus absent in architectures of data-driven systems. Control-driven execution creates a dependency of data flow on control flow. Firstly, it is meaningless to have data flow going to a component without a corresponding control flow: the data will never be processed by the component, because it is never triggered. Secondly, there is a temporal dependency between the flows: input data have to arrive to a component before the control signal, otherwise the control signal cannot immediately trigger component execution.

Consequently, different computation triggering kinds are suitable for different application domains. Data-driven execution is suitable for data-processing systems (e.g., digital signal processing or some computationally intensive scientific calculations), in which modelling data flow dominates over modelling the exact order of executions by control flow. Control-driven execution is suitable for control-intensive systems (e.g., embedded control systems), in which control flow is the primary modelling concern. In fact, there exists a whole spectrum of systems (see Figure 4.10) and most systems fall between these two extremes: they combine control-oriented and data-oriented functionalities and their models therefore comprise a mixture of control and data flow.

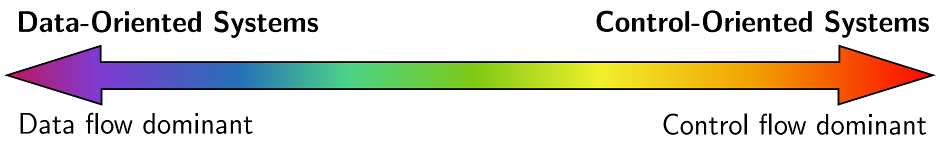


Figure 4.10: Spectrum of Systems from Data-oriented to Control-oriented

4.6 Existing Component Models

In this section, we introduce the following component models: UML 2.0, ProCom, Scade, Simulink, X-MAN, web services and Reo. They illustrate the terminology introduced earlier in the chapter, and we will compare our approach to them in Chapter 11. UML 2.0 represents component models based on architectural units with operation-based interfaces. ProCom, Scade and Simulink represent component models based on architecture units with port-based interfaces; they are specific to the domain of control systems and thus are closely related to the case study in Chapter 10. X-MAN and web services represent component models based on encapsulated components. Finally, Reo is a component model in which active components are composed by means of data flow coordination.

4.6.1 UML 2.0

UML 2.0 is a generic software modelling language, which also defines an architecture description sub-language for component-based system design (component diagrams).

UML 2.0 components are architectural units with operation-based interfaces. They are first-class component model entities: new components can be defined either by code that implements their interface or by hierarchical containment and by delegating their functionality to their subcomponents. Components have explicit architectural representation (as boxes).

UML 2.0 components are composed by connecting provided and required ports of different components associated with the same abstract data type. At implementation level, the underlying composition mechanism is method delegation. A component calls operations (methods) defined by the abstract data type associated with its required port; these calls are delegated to the same operations exposed by the connected provided port of another component.

The composition mechanism is represented in architecture by connectors. Unlike components, connector types are fixed, and new connectors cannot be defined. In architecture, they are represented as links between component ports. UML 2.0 supports two connector types: assembly and delegation. The former represents a connection between two ports,[3] one required and one provided, on the same

[3]Somewhat confusingly, UML 2.0 components communicate via 'ports' that – unlike data and control ports in port-based component models – are associated with interfaces consisting

level of composition; the latter connects two ports of the same type, one of which belongs to a composite component and the second of which belongs to the sub-component realising the delegated interface.

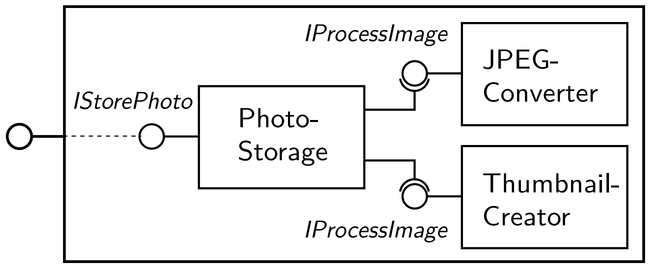


Figure 4.11: UML 2.0 Composite Component

Figure 4.11 shows an example of a UML 2.0 composite component. The component contains subcomponents of the photo storage system from Figure 2.3 on page 28. Components are represented as boxes with sockets for provided interfaces and receptacles for required interfaces. The figure gives examples of both connector types: see, e.g., the assembly connector between PhotoStorage and JPEGConverter and the delegation connector exporting the provided port of PhotoStorage to the interface of the composite component (shown as a dashed line).

4.6.2 ProCom

ProCom is a research component model for the domain of control-intensive systems [103]. ProCom components are architectural units with port-based interfaces. ProCom explicitly models control flow and data flow by means of two kinds of connections, and it also contains connectors for more complex control and data flow routing. It is a control-driven model, based on the pipe-and-filter interaction style.

The ProCom component model comprises two modelling layers, ProSys and ProSave, which enables system modelling on two different levels of abstraction. The former focuses on modelling subsystems at a higher level of abstraction (active components exchanging messages); the latter allows more detailed modelling of interaction by means of data flow and control flow. Both layers are integrated: ProSys subsystems can be represented by ProSave composite components. In this thesis, we only consider ProSave because of its focus on interaction modelling.

of operations.

ProSave components are units of functionality, whose interfaces comprise trigger and data ports. Ports are aggregated into port groups, containing one trigger port and several data ports; port groups are further organised into services, each of which comprises one input port group and one or more output port groups. A ProSave component's interface is defined as a set of services. Components can be composed hierarchically by port connection. At run-time, components are passive entities: their services need to be triggered by a control signal arriving at their input trigger port to be executed. ProCom thus has control-driven execution semantics.

The interaction between subcomponents is modelled by means of control flow and data flow. ProCom uses connections (arrows connecting either a pair of trigger or data ports) to model point-to-point transfers of control or data, and connectors (rounded rectangles routing control flow or data flow) to express more complex routing behaviour.

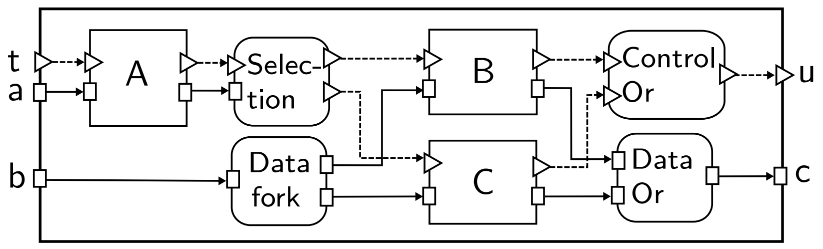


Figure 4.12: ProCom Composite Component

Figure 4.12 depicts an example ProCom composite component. It comprises three subcomponents: A, B and C, each of which has a single service consisting of trigger ports (triangles) and data ports (rectangles). The connections model control flow (dashed arrows) or data flow (solid arrows). The composite component also comprises the following connectors: Selection directs its input control flow to either B or C, depending on its data input; Data fork copies its data input to its outputs; Control Or and Data Or merge their two input flows to produce a resulting flow of the same type. Overall, the composite component delegates the processing of input b either to B or C, depending on A's decision based on input a, and outputs the result via output port c. The execution starts when t receives and consumes a control signal; when the computation is complete, u emits the control signal.

4.6.3 Scade

Scade is an industrial tool for designing critical reactive systems, used mainly in the avionic and automotive domains [18]. Its underlying component model is based on the data flow programming language Lustre [51]: components are passive functional blocks with data ports, composed by means of data connections. Scade conforms to the pipe-and-filter interaction style. Scade is not control-driven since control flow does not trigger computation and cannot even be expressed explicitly; instead, components are executed according to the synchronous data flow execution semantics [17].

Synchronous data flow is a particular realisation of the synchronous paradigm (explained in Section 6.5.4). System execution is driven by the basic clock – each system reaction corresponds to one tick of the basic clock. The clock represents an implicit form of control, rendering modelling control flow in system designs redundant.

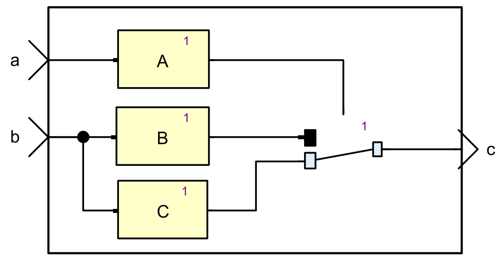


Figure 4.13: Scade Composite Component

Figure 4.13 illustrates that control flow is absent in Scade architectures; it shows a composite component implementing the behaviour of the component from Figure 4.12. All connections represent data flow. There is no control flow to selectively trigger B or C; instead, both components process input b and the if-else block chooses (based on A's output) one of their outputs to become the output of the whole composite.

4.6.4 Simulink

Simulink is an industrial tool [106] for modelling and simulation of dynamic systems. Components (called blocks) have port-based interfaces; they correspond to equations specifying the relations between input and output ports. They are composed via data flow connections. Simulink is primarily used for simulating continuous systems, but can also simulate discrete systems, including software systems.

During simulation, a Simulink system is represented as a set of ordinary differential equations that depend on the time variable. Simulation amounts to the repeated evaluation of the equations for certain values of the time variable. The simulation behaviour is therefore influenced by the selected method for solving the equations. Compared to Scade, the execution semantics is not formally defined, but it depends on a chosen implementation of a particular technique for finding a solution [122].

Despite modelling data flow only, Simulink features a richer palette of modelling constructs than Scade, some of which try to mimic control flow. For example, it supports so-called triggered and enabled subsystems. Both have an extra data port that represents an incoming control signal. Subsystems of the former type produce outputs only if their 'control' port values change (e.g., from 0 to 1); subsystems of the latter type produce outputs while their 'control' signal remains non-zero.

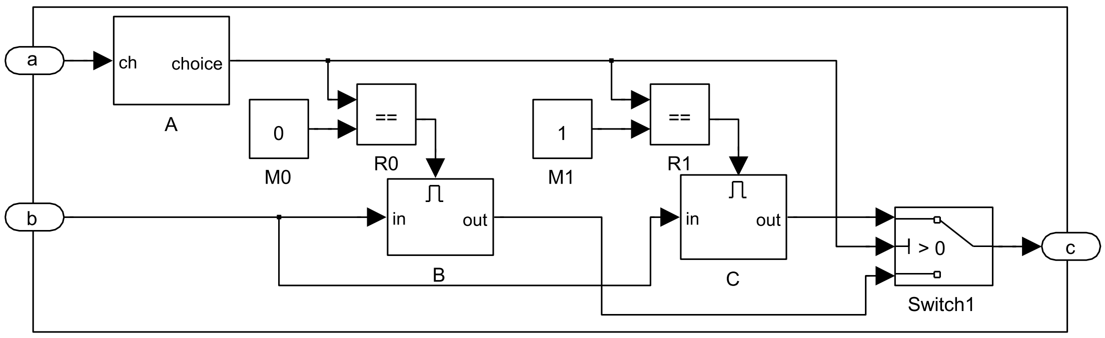


Figure 4.14: Simulink Composite Component

Figure 4.14 shows a Simulink composite component that has the same behaviour as the ProCom composite component in Figure 4.12. All components have data ports only, and the connections between them represent data flow. Components B and C exemplify enabled subsystems (notice their extra ports denoted as square waves): they are active only when A's output equals 0 or 1, respectively.

4.6.5 X-MAN

X-MAN [74] is a component model based on encapsulated components. It has the control coordination composition mechanism. As a result, connectors are explicit in architecture and can be composed to define composite connectors.

Components in X-MAN are passive units of computation with operation-based interfaces. Because components do not call operations of other components, they encapsulate control flow – the control flow that enters a component to trigger one of its services does not leak to other components; instead, it returns back upon the completion of the triggered operation. Components can be either atomic, with their operations' behaviour defined in some programming language, or composite, i.e., composed out of other components by means of connectors.

Connectors in X-MAN coordinate component execution. Basic connectors correspond to control flow structures known from imperative programming: sequencing, branching and looping. Connectors can be composed to form more complex control coordination behaviour. X-MAN connectors enable the construction of composite components. Each connector composing components $C_1, \ldots, C_n$ represents a composite component (with $C_1, \ldots, C_n$ as subcomponents), whose behaviour (and interface) is defined by the connector out of the behaviour (and interfaces) of the subcomponents.

X-MAN lacks explicit data flow modelling. The general rule is that connectors can also pass data along the control flow they define. For instance, the pipe connector defines (on top of its sequencing coordination behaviour) data flow between the composed components: it passes the results of some of the operations of the first component as input parameters to some operations of the second component. However, the flow of input parameters of a composite component's operations to the subcomponents and, conversely, the flow of subcomponents' outputs to the outputs of the composite component are left unspecified in X-MAN.

Execution of X-MAN systems is control-driven. The initial control flow enters the top of the control connector hierarchy and traverses it according to the coordination semantics of constituent connectors. When the control flow reaches a component, it triggers its execution and then returns back. Ultimately, a system terminates when the control flow finishes the traversal of the system's connector hierarchy.

An X-MAN realisation of our example composite component from Figure 4.12

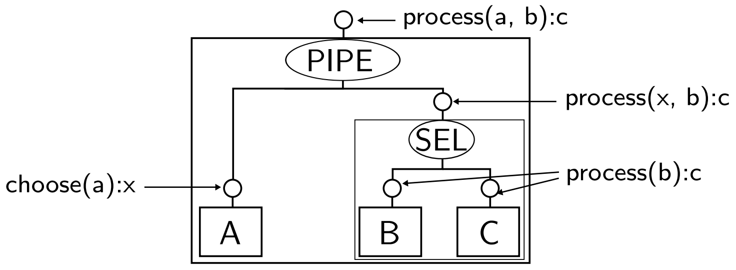


Figure 4.15: An X-MAN Composite Component

is depicted in Figure 4.15. The atomic components A, B and C are composed by two control connectors: pipe and selector. The composite component's interface provides a method[4] process(a, b):c, which corresponds to the port-based interface in Figure 4.12. When the method is invoked, the pipe first triggers A, passes A's output to the selector, which is triggered second. The selector then triggers either B or C, based on the A's output.

4.6.6 WS-BPEL Coordinated Web Services

WS-BPEL (Web Services Business Process Execution Language) [61] is a standardised language for orchestration of web services. Orchestration is a type of service composition based on exogenous coordination: individual existing web services are composed by means of a coordinator, which is called a process in WS-BPEL, into another web service(s). Thus, WS-BPEL is an exogenous coordination language. It can be also considered as a component model, in which web services are components and WS-BPEL processes represent composition mechanism [75].

Web services are passive components exposing their functionality through operation-based interfaces over the World Wide Web. Operations may or may not return results, thereby supporting one-way or two-way remote procedure call. They are based on a set of standards centred around XML (Extensible Markup Language): WSDL (Web Service Description Language) for interface description, SOAP (Simple Object Access Protocol) for exchanging structured data over a network and UDDI (Universal Description, Discovery and Integration) for locating web services. This reliance on XML-based standards and their inherently distributed nature helps web services achieve inter-operability.

A WS-BPEL process coordinates several web services through their WSDL

[4]Method signatures serve for illustration only: they do not contain type declarations.

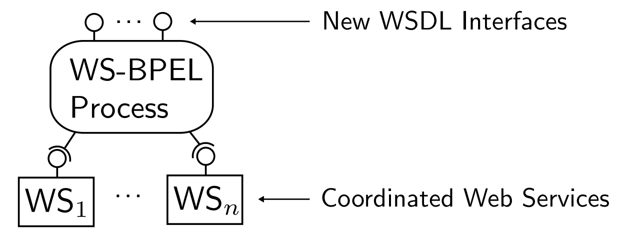


Figure 4.16: Web Service Composition in WS-BPEL

interfaces by defining a workflow, and it exports the resulting functionality as web services via other WSDL interfaces, thereby defining new composite web services (see the schema in Figure 4.16). The workflow is defined by control flow constructs: sequencing, branching, looping and concurrent execution. The execution semantics is control-driven. Data flow is modelled indirectly, in the imperative programming style, by assignments to mutable variables, which can store inputs and outputs of web service invocations. The reader can find an example of a WS-BPEL process in Section 11.2.2.

4.6.7 Reo

Reo [5, 6] is a dataflow-oriented, exogenous coordination language. It defines a set of basic data channels, which can be composed into so-called connectors. Connectors act as coordinators of components. Reo focuses on composition [7] and leaves some details of the definition of components open. However, it makes some assumptions about components in order to guarantee their composability by Reo connectors. Therefore, we consider Reo to be a component model.

Data channels are basic building blocks of Reo connectors. They are data flow connections with two ends. A channel end can be of two types: a source (a data entry point to a channel) or a sink (a data exit point from a channel). Combinations of different channel ends, channel buffer sizes and associated synchronisation constraints give rise to different data channels. Data channels are composed into connectors by means of nodes. Nodes join several channel ends and actively move data among the connected channels while they can. Nodes also form connector interfaces.

Reo components are active. They are composed via interface nodes of Reo connectors and can invoke some pre-defined operations on these nodes, e.g., to read a value or to write a value. The node operations can block component execution if they cannot be performed immediately. This way, Reo connectors coordinate components they compose.

Reo connectors model data flow explicitly in architecture. However, control flow is implicit because it is defined within the code of Reo components. The reader can find examples of Reo connectors in Section 11.2.3.

Chapter 5

Representing Interaction Patterns in Component Models

Our research aims to represent interaction patterns explicitly in architecture and to reuse associated design and implementation artefacts in component-based software development. To achieve this aim, we need to define interaction patterns as an abstraction within a component model.

In this chapter, we formulate the basic principles of representing interaction patterns underlying our approach [109], and we use them to survey existing component models. The results of the survey confirm that current component models are not suitable for this purpose. This motivates us to define a new component model suitable for expressing interaction patterns (in Chapter 6).

Section 5.1 explains the choice of connectors as a component model abstraction for representing interaction patterns. In Section 5.2, we identify a number of characteristics that a component model should possess to be able to express interaction patterns according to our research goals. The survey of current component models with respect to these characteristics follows in Section 5.3.

5.1 The Abstraction for Interaction Patterns

Component-based software engineering is focused on reuse of design and implementation artefacts and on representing component model abstractions in system architecture (see Section 4.1). In this thesis, we aim to define interaction patterns as first-class CBSE abstractions – defined formally within a component model – in order to achieve full reuse of design expertise and of design and implementation artefacts associated with interaction patterns.

Existing component models are based on two main abstractions, components and connectors, originally conceived in software architecture [105]. In this paradigm, components are units of computation, connectors deal with interactions between components and systems are ensembles of components composed by means of connectors. Interaction patterns focus on defining collaborations between sets of participants. They primarily prescribe data flow and control flow between participants (Section 3.3). Participants, on the other hand, are entities endowed with some system-specific behaviour (computation); they have to conform to the requirements of the interaction pattern to be able to join the pattern-defined collaboration.

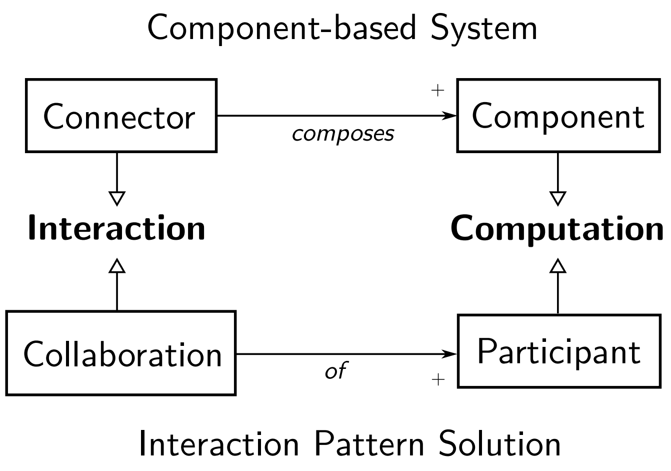


Figure 5.1: Central Role of Interaction in CBSE and Pattern Solutions

Figure 5.1 shows the entities comprising meta-models of both component-based systems and interaction pattern solutions.[1] It illustrates the semantic correspondence between connectors and collaborations defined by patterns on the one hand, and components and pattern participants on the other hand. Collaborations defined by interaction patterns are semantically closer to the abstraction of connectors since both entities share the goal of establishing interaction between a set of other entities. Components are better match for representing participants as they both exhibit computation behaviour.

Therefore, it is our research hypothesis that *interaction patterns can be defined as connectors and used as high-level composition mechanisms for composing software components.*

[1]The plus sign in Figure 5.1 denotes the multiplicity of “at least one”.

Using different abstractions for representing an interaction pattern itself and pattern participants is of prime importance in our research. It allows us to distinguish both entities semantically at the level of system architecture and to maintain the separation of computation and interaction in system design, which is one of the main motivators of our research (see Section 3.1).

Representing interaction patterns as connectors, encapsulating control flow and data flow of the pattern collaboration, impacts the underlying component model. In the next section, we identify characteristics that a component model suitable for expressing interaction patterns as connectors should possess.

5.2 Desirable Component Model Characteristics

In this section, we identify characteristics that make a component model suitable for representing interaction patterns. Since we propose to define interaction patterns' solutions as connectors, explicit in system architecture and reusable via repositories, most characteristics state requirements on the composition mechanism part of a component model specification.

In particular, a component model suitable for our aims should have the following characteristics:

- explicit architectural representation of connectors,
- the ability to define new connectors,
- separate specification of interaction and computation,
- explicit control flow and data flow modelling,
- composable connectors,
- separate control flow and data flow modelling.

In the rest of this section, we explain the above characteristics in detail and discuss their implications.

Explicit Architectural Representation of Connectors

A minimal requirement is that connectors must have an explicit representation in software architecture. In the absence of such a representation, our aim of interaction patterns being first-class architectural abstractions (see Section 3.5) would be thwarted.

Ability to Define New Connectors

This allows developers to extend the set of available connectors in a particular component model by defining new ones (without re-defining the component model). Since there exists a large and ever growing pool of interaction patterns, the ability to define new connectors representing these patterns is needed. This requirement implies that the complying component model needs some connector specification language for defining new connectors.

Separate Specification of Interaction and Computation

Although connectors by definition mediate inter-component communication, the extent to which the underlying composition mechanism separates the specification of computation and the specification of interaction varies (see Section 4.4.4). For our purpose of encapsulating the interaction between participants of an interaction pattern, we need the composition mechanism to separate the behaviour specification of pattern participants from that of their mutual interactions. This enables the interaction to be encapsulated in the form of a connector, which can stand on its own, without tight coupling to particular components (representing participants).

In Section 4.4.4, we have defined a class of composition mechanisms – called exogenous composition mechanisms – that separate the specification of connectors from the specification of components. The underlying composition mechanism in a component model suitable for expressing interaction patterns thus needs to be exogenous.

Explicit Control Flow and Data Flow Modelling

A component model supports explicit control flow and data flow modelling if it defines entities that represent control flow and entities that represent data flow. As a result, a system architecture constructed in such a component model explicitly models control flow and data flow.

Since we intend to model interaction by means of control and data flow (see Section 3.1.2), a component model with explicit control flow and data flow gives us means to define both aspects of interaction patterns, resulting in their more precise representation.

Composable Connectors

Connectors in a component model are composable if there exists a mechanism for building new connectors out of existing connectors – connector composition. In other words, the connector specification language allows composing new connector specifications (of so-called composite connectors) out of existing ones. This requirement is closely related to one of the previous requirements: connector composition is one possible way of creating new connector definitions.

The compositional representation of interaction patterns allows complex patterns to be defined as composition of simpler patterns. A component model can thus only define a fixed set of basic connectors, corresponding to simple interaction patterns, out of which all other patterns can be composed. Furthermore, composite connectors are defined explicitly in software architecture, as opposed to using some connector definition language without explicit architectural representation.

Separate Control Flow and Data Flow Modelling

Most current component models cannot specify control flow and data flow independently. There are usually some constraints, related to a component model's execution semantics, that make one of the flows dominant in interaction modelling: control flow modelling is redundant for data-driven component models since the mere presence of all inputs triggers component execution; in control-driven systems, data flow and control flow are inter-dependent (see Section 4.5.2).

Figure 4.10 on page 67 illustrates that different domains require different proportion of control flow and data flow in their system designs. Component models with separate control flow and data flow modelling can model both flows explicitly and independently of (without any constraints with respect to) each other. Consequently, they could be used to model interaction across the whole spectrum of systems, from data flow dominated interactions all the way to control flow dominated interactions. As a result, such models would be able to represent a greater variety of interaction patterns.

5.3 Analysis of Existing Component Models

In this section, we assess the suitability of current component models for expressing interactions patterns. We analyse component models aggregated from two major component model surveys found in the literature [75, 36] with respect to the requirements identified in Section 5.2.

Section 5.3.1 defines the set of analysed component models; Section 5.3.2 outlines the overall methodology of the analysis; the analysis is carried out in three iterations in Sections 5.3.3, 5.3.4 and 5.3.5; Section 5.3.6 presents analysis conclusions.

5.3.1 Surveyed Component Models

To identify existing component models, we use two major component model surveys by Lau and Wang [75] and by Crnkovic et al. [36]. The former is the most cited component model survey [81]; the latter is more recent and surveys more component models. Table 5.1 gives an overview of component models identified in each survey.

Survey	**Surveyed Component Models**
Lau and Wang	.NET, ACME, Corba CM, EJB, Fractal, JavaBeans, KOALA, KobrA, MS COM, PECOS, SOFA 2.0, UML 2.0, Web Services
Crnkovic et al.	AUTOSAR, BIP, BlueArX, Corba CM, COMDES II, CompoNets, EJB, Fractal, IEC 61131-3, IEC 61499, JavaBeans, KOALA, KobrA, MS COM, OpenCOM, OSGi, Palladio, PECOS, Pin, ProCom, ROBOCOP, RUBUS, SaveCCM, SOFA 2.0
Our Survey	all of the above + ACME, IEC 61131-3, IEC 61499 excluded and X-MAN, Reo, Scade, Simulink added

Table 5.1: Component Models Identified in Different Surveys

In our analysis, we additionally include X-MAN [74], Reo [5], Scade [18] and Simulink [106], the four component models with which we compare our approach in Chapter 11. The reasons why these component models have not been included in the two cited surveys vary: X-MAN is a relatively new research component model, which was not mature enough at the time when the surveys were published; Reo focuses on defining connectors and components are specified only implicitly (e.g., no component definition language is prescribed); Scade and Simulink, although they implicitly define component models, are generally more known as industrial tools for developing control software systems.

We exclude ACME [45] since it is a language for exchanging architectural descriptions without any fixed semantics; we also exclude IEC 61131-3 and IEC 61499, standards for modelling and implementing programmable logic controllers, because they do not clearly define the notion of components [111] and thus do not fit our component model definition from Section 4.2. Overall, we survey twenty-nine component models.

5.3.2 Methodology and Results

Because of the high number of analysed component models, we did not evaluate all the characteristics from Section 5.2 for all the component models; instead, we took an iterative approach.

In each iteration, we focused on just a subset of analysed characteristics, and we evaluated component models with respect to these characteristics. In the following iteration, we chose another subset of characteristics to analyse but only included component models that had passed the previous iteration. Thus, we gradually filtered out component models without the desired characteristics.

We carried out our survey in three iterations. In the first iteration, we evaluated the architectural representation of connectors in all component models. In the second iteration, we evaluated the ability of component models that had passed the first iteration to define new connectors. In the third interaction, we evaluated all the remaining characteristics for all the remaining component models. The results of the survey are analysed in the next sections. For completeness, Table 5.2 shows the survey results (the horizontal lines separate the results obtained in different iterations).

5.3.3 Connectors' Architectural Representation

The basic characteristic required of a component model for expressing interaction patterns as connectors is an explicit representation of connectors in system architecture. However, many component models fail to comply with this requirement since their composition mechanisms have no architectural representation.

Among the analysed models, we have identified three types of connector representations: (i) connectors are absent from architecture, (ii) connectors are represented as links (connections) between elements of component interfaces, and (iii) connectors are more complex architectural units with their own identity and possibly reusable on their own. The component models belonging to the second and third category represent connectors explicitly in software architecture, and

Component model	**EC**[1]	**NC**[2]	**SCI**[3]	**ECF**[4]	**EDF**[5]	**SCD**[6]	**CC**[7]
.NET	×	–	–	–	–	–	–
Corba CM	×	–	–	–	–	–	–
CompoNets	×	–	–	–	–	–	–
EJB	×	–	–	–	–	–	–
JavaBeans	×	–	–	–	–	–	–
MS COM	×	–	–	–	–	–	–
OpenCOM	×	–	–	–	–	–	–
OSGi	×	–	–	–	–	–	–
KobrA	×	–	–	–	–	–	–
ROBOCOP	×	–	–	–	–	–	–
AUTOSAR	✓	×	–	–	–	–	–
BlueArX	✓	×	–	–	–	–	–
COMDES II	✓	×	–	–	–	–	–
KOALA	✓	×	–	–	–	–	–
Palladio	✓	×	–	–	–	–	–
PECOS	✓	×	–	–	–	–	–
Pin	✓	×	–	–	–	–	–
RUBUS	✓	×	–	–	–	–	–
SaveCCM	✓	×	–	–	–	–	–
Simulink	✓	×	–	–	–	–	–
Scade	✓	×	–	–	–	–	–
UML 2.0	✓	×	–	–	–	–	–
Fractal	✓	✓	×	×	×	×	✓
SOFA 2.0	✓	✓	×	×	×	×	✓
BIP	✓	✓	✓	×	×	×	✓
Reo	✓	✓	✓	×	✓	×	✓
Web Services	✓	✓	✓	✓	×	×	×
ProCom	✓	×[8]	✓	✓	✓	×	×
X-MAN	✓	✓	✓	✓	×	×	✓

[1] Explicit Connector Representation in Architecture
[2] Ability to Define New Connectors
[3] Separate Specification of Computation and Interaction
[4] Explicit Control Flow Modelling
[5] Explicit Data Flow Modelling
[6] Separate Control Flow and Data Flow Modelling
[7] Composite Connectors
[8] ProCom does not allow the definition of new connectors, but its pre-defined connectors can be connected to form more complex coordination structures, which is why it is included in the third iteration of the analysis.

Table 5.2: The Results of Our Survey

thus conform to our requirement. Table 5.3 classifies the surveyed models according to their connector representation. Models in the first column do not conform to the analysed characteristic, and they thus do not proceed to the next iteration of our analysis (see Table 5.2).

No Representation	**Explicit Architectural Representation**	
	Connection	**Architectural Entity**
.NET, Corba CM, CompoNets, EJB, JavaBeans, MS COM, OpenCOM, OSGi, KobrA, ROBOCOP	AUTOSAR, BlueArX, COMDES II, KOALA, Palladio, PECOS, Pin, RUBUS, SaveCCM, Simulink, Scade, UML 2.0	BIP, Fractal, ProCom, Reo, SOFA 2.0, Web Services, X-MAN

Table 5.3: Component Models Categorised By Connector Representation

No Architectural Representation of Connectors

Most component models that do not represent connectors in architecture are object-based (see Section 4.3.2): .NET [118], Corba CM [86], CompoNets [12], EJB [113], JavaBeans [112], MS COM [22], OpenCOM [32] and OSGi [120]. The models that rely solely on object-oriented mechanisms to define components, such as EJB, can only specify provided services; they cannot specify required services and therefore cannot represent composition using links between components' required and provided interface elements. But even if they use some other mechanism for defining component interfaces, such as meta-data in OSGi bundle manifests, they lack any standard architecture description language.

The other two models in this category – KobrA [10] and ROBOCOP [79] – represent components as collections of object-oriented design models. KobrA models components using UML diagrams (class and sequence diagrams). ROBOCOP defines its own design models and focuses on run-time composition of services (objects implementing some pre-defined interfaces).

Explicit Architectural Representation of Connectors

Most component models that represent connectors in architecture explicitly have components that are architectural units or encapsulated components (see

Section 4.3.2). The connector representation in these models can be further differentiated: most approaches represent connectors as links (lines in graphical architectural descriptions) along which messages are sent between component interface elements, whereas some component models embody connectors as architectural entities – often nameable and reusable – with more complex structure (sometimes composed hierarchically) and behaviour (e.g., mediating communication between more than two parties).

The explicit architectural representation of connectors in component models based on architectural units is due to their shared common fundamentals with first-generation architecture description languages, in which connectors were first-class citizens of the same significance as components. However, the role of connectors in most component models in this category has diminished: they have become thin wrappers over underlying primitive composition mechanisms, such as method call or message passing, unable to model more complex interactions. Connectors linking operation-based architectural units (KOALA [91], UML 2.0 [87] and Palladio [16]) represent method calls between provided interfaces and required interfaces of different components. Connectors linking port-based architectural units (BlueArX [65], COMDES II [63], Scade [18], PECOS [48], Pin [58], RUBUS [53] and Simulink [106]) represent data flow (or control flow).

Nevertheless, some component models based on architectural components retain the first-class connector status. ProCom [103] connectors can route control or data flow between several component ports. Fractal [24] allows modelling of complex interaction via so-called binding components. SOFA 2.0 [26] connectors are composed of so-called connector elements, which can model multiple interaction styles (see Section 4.4.2) and complex middleware mechanisms (e.g., remote procedure call with data compression and encryption) [27].

Component models with encapsulated components composed by means of coordination (X-MAN [74] and web services[2] [61]) express connectors, which embody the possibly complex coordination behaviour, explicitly. Likewise, connectors are treated as first-class entities in BIP (Behaviour, Interaction, Priority) [13], where they define the interaction layer of system designs, and in Reo, where they coordinate active components by means of data flow.

[2]We consider web services composed via WS-BPEL orchestration.

5.3.4 Defining New Connectors

Component models suitable for representing interaction patterns need to have an extensible set of connectors in order to define new connectors for new interaction patterns. In this section, we assess how the models with an explicit architectural representation of connectors, which passed the previous iteration of our analysis, conform to this requirement.

The distinction between component models representing connectors as links and as more complex architectural units made in Table 5.3 proves also useful in the analysis of this requirement: none of the surveyed component models representing connectors as links supports definition of new connectors. As mentioned in Section 5.3.3, the links between component interfaces in these models correspond to the underlying composition mechanisms, which are fixed in component model definitions and non-composable. As a result, these models cannot represent interaction patterns as connectors.

The component models representing connectors as more complex architectural units (in the third column of Table 5.3) support defining new connectors. BIP connectors represent sets of interacting component behaviours, and new connectors define new component interactions. Fractal defines new connectors ('composite bindings') by means of so-called binding components. SOFA 2.0 provides a compositional way of building new connectors out of connector elements. In X-MAN, new connectors can be composed from existing control connectors. Reo connectors are composed by so-called nodes that actively move data between connectors. In web services, new WS-BPEL processes can be created to compose web services. ProCom does not define composite connectors as such, but its pre-defined basic control flow and data flow connectors can be connected to form more complex coordination structures[3], albeit the resulting structures do not have the identity of their own and cannot be reused as connectors.

5.3.5 Analysis of the Remaining Criteria

Having ruled out most surveyed component models in the previous two iterations of our analysis, in this section we analyse the conformance of the remaining models to the rest of the requirements from Section 5.2. The results for particular

[3]This is why we considered ProCom in the next iteration of our analysis, although it did not completely pass this iteration.

component models are shown in Table 5.2.

Separate Specification of Interaction and Computation

Component models based on encapsulated components (X-MAN and web services) separate the specification of interaction and computation since they rely on control coordination, which is an exogenous coordination mechanism. Component models with port-based architectural units (ProCom) composed according to the pipe-and-filter interaction style also conform to this requirement. BIP distinguishes between interaction and computation by design: BIP architectures separate behaviour (i.e., computation) and interaction in separate layers. Similarly, Reo connectors encapsulate interaction separately from components.

On the other hand, Fractal and ProCom fail to separate computation and interaction due to their reliance on method call, which is not an exogenous composition mechanism.

Explicit Control Flow and Data Flow Modelling

The method call composition mechanism, used by SOFA 2.0 and Fractal, combines control flow and data flow, precluding their explicit representation. Neither can BIP model control flow and data flow explicitly in architecture. Component models based on control coordination – X-MAN and web services – model control flow explicitly; however, data flow is only implicit in software architecture. On the contrary, Reo as a data-oriented coordination language models data flow explicitly but lacks explicit control flow modelling.

The only model in our survey that models both flows explicitly is ProCom. ProCom distinguishes between two kinds of ports (control and data ports), and also between the corresponding two types of port connections.

Composable Connectors

Only two of the models surveyed in this iteration – ProCom and web services – cannot define composite connectors. ProCom connectors for routing data flow and control flow can be composed together to represent more complex coordination behaviour; however, these compositions cannot be encapsulated in the form of composite connectors. WS-BPEL processes, playing the role of coordination connectors in web service composition, cannot be constructed out of other WS-BPEL processes in a compositional manner.

The rest of the models compose connectors in various ways. BIP connectors, corresponding to interacting component behaviours, can be composed by set union. The Fractal component model supports creation of composite connectors ('composite bindings' [24]), realised by so-called binding components. SOFA 2.0 models connectors as compositions of, possibly hierarchically structured, connector elements. Connector elements represent implementation artefacts (such as stubs, skeletons and interceptors) realising a particular middleware mechanism. SOFA 2.0 connectors thus represent these mechanisms (e.g., remote procedure call, message passing or blackboard); the connectors' composition structure models the infrastructure implementing the mechanisms. X-MAN allows composition of its control connectors to form more complex coordination connectors. Reo composes connectors through nodes.

Separate Control Flow and Data Flow Modelling

None of the surveyed models model control flow and data flow separately, without mutual dependencies imposed by their execution semantics. Firstly, the component models that do not model control flow explicitly – BIP, Fractal and SOFA 2.0 – cannot model these two flows separately. Likewise, because it does not explicitly model control flow, Reo cannot model control flow and data flow separately. The rest of the models in this iteration – ProCom, X-MAN and web services – have control-driven execution semantics, which imposes dependencies on the flows described in Section 4.5.2.

5.3.6 Analysis Conclusions

The main finding of our analysis of existing component models is that none of the surveyed models exhibits all the desired properties for expressing interaction patterns identified in Section 5.2. The majority of surveyed component models – 22 out of 29 – even fail to treat connectors as first-class citizens: they completely lack connectors in architecture (object-based component models) or they only represent connectors as fixed types of connections between component interfaces without the ability to define new, more complex connectors (most component models based on architectural units).

The analysis shows that some composition mechanisms are more suitable for our approach than others. Composition mechanisms based on the request-response interaction style (e.g., method call) are not suitable for expressing interaction patterns, because they preclude the separation of computation and interaction as well as modelling control flow and data flow explicitly. On the other hand, composition mechanisms based on exogenous coordination (e.g., in X-MAN) and the pipe-and-filter (e.g., in ProCom) allow these properties to be achieved.

The two component models that fulfil most of the desired properties are ProCom and X-MAN. ProCom has first-class connectors and models data flow and control flow explicitly, but it does not allow the definition of composite connectors. X-MAN supports composite connectors and models control flow explicitly, but it does not model data flow explicitly. Neither of the two models supports separate (independent) modelling of data flow and control flow since they are both control-driven.

As a result, the absence of an existing component model suitable for our approach motivates us to define a new component model that possesses all of the desirable characteristics. In the next chapter, we define such a component model.

Conceptually, the new component model builds on the component models that fulfil many of the survey's criteria. Like ProCom, it employs port-based component interfaces and data connectors based on the pipe-and-filter paradigm. Like X-MAN, it features hierarchical control connectors coordinating components. However, we define our component model from scratch, not as an extension of some existing component model, in order not to be restricted by the existing model's limitations (mainly related to execution semantics).

Chapter 6

A Component Model with Control Flow and Data Flow Separation

The survey of current component models presented in Section 5.3 found out that there is no existing component model that would be suitable for representing interaction patterns as connectors, with respect to the characteristics identified in Section 5.2. This motivated us to develop a new component model suitable for this purpose [72].

In this chapter, we define the new component model, in terms of its components (Section 6.2) and connectors (Section 6.3), and show the characteristics of system architectures constructed in the model (Section 6.4). We also describe its execution semantics (Section 6.5) and present its formalisation using Coloured Petri Nets (Section 6.6). In this chapter, we only define basic connectors; the definition of composite connectors – the abstraction for representing interaction patterns in our approach – can be found in Chapter 7.

6.1 Component Model Overview

In this section, we explain what decisions we took in designing our component model to conform to the desirable properties for representing interaction patterns from Section 5.2, and we give an overview of the model using a simple example.

In general, the architectures of systems developed in our model comply with the schema in Figure 6.1. A system architecture (denoted by a dotted ellipse) contains the definition of the computation and interaction aspects of system behaviour, which transforms the system's inputs to outputs. Input data can come from data sources external to the system (such as files, databases or sensor readings) or can be generated by the system itself (e.g., random number generators or previously saved state); output data can also be sent to data stores outside of the system or they can be saved as a system state.

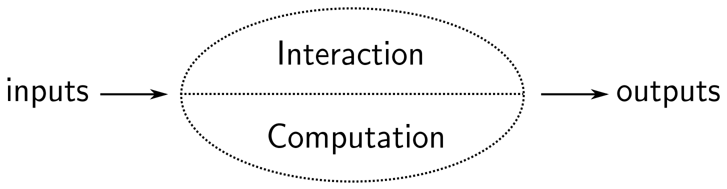


Figure 6.1: Generic Schema of a System Architecture

Our component model separates computation and interaction in software architecture: components define computation only, connectors define interaction only. Components in our model are encapsulated components with port-based interfaces performing a single function that transforms input data to output data. Connectors in our model are explicit in architecture; they transfer data and control between components, thereby coordinating component execution.

To model control flow and data flow explicitly in architecture, component interfaces distinguish between control ports and data ports, and there exist two kinds of connectors: data and control connectors (see Section 6.3). Data connectors are further divided into data channels, unidirectional point-to-point port connections, and data coordinators, which route data dynamically within a group of data channels. Control connectors correspond to the basic control structures of sequencing, branching (selection or conditional execution) and looping.

To enable separate modelling of control flow and data flow, both flows can trigger component execution. To this end, our component model defines the following types of components (see Section 6.2): data-driven, control-driven and control-switched data-driven components. Data-driven components are triggered by the mere presence of their data inputs; control-driven components are triggered by an incoming control signal; control-switched data-driven components are triggered by the presence of their data inputs as long as their state, which is switched by control signals, is 'on' (a so-called enabled state). As a result, systems can comprise of control-driven and data-driven parts. The control-driven and data-driven execution semantics (see Section 6.5) addresses the problem of concurrent execution of these two parts.

Our component model also defines connector composition to be able to create new connectors out of existing ones. Composite connectors are described in Chapter 7.

6.1.1 An Illustrative Example

To illustrate the aforementioned characteristics of our component model, we show the architecture of a simple window controller system in Figure 6.2.[1] The system prevents the window motor in a car from being activated when the window is already in the requested position. System inputs come from a window position sensor and a user-operated button that controls the window movement. System outputs control the motor.

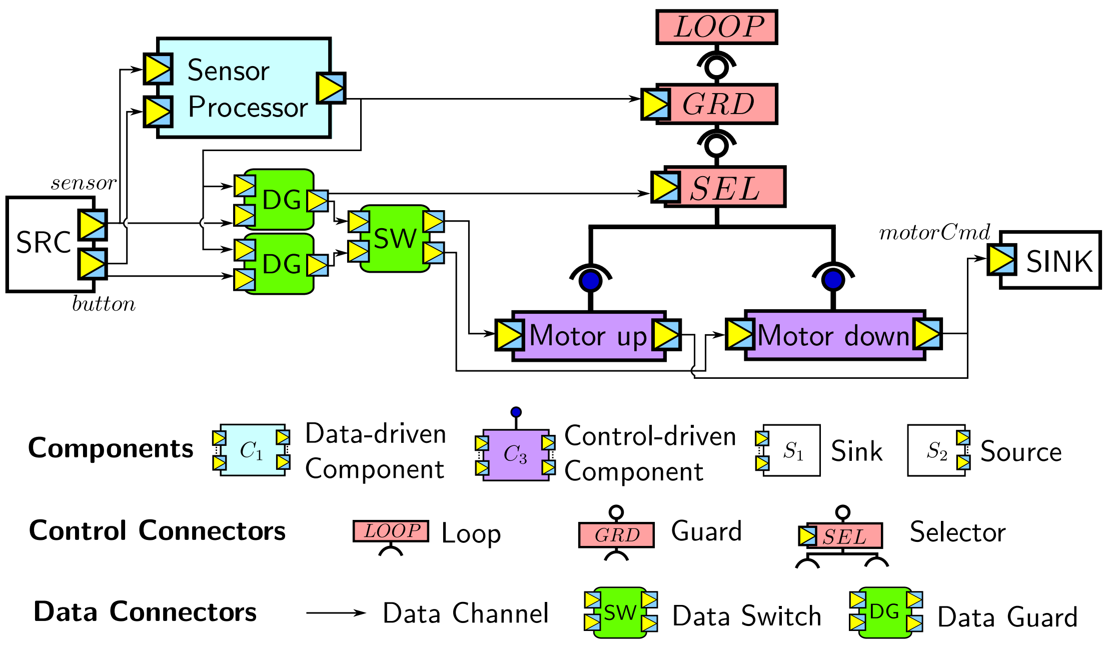


Figure 6.2: Architecture of the Window Controller System

The figure shows a clear separation between computation, defined by components, and interaction, defined by control and data connectors to make both data flow and control flow explicit.

The system's inputs are fed by the SRC source component; the system's outputs are sent to the SINK sink component. The data-driven component decides

[1]The figure serves for illustrative purposes only: component model elements depicted in the figure are defined in detail further in the chapter.

whether the motor needs to be activated. Because it is data-driven, it is active all the time. Motor up and Motor down are control-driven components that compute the system's output to the motor, based on the current window position coming from the sensor, moving the window up or down, respectively. At most one of them is active at a time, depending on the user's preference, represented by the button input.

Control flow is defined by control connectors. They define an infinite loop that can optionally (based on SensorProcessor's output) select (based on the button input) one of the control-driven components to be sent a control signal and thus executed. Data flow between components is defined by data channels, data guards and a data switch. The data guards ensure that data are sent to the selector and control-driven components only if the motor needs to be activated. The switch routes sensor inputs to only one of the two control-driven components, depending on the state of the user-operated button.

As a result, the SensorProcessor component determines whether the system outputs any command to the motor. If it finds that the window should move, its output makes the guard connector send a control signal to one of the control-driven components, which is then executed, computes the command and outputs it via the sink component. Otherwise, control flow returns to the loop connector to start another loop iteration.

6.2 Components

Components in our model perform a single function, transforming input data to output data. They strictly carry out computation only, and they do not coordinate computation in other components. Instead, the execution of their computation is being coordinated by connectors (see Section 6.3).

We have defined three kinds of components that differ in the way their computation is triggered: (i) data-driven components, (ii) control-switched data-driven components and (iii) control-driven components. Additionally, there are two special kinds of components: sources, providing system inputs, and sinks, consuming system outputs. All of the component types are depicted in Figure 6.3 and defined in Section 6.2.3. First, we cover characteristics common to all component types: their interface and computation function.

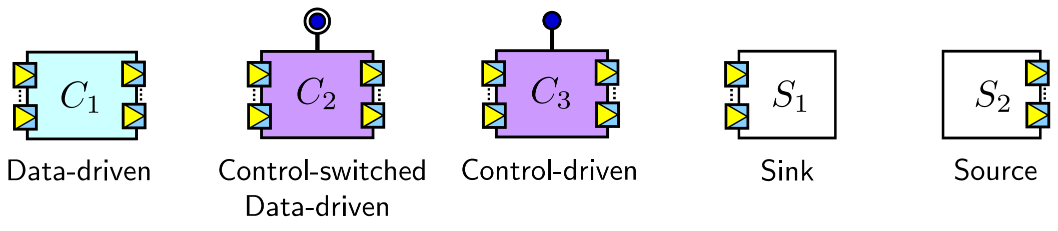


Figure 6.3: Component Types

6.2.1 Component Interface

A component's interface comprises *control ports* and *data ports*. Control ports are places through which control flow interacts with a component. Each component has at most one control port. Control flow enters a component via its control port and, depending on the component's type, either triggers the execution of its computation or changes the component's state, and finally leaves the component. Control ports are not typed since there is only one kind of control flow. In our notation, control ports are depicted as dark sockets on top of components (see Figure 6.3).

Data ports are places of interaction of data flow and a component's computation function. Their semantics corresponds to that of variables in imperative programming languages: they can hold a value and are, unlike control ports, typed. We distinguish two kinds of data ports: input data ports hold the inputs of a component's function and output data ports hold the results of a component's function. Each component can have many input and output data ports. In our notation, data ports are depicted as light triangles within darker squares (see Figure 6.3).

6.2.2 Component Function

Computation carried out by a component is defined by a single function. A component's function f is defined as a function of the values of its input data ports and the component's state. It computes the values of output data ports of the component and possibly changes the component's state. Formally,

$$f : I \times S \to O \times S$$

where I, O, S are the products of the data types (viewed as sets) of a component's input ports, output ports and state variables, respectively.

Figure 6.4 shows an example of a component function written in the domain-specific language for defining functions, implemented in our prototype tool. The function updates the component's state, stored in the integer variable sum, by adding to it the value of the input port input, and outputs the state to the output port output.

```
component Sum {
  in input: int, out output: int
  state sum: int = 0
  {
    sum = sum + input
    output = sum
  }
}
```

Figure 6.4: The Sum Component Function

Components are passive entities, which are executed from outside (by connectors) to perform their computation. The component execution follows the *read-execute-write* schema. A component first reads all the data from its input ports. Then it executes its function, which computes the values to be written to output ports. Finally, it writes the output values to its output data ports.

6.2.3 Component Types

Although every component in our model represents a single function, we distinguish different kinds of components according to the two following criteria:

- whether they exclusively produce or consume data, or whether they transform non-empty inputs to non-empty outputs, and
- what triggers their execution.

The first criterion allows us to distinguish between components whose role is only to feed data into a system (*sources*) or, conversely, only to consume the data output by a system (*sinks*), and components that both consume and produce data by transforming their inputs to their outputs.

The second criterion is related to the separation of control flow and data flow in our component model. Whereas in most component models only one of the two flows is responsible for triggering component execution, which creates the dependence of one flow on the other (see Section 4.5.2), we allow both flows to trigger execution on their own to avoid the above-mentioned inter-dependence. As a result, the component model has component types whose execution is triggered by the presence of their data inputs as well as components triggered by control signals. Table 6.1 summarises all component types; their definition follows below.

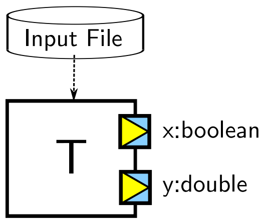
Component type	**Interface (ports)**			**Execution**	
	Control	Input data	Output data	Trigger	Additional condition
Source	0	0	$[1, \infty)$	scheduler	more input data available
Sink	0	$[1, \infty)$	0	data	–
Data-driven	0	$[1, \infty)$	$[1, \infty)$	data	–
Control-switched Data-driven	1	$[1, \infty)$	$[1, \infty)$	data	component is enabled
Control-driven	1	$[0, \infty)$	$[0, \infty)$	control	all input data present

Table 6.1: Summary of Component Types

Sources

Source components provide input data to other components in a system. The data typically come from files or, in the case of embedded systems, sensors. Conceptually, the computation function of a source operates on its inner state, which corresponds to the data that remain to be read, producing its new state and values for output ports. Its interface therefore comprises output data ports only. All of them are filled at once during the write phase of a source's execution. Source components keep providing data until their data source is empty (see Section 6.5 for the details of their execution semantics).

Figure 6.5 shows an example of a source component. T has two output ports, x and y of the data types *boolean* and *double*, respectively. T reads the data from an input file, in which each line contains a value of one of the T's ports together with the port's name. The table on the right shows the values of T's ports (after each execution) and the contents of the input file (before each execution) over three consecutive executions of T (each column in the table corresponds to one execution).



(a) Interface

Input File (before)	x True y 1.1 x False y 0.4 x True y -2.3	x False y 0.4 x True y -2.3	x True y -2.3
Port x (after) Port y (after)	True 1.1	False 0.4	True -2.3

(b) Values of ports and the contents of the file over three consecutive executions

Figure 6.5: Source Component Reading From a File

Sinks

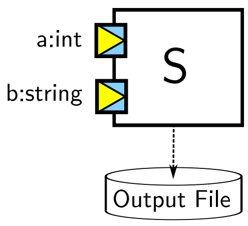
Sink components consume data produced by other components and send them to some external entity outside the scope of a modelled system. The target entity may be a data store, such as a file or a database, or an actuator, in the case of embedded systems. A sink's computation function operates on its input data ports and its inner state, which corresponds to the data that have been consumed by the sink, producing its new state. Its interface therefore comprises input data ports only. All of them are consumed at once during the read phase of a sink's execution. The execution is triggered by the presence of data at all of its input ports.

For an example of a sink component, see Figure 6.6. S has two input ports, a and b of the data types *int* and *string*, respectively. S sends the data it consumes to an output file, where it stores the value of each of its input ports preceded by the port's name on separate lines. The table on the right shows the values of S's input ports (before each execution) and the contents of the output file (after each execution) over three consecutive executions of S[2].

Data-driven Components

indexcomponent!data-driven Data-driven components transform their non-empty data inputs to non-empty data outputs, possibly also changing their state in the process (so-called *transformational* components). Their execution is triggered by the presence of data at all of their input ports, hence data-driven, and it follows the read-execute-write scheme. The interface of a data-driven component consists of input and output data ports.

[2]We assume the file is empty initially.



(a) Interface

Port a (before) Port b (before)	1 "a"	2 "b"	0 "c"

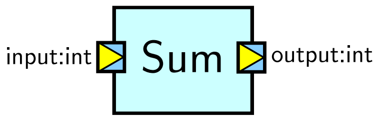
Output File
(after)
	`a 1`
	`b "a"`
		`a 1`
		`b "a"`
		`a 2`
		`b "b"`
			`a 1`
			`b "a"`
			`a 2`
			`b "b"`
			`a 0`
			`b "c"`

(b) Values of ports and the contents of the file over three consecutive executions

Figure 6.6: Sink Component Writing To a File

Data-driven components are suitable for modelling data-processing computations that are active as long as the data are available. For example, the SensorProcessor component in our illustrative example is data-driven for this reason. Possible application domains include digital signal processing in multimedia systems, processing sensor readings in embedded systems or data compression in archiving software.

Figure 6.7 shows a simple data-driven component. Sum has one input and one output port of the *int* data type. It gradually sums all its inputs and outputs partial sums (see Figure 6.4 for its definition). The table in Figure 6.7 shows the values of Sum's input and output ports as well as of its state variable over three consecutive executions (assuming sum= 0 initially).



(a) Interface

Port input (before)	1	2	3
State sum (before)	0	1	3
State sum (after)	1	3	6
Port output (after)	1	3	6

(b) Values of the component's ports and state over three consecutive executions

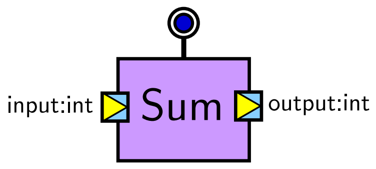
Figure 6.7: The Sum Data-driven Component

Control-switched Data-driven Components

Control-switched data-driven components are transformational components whose execution is triggered by data flow. Additionally, they always find themselves in one of the two execution states – enabled or disabled, which are switched between by incoming control signals. Their interfaces contain data ports and control ports since both control flow and data flow can interact with them.

The computation function of a control-switched data-driven component is executed when all of its inputs are present and the current execution state is enabled; the execution proceeds in the same manner as for a data-driven component. Whenever a control signal arrives at a control port, it switches the component's execution state (from enabled to disabled, and vice versa), and it immediately returns back via the control port.

Components of this type are suitable for modelling data processing computations that can be switched on and off if needed. This is often the case in a class of embedded systems, called systems with modes. Their functionality changes at run-time when their mode changes. In our model, control flow can represent a mode-changing mechanism and several control-switched data-driven components can model functionalities of system modes.



(a) Interface

Control port	–	T	–	T	–
Execution State	E	D	D	E	E
Port input (before)	1	–	–	2	3
State sum (before)	0	1	1	1	3
State sum (after)	1	1	1	3	6
Port output (after)	1	–	–	3	6

T .. triggered, E .. enabled, D .. disabled

(b) Values of the component's ports and state over several executions

Figure 6.8: The Sum Control-switched Data-driven Component

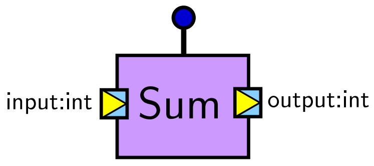
Figure 6.8 shows a control-switched data-driven component that has the same computation function as the component Sum in Figure 6.7 and thus has the same data ports and, additionally, one control port. The table in Figure 6.8b captures several snapshots of the component's port and state values, its execution state (Enabled or Disabled) and whether control flow switched the execution state by triggering the control port (indicated by T). The component computes outputs and changes its internal state in the same way as its data-driven version earlier unless it is disabled.

Control-driven Components

They are also transformational components, but their execution is triggered by control flow, hence control-driven. Because they interact with data flow and control flow, they have ports of both kinds.

A control-driven component's computation function is triggered when a control signal arrives at its control port and all of its inputs are present. The execution then follows the read-execute-write scheme. At the end of the write phase, output data ports are filled with the computed values, and the control signal returns back via the control port. If a control signal arrives at a component's control port while some inputs are still missing, the component cannot be executed; instead, control flows back without triggering any computation.

Control-driven components are suitable for modelling activities that are triggered on-demand, with no action when inputs are missing. They can still be used for data processing where every execution is individually triggered. Since they may have no input data at all, they can be pure data producers (e.g., random number generators and on-demand sensors), like sources. Similarly, they can have no data outputs, like sinks, which may be used to control actuators in embedded systems. Unlike sources and sinks, the execution is not driven by the availability of data, but by control signals.



(a) Interface

Control port	T	T	T	–	T
Port input (before)	1	–	2	3	3
State sum (before)	0	1	1	3	3
State sum (after)	1	1	3	3	6
Port output (after)	1	–	3	–	6

T .. triggered

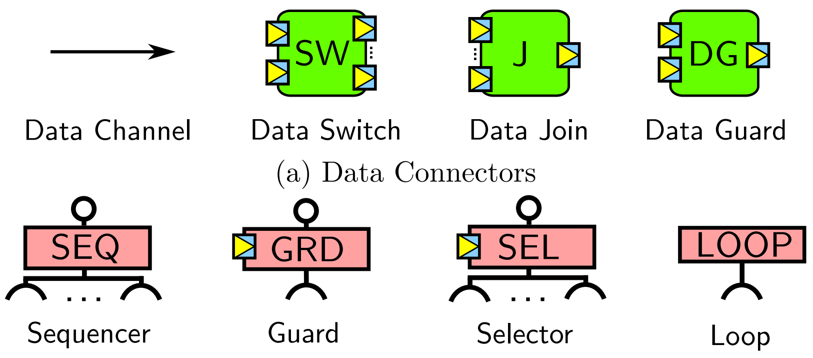
(b) Values of the component's ports and state over several executions

Figure 6.9: The Sum Control-driven Component

Figure 6.9 gives an example of a control-driven component that, again, has the Sum computation function and the same interface as the example control-switched data-driven component. The only difference is a graphical symbol for a control port. The port and state values in the table in Figure 6.9b illustrate that the component function executes only when it is triggered and its inputs are available.

6.3 Connectors

Connectors in our model transfer data and control in a system, thereby coordinating computation by triggering the execution of data-driven and control-driven components. To maximise the separation of computation and interaction, they represent an exogenous coordination composition mechanism, based on the pipe-and-filter interaction style. We distinguish between two kinds of connectors by the flow they carry: data and control connectors. Data connectors are further divided into unidirectional *channels*, connecting an output data port to an input data port, and *data coordinators*, which route data dynamically within a group of channels according to some condition. Control connectors correspond to the basic control structures of sequencing, branching (selection or conditional execution) and looping. Connector types in our model are summarised in Figure 6.10 and defined in the rest of this section.



(a) Data Connectors

(b) Control Connectors

Figure 6.10: Connector Types

In this section, we define connectors on their own. The ways of their composition are discussed in Section 6.4.1. Section 6.5 describes their run-time behaviour; in particular, it addresses the problems of synchronising data flow and control flow that connectors define in system architectures.

6.3.1 Data Channels

A data channel transfers data in one direction between an output data port (source port) and an input data port (sink port). The data ports may belong to a component, data coordinator or control connector. Data channels preserve the type of transferred data, and they strictly require that the types of connected ports are equal, which encourages type-safety and avoids the need for type conversions.

A data channel operates as follows: when its source port contains a value, the channel removes the value from the port, which becomes empty, and writes it to its internal buffer. Whenever the sink port is empty and the channel's internal buffer contains a value, the value is read from the buffer and is written to the sink port.

The prototype tool (see Chapter 9) currently supports the following data channel types: FIFO, FIFO-1 and NDR-1. As shown in Table 6.2, they differ in the capacity of their internal buffer and whether accesses to the buffer destroy the values stored there. The list in Table 6.2 is not fixed since the model can be extended with other types of data channels if necessary. However, due to our model's execution semantics, data channels have to be asynchronous, i.e., they can never block component execution (see Section 11.4.2).

Data Channel	Buffer Capacity	Buffer Writing	Buffer Reading
FIFO	$+\infty$	non-destructive	destructive
FIFO-1	1	destructive	destructive
NDR-1	1	destructive	non-destructive

Table 6.2: Currently Defined Data Channels

Figure 6.11 illustrates the behaviour of the three channel types over the following series of operations on their buffers: two writes (denoted by W) and several reads (denoted by R). For each channel type, the table shows the contents of internal buffers (as sequences of values) and source and sink ports (both before and after an operation). We see that FIFO and FIFO-1 can do only a limited number of reads, due to the destructive nature of their buffer reading operation; in contrast, NDR-1 produces values indefinitely.

6.3.2 Data Coordinators

Data coordinators are data connectors that route data among a set of data channels dynamically, based on some condition. We have defined three kinds of data coordinators: (i) *data switches* for splitting a data flow into several data flows, (ii) *data guards* for filtering values of a data flow, and (iii) *data joins* for joining several data flows into one.

The interface of data coordinators comprises data ports, to which data channels connect. Data coordinators have at least two input data ports and at least one output data port: the values at one input port (also called control data) control the routing behaviour, and the rest of input ports provide the data to be

Before	Source port	1	2	–	–
	Sink port	–	–	–	–
	Buffer	()	(1)	(1, 2)	(2)
	Buffer operation	W	W	R	R
After	Source port	–	–	–	–
	Sink port	–	–	1	2
	Buffer	(1)	(1, 2)	(2)	()

(a) FIFO

Before	Source port	1	2	–
	Sink port	–	–	–
	Buffer	()	(1)	(2)
	Buffer operation	W	W	R
After	Source port	–	–	–
	Sink port	–	–	2
	Buffer	(1)	(2)	()

(b) FIFO-1

Before	Source port	1	2	–	–	
	Sink port	–	–	–	–	
	Buffer	()	(1)	(2)	(2)	
	Buffer operation	W	W	R	R	. . .
After	Source port	–	–	–	–	
	Sink port	–	–	2	2	
	Buffer	(1)	(2)	(2)	(2)	

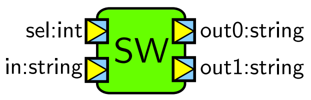
(c) NDR-1

Figure 6.11: Illustration of Data Channel Behaviour

routed out of the coordinator via its output ports. Data coordinators preserve the type of transferred data. Each execution, triggered by the presence of input data, consumes a control datum and an input datum, and produces at most one output datum that is identical to the input datum. The detailed description of the three types of data coordinators follows.

Data Switches

A data switch has two input and at least two output ports. It redistributes input data from an input channel among the group of output channels, according to control data coming from the other input channel. That is, a control datum determines the target channel of an input datum. In our prototype tool, control data are integers whose values correspond to the selected target channels (numbered from 0).



(a) Interface

Port sel (before)	0	0	1
Port in (before)	"a"	"b"	"c"
Port out0 (after)	"a"	"b"	–
Port out1 (after)	–	–	"c"

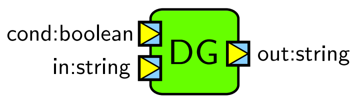
(b) Values of ports over several executions

Figure 6.12: An Example Data Switch

For example, Figure 6.12 shows a data switch redistributing the flow of string values coming from its port in among two output flows via ports out0 and out1, according to the control data from the input port sel. The table illustrates the operation of the data switch over three executions, showing the values of input ports before execution and the values of output ports after. Notice that only one of the output ports is sent a value at a time.

Data Guards

A data guard has two input and one output ports. It filters the input data from an input channel according to the control data (boolean conditions) coming from the other input channel. That is, the condition determines whether the current datum will be routed to the output channel or will be filtered out by the data guard.



(a) Interface

Port cond (before)	true	true	false
Port in (before)	"a"	"b"	"c"
Port out (after)	"a"	"b"	–

(b) Values of ports over several executions

Figure 6.13: An Example Data Guard

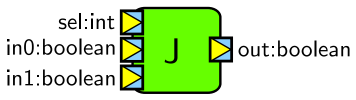
Figure 6.13 shows a data guard filtering the flow of string values coming from its port in to the output port out, according to the conditions from cond. The table illustrates the operation of the data guard over three executions, showing the values of input ports before execution and the values of the output port after.

Data Joins

A data join has one output and $N > 2$ input ports. It builds the output data flow by interleaving its $(N-1)$ input data flows in an order determined by control data coming from the Nth input channel. That is, a control datum determines the source channel providing the next datum to be routed to the output flow. In our prototype tool, control data are integers whose values correspond to the selected source channels (numbered from 0).

For example, Figure 6.14 shows a data join interleaving two flows of boolean values coming to its ports in0 and in1 into the output port out. The table illustrates the operation of the data join over three executions, showing the values of input ports before execution and the values of output ports after.

Data coordinators share many similarities with data-driven components: their



(a) Interface

Port sel (before)	0	0	1
Port in0 (before)	T	T	T
Port in1 (before)	F	F	F
Port out (after)	T	T	F

(b) Values of ports over several executions

Figure 6.14: An Example Data Join

interface comprise data ports, and they are data-driven. However, data coordinators are only data connectors: they forward data, but they cannot transform data. Moreover, there exist subtle differences in execution semantics. For instance, a data join does not have to wait for all its inputs but can execute as long as the control data and the selected input channel contain values.

6.3.3 Control Connectors

Control connectors route control signals, thereby defining control flow in a system and coordinating the execution of control-driven components. Each control connector routes control according to one of the basic control patterns – sequencing, branching, conditional execution and looping. A connector's routing pattern can be both static, such as sequencing and looping, or vary dynamically based on a condition, such as branching and conditional execution.

Control connectors may interact with control flow and data flow. Their interface, therefore, comprises a control port[3], at least one control parameter (corresponding to a required control port) and, optionally, a data port for connectors realising a dynamic control pattern.

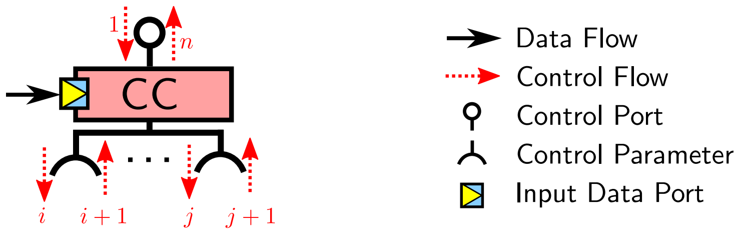


Figure 6.15: Generic Schema of Control Connector Behaviour

Figure 6.15 illustrates the behaviour of a generic control connector. Its execution is triggered by a control signal arriving at its control port. If the connector needs data for its routing decision (e.g., a selector needs a branching condition),

[3]Except for the loop connector.

it waits until the data arrive at its input data port. Once the connector determines its routing pattern (from its data input or statically), it invokes one of its control parameters, by sending the control signal to the parameter and waiting for the signal to return back. This is sequentially repeated until the routing pattern is completed. Finally, the connector returns the control signal back via the control port. Figure 6.15 depicts the sequence of control signal transfers by labelling them: starting with 1 for the initial arrival of the control signal, continuing with some intermediate control transfers ($1 < i, j < n$) and finishing with n for the signal's return. The exact sequence of invocations depends on the control connector's type.

The following control connectors are defined in our model: sequencers, selectors, guards and loops. Table 6.3 gives an overview of their interface and behaviour, which is defined in terms of possible invocation sequences of their control parameters (numbered from 1 to n). Connectors with a static control routing pattern have only one such sequence; whereas connectors with a dynamic routing pattern have several possible invocation sequences, one of which is selected based on the condition coming from their input data port.

Control Connector	**Interface (ports)**			**Invocation Sequence(s)**
	Control Port	Data Port	Control Parameters	
Sequencer	1	0	$[2, +\infty)$	$(1, 2, \ldots, n)$
Selector	1	1	$[2, +\infty)$	$(1), (2), \ldots, (n)$
Guard	1	1	1	$(), (1)$
Loop	0	0	1	$(1, 1, \ldots)$

Table 6.3: Summary of Control Connector Types

Sequencers

A sequencer has a control port and at least two ordered control parameters. It invokes its control parameters in sequence in increasing order.

Selectors

A selector has a control port, an input data port and at least two control parameters. It invokes one of its control parameters identified by an input datum; it thus realises the control branching pattern. In our prototype tool, the input port has to have the integer type, and the selection value must lie in the integer interval $[0, n)$.

Guards

A guard has a control port, an input data port of the boolean type and one control parameter, which is invoked if the condition value coming from the input port equals *true*.

Loops

A loop has an interface consisting of just one control parameter. Therefore, it can only be placed at the top of the connector hierarchy. It is the only control connector that does not have any control port since it does not receive any control signal from its parent control connector; instead, it emits the control signal at the beginning of system execution and then continues to invoke its only control parameter until the system terminates.

6.4 System Architecture

Having defined basic building blocks of our model in previous sections, in this section we discuss how to compose these building blocks to construct system architectures.

We can now refine the generic schema of a system architecture in our model from Figure 6.1 on page 92 into the schema in Figure 6.16. A system reads inputs from data sources, which can be source components or control-driven producer components; system outputs are written to data sinks, which can be sink components or control-driven consumer components. The computation part of the original scheme comprises control-driven components, control-switched data-driven components and data-driven components. The interaction part of the original scheme comprises control connectors, coordinating composed control-driven components and switching the state of composed control-switched data-driven components, as well as data connectors (data channels and data coordinators), exchanging data between components and thereby coordinating the data-driven components

The two partitions in the schema distinguished by two shades of grey correspond to control-driven (darker grey) and data-driven elements (lighter grey) of

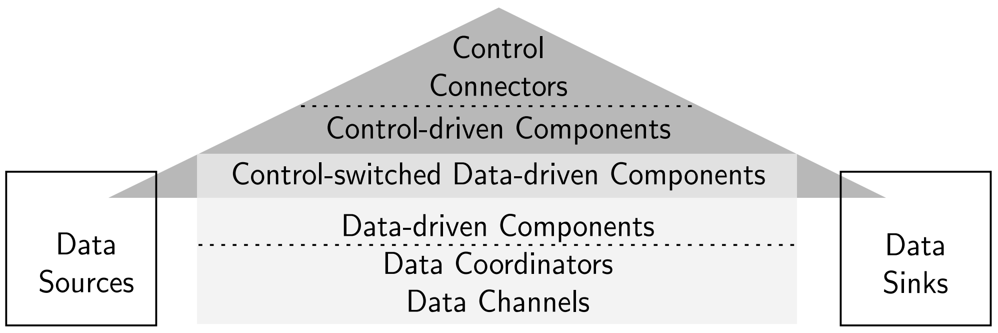


Figure 6.16: Refined Schema of a System Architecture

the model. Such partitioning of an architecture promotes clear separation between the two kinds of system behaviour on the global architectural level. This is advantageous, for example, in reactive control software systems (see Section 3.1.1) because it makes it easier to identify controllers and processing parts in architecture.

In Section 6.4.1, we describe the mechanisms using which a system architecture can be composed from components and connectors. Section 6.4.2 illustrates the breadth of possible system architectures that can be created in our model. Section 6.4.3 summarises the main structural constraints to which valid architectures have to conform.

6.4.1 Composing Model Elements

In our model, connectors establish inter-component interactions since components are self-contained computation functions. Connectors represent an exogenous coordination composition mechanism, modelled in terms of inter-component data flow and control flow. Components are thus composed by means of connectors. Connectors can also be composed together to represent more complex data flow and control flow routing patterns. In general, model elements are composed by connecting their ports.

Table 6.4 summarises for each connector type what model entities it can compose and what kind of interaction it establishes. Data channels compose entities with data ports (components, data coordinators and some control connectors) to establish one-way data flow links; data coordinators compose data channels to route their data flows in a dynamic manner; control connectors compose other control connectors and components with a control port (control-switched data-driven and control-driven components) to establish control flow between them.

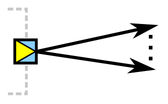
Connector Type	**Composed Entities**	**Established Interaction**
Data channel	Entities with data ports	One-way data link
Data coordinator	Data channels	Dynamic data flow routing pattern
Control connector	Entities with control ports	Control flow pattern

Table 6.4: Connectors as a Means for Composition

We can distinguish between two main kinds of composition in our model: composition via data ports and composition via control ports.

Composition via Data Ports

Data channels and data coordinators compose via data ports. An output data port can be a data channel's source and can be read by the channel once it contains a value. An input data port can be a channel's sink and can be written by the channel once it is empty. During system composition, multiple data channels can share one source or target data port, as illustrated in Figure 6.17. If multiple data channels share a single source port (Figure 6.17a), a value from the port is copied to all of them. Conversely, if multiple data channels share a single sink port (Figure 6.17b), only one of them will be able to write a value to the port. The choice of the writing channel is arbitrary, and thus non-deterministic. Developers can avoid non-determinism by ensuring that only one channel contains data or by choosing the value deterministically, according to some condition, using a data join.


(a) Multiple Outgoing Channels

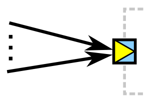
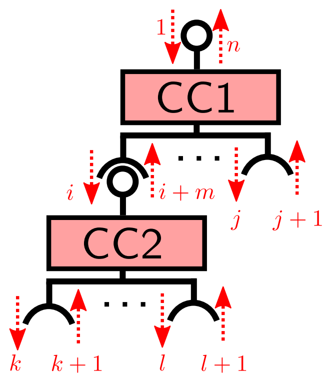

(b) Multiple Incoming Channels

Figure 6.17: Multiple Data Channels Connected to a Single Data Port

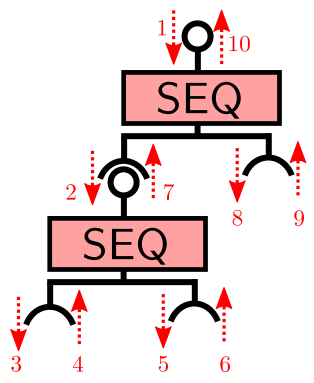
Composition via Control Ports

Control connectors compose via control ports and parameters and, depending of their type, via an input data port. To compose a control connector with another entity with a control port, a control parameter of the control connector needs to be connected to the control port of the other entity. The control connector then becomes the coordinator of the connected entity as it forwards control flow to the entity in accordance with its control routing pattern. If the connected entity is a component, control switches its state (in the case of control-switched components) or triggers its execution (in the case of control-driven components). If the connected entity is a connector, the connection results in composing the control routing patterns of the two connectors.

The resulting control flow is obtained by replacing the part of the flow corresponding to the control parameter of the parent connector with the control flow defined by the child connector. Figure 6.18a shows the composition of two control connectors (the same notation as in Figure 6.15 is used). We see that ith and $(i+1)$th control transfers of CC1 (corresponding to sending control flow to the left-most control parameter and its return in Figure 6.15) have been unified with the first and last (mth) control transfer of the connector CC2, and the arrows have been re-labelled to reflect the fact that the ith transfer is followed by the invocation sequence of the whole CC2. For the scheme to be correct, invocations at different levels must be properly nested ($1 < i, i+m, j, j+1 < n$ and $i < k, k+1, l, l+1 < i+m$), and invocations at the same level cannot overlap ($j+1 < i \vee i+m < j$ and $k+1 < l \vee l+1 < k$). Figure 6.18b illustrates the scheme on an example composition of two sequencers.



(a) Composition of Two Generic Connectors



(b) Composition of Two Sequencers

Figure 6.18: Composing Control Connectors' Flows

6.4.2 Possible System Architectures

To explore the variety of system architectures that can be built in our component model and to demonstrate separate control flow and data flow modelling, in this section we present three categories of system architectures: data-driven systems, control-driven systems and hybrid systems (using both control and data flow coordination).

Data-driven Systems

Figure 6.19 shows an architecture of a data processing system. A typical example is a video processing system; individual data-driven components might correspond to different processing filters that perform signal transformations, such as deinterlacing, noise reduction or edge enhancement. In our model, sources, sinks, data-driven components, data channels and data coordinators can be used to model these systems. Data flow is the only means for interaction modelling and drives the computation.

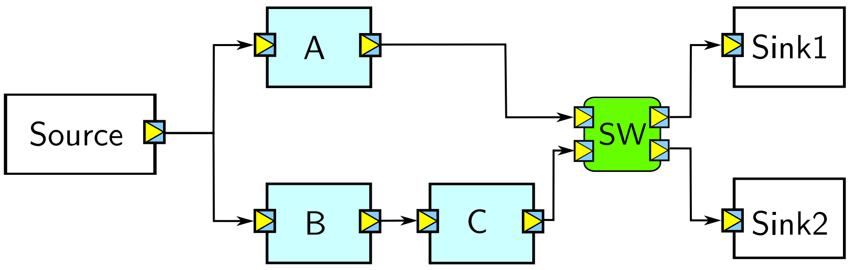


Figure 6.19: Architecture of a Data-driven System

Control-driven Systems

Control-driven systems lie at the opposite end of the software spectrum to data-driven systems: control flow dominates interaction modelling and drives the computation.

An architecture of a simple, control-intensive system is shown in Figure 6.20. Sources, sinks, control-driven components, data and control connectors can be used to model these systems. Control connectors coordinate the composed control-driven components. Because they do not need data inputs to be executed, control-driven components can become pure data producers (such as the component A in Figure 6.20) or even lack data interface at all (C in the figure). As a result, their architecture may lack sources and sinks.

For example, A could be a random number generator. C can only change its internal state during its execution, which may have a manifestation in the physical world, such as activating a machine's actuator.

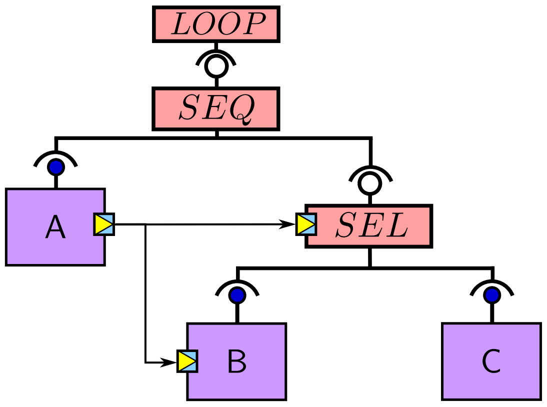


Figure 6.20: Architecture of a Control-driven System

Hybrid Systems with Control and Data Coordination

Finally, all component model entities can be combined to compose a single system, as shown in the illustrative example in Figure 6.2 on page 93. Both control flow (control connectors) and data flow (data channels and data coordinators) model interaction and drive the execution of system components.

6.4.3 Architectural Constraints

System architectures constructed in our model have to conform to certain structural constraints to be valid, i.e., to represent well-defined systems.

In short, an architecture is valid (structurally) as long as data sources and sinks and the main computation and interaction parts are well-defined. All data ports need to be connected to a data channel of the same type and right directionality. Likewise, all control ports and control parameters need to be connected.

The control connector hierarchy has two main constraints, both imposed by the current execution semantics (see Section 6.5): (i) there must be at most one top-most control connector and (ii) the hierarchy must form a directed acyclic graph. The former requirement stems from a limitation of the execution semantics that allows at most one control thread. The latter requirement prevents the infinite control loop in a subset of the control hierarchy and therefore guarantees that one iteration of the control hierarchy finishes in a finite time.

In addition, data flow loops – composed of data-driven components, data coordinators and channels – should contain some conditional exit from the loop (e.g., a data switch) in order for a system with such loops to terminate (see Section 6.5.7).

6.5 Execution Semantics

In this section, we define how systems constructed in our component model behave at run-time. The system behaviour is derived from the behaviour of its constituents and from additional system-level rules that enforce properties that composition itself does not guarantee, such as deterministic system behaviour.

The challenge of defining execution semantics for our component model lies in synchronising execution of control-driven and data-driven component model elements. Additionally, we want systems to execute in a deterministic manner (i.e., a system always produces the same outputs for the same inputs) to simplify their testing.

It is important to realise that execution semantics can be defined in many possible ways, based on different rules one decides to impose on system execution. The execution semantics presented in this section (and implemented in the prototype tool) is suitable for reactive control systems, the domain we chose for validating our approach (see Chapter 10).

In Section 6.5.1, we introduce the main idea of control-driven and data-driven execution using two run-time schedulers. However, the possibility of non-deterministic system execution (illustrated with an example in Section 6.5.2), caused by various design configurations and component models elements (listed in Section 6.5.3), motivates us to synchronise the two run-time schedulers. In particular, we take inspiration from synchronous execution of reactive systems (explained in Section 6.5.4) and introduce the concept of a reactive execution cycle in Section 6.5.5. The synchronisation constraints are formulated in terms of so-called synchronisation rules in Section 6.5.6. Finally, we list some limitations of the presented execution semantics in Section 6.5.7.

6.5.1 Control-driven and Data-driven Execution Semantics

We have designed our component model so that both data and control can independently trigger component execution. As a result, some components are control-driven and some components are data-driven. All component types (and all other component model elements) are passive (see Section 4.5), i.e., they rely on some active entity to trigger their execution at run-time. In control-driven models, it is a control thread (or many of them in concurrent systems); in data-driven models, it is a scheduler.

A control thread traverses control paths defined in a system, triggers component execution and moves data in a system. A data-driven scheduler is essentially an infinite loop moving data in a system, detecting the presence of all inputs of components and consequently triggering their execution.

The basic idea of combining both approaches to system execution in our model is to have systems run by a hybrid scheduler comprising two schedulers – the control thread and the data-driven scheduler – concurrently operating on different model elements. The control thread (i) traverses the control connector hierarchy, (ii) triggers execution of control-driven components, and (iii) switches the execution state of control-switched data-driven components. The data-driven scheduler (i) moves data throughout a system (by reading and writing data channels and running data coordinators), (ii) executes data-driven components, (iii) executes enabled control-switched data-driven components, and (iv) executes sink and source components. Table 6.5 summarises the activities of both schedulers carried out on different component model elements.

Model Element	**Control Thread**	**Data-driven Scheduler**
Control connector	traversing	supplying inputs
Data connector	–	writing & reading
Data-driven component	–	execution & moving data
Source & Sink	–	execution & moving data
Control-driven component	triggering	moving data
Control-switched data-driven component	switching state	execution & moving data

Table 6.5: Activities Carried Out by the Two Schedulers on Model Elements

Although some model elements interact with just one scheduler (such as data-driven components or input-less control connectors), Table 6.5 shows that some model elements (control-driven components, control connectors with data ports and control-switched data-driven components) are operated on both by the control thread and data-driven scheduler. Because the two schedulers run concurrently, there is a potential for race conditions. The next section demonstrates that, without additional synchronisation between the schedulers, systems may indeed behave in a non-deterministic way.

6.5.2 Example of Non-deterministic System Behaviour

In this section, we demonstrate non-deterministic behaviour of systems built from control-driven and data-driven model elements on an example.

Systems in our model can be composed of control-driven and data-driven components. Therefore, they need both schedulers for their execution. As Table 6.5 shows, there exist model entities that both of the schedulers operate on. Because they run concurrently, the order in which they interact with the entities cannot be determined without additional synchronisation rules. Consequently, even if the input data remain the same, different runs of a system may yield different results – a system behaves non-deterministically.

Let us illustrate possible non-deterministic behaviour of the system whose architecture is shown in Figure 6.21. The system operates in one of the two possible modes: it computes the square of its input in mode 0 and the square root of its input in mode 1. The data-processing logic for both system's modes is implemented by control-switched data-driven components SQR and SRT, which are enabled or disabled by control when the corresponding mode is being activated or deactivated, respectively. The data-driven component Chooser computes the next mode and whether the current mode needs changing; the mode-switching logic is realised by control connectors and data coordinators.

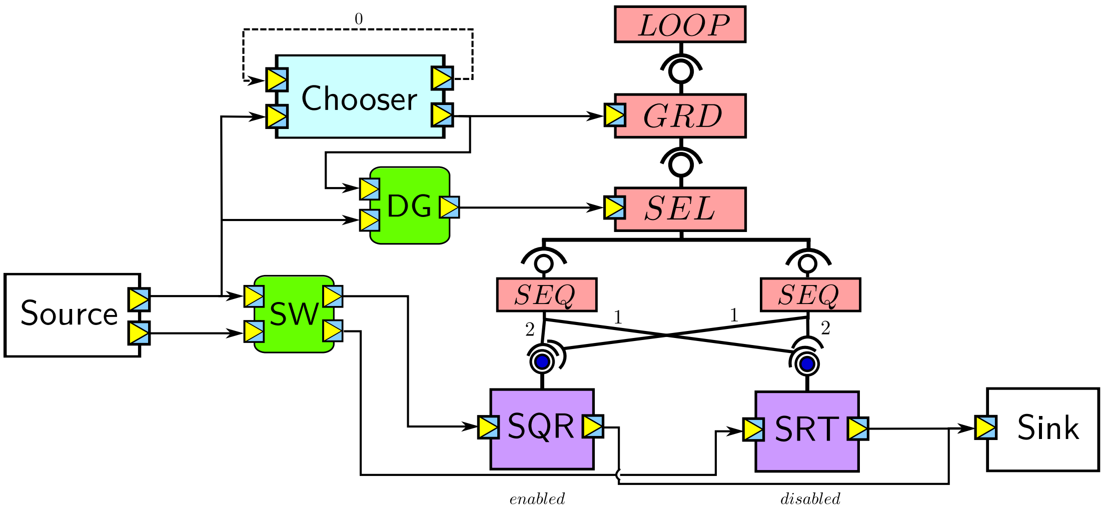


Figure 6.21: System Computing the Square or Square Root of a Number

The input of the system comprises a sequence of pairs of integers. The first integer in a pair determines the component that processes the second integer: 0 denotes SQR and 1 denotes component SRT. For the following input, we would thus expect Output 1:

Input	(0, 1), (1, 2), (1, 3)
Output 1	1^2, $\sqrt{2}$, $\sqrt{3}$
Output 2	$\sqrt{2}$, $\sqrt{3}$

Consider the component SQR. It is initially enabled, and the arrival of the second input pair causes the selector connector to deactivate it and to activate SRT instead (through the two sequencers). However, SQR is also being sent the value 1 from the first input pair for processing. That is, we have several activities – switching SQR's execution state, supplying SQR with data and its consequent execution – performed by different schedulers. Without any synchronisation, there is no guarantee that the data-driven scheduler moves 1 to the SQR's input port and executes SQR, before the selector disables SQR based on the second input pair. If SQR is executed before it is disabled, the system produces Output 1. If, however, SQR is disabled before it processes its input, it will never be enabled again (because the second and third input pairs have 1 as their second element), and the system produces Output 2.

6.5.3 Sources of Non-determinism

Having shown that unsynchronised execution of the control-thread and data-driven scheduler can lead to non-deterministic system behaviour, in this section we identify potential sources of non-determinism in our component model. Non-deterministic behaviour comes from non-deterministic model elements, architectural configurations and race conditions between the two concurrent schedulers.

The following list enumerates these potential sources of non-determinism in our component model:

- components with a non-deterministic computational function (e.g., using random number generators),
  Cause: A developer's choice.

- multiple data channels connected to one input port (see Section 6.4),
  Cause: A developer's choice. Such configurations can be replaced by a data join if determinism is required.

- control-switched data-driven components,
  Cause: A race condition between the data-driven scheduler's activities of

supplying input data and of executing these components, and switching their execution state carried out by the control thread.

- control-driven components, and
  Cause: A race condition between the data-driven scheduler, supplying input data, and the control thread, triggering their execution. If inputs arrive before triggering, a component's behaviour is executed; otherwise, its execution is skipped.

- data channels with destructive write operations (see Table 6.2).
  Cause: A race condition between destructive write and read operations if both operate on the same buffer location (the problem of accessing shared memory for reading and writing simultaneously); it can be avoided by using channels with non-destructive write operations.

Whereas non-deterministic components and the non-determinism related to data channels are design choices of system developers (in other words, they can be avoided by developers unless desired), the non-deterministic behaviour of control-switched data-driven components and of control-driven components follows from the lack of synchronisation between the two schedulers. To avoid such non-determinism and to construct easily testable, deterministic systems, we need to extend the execution semantics by imposing additional synchronisation rules.

6.5.4 Synchronous Execution of Reactive Systems

To synchronise the execution of the control thread and data-driven scheduler, we use the idea of synchronous execution from the domain of reactive systems. In this section, we explain this execution paradigm. Our mechanism for scheduler synchronisation based on this principle is introduced in the next section.

The behaviour of reactive software systems (see Section 3.1.1) can be conceptualised as an infinite sequence of reactions to the environment. In each reaction[4], a system reads inputs from the environment, computes the reaction of the system and outputs the reaction back to the environment (Figure 6.22). This contrasts with the classical, so-called transformational, view of a program as an algorithm that processes a set of given inputs, computes its outputs and finishes by emitting them at the end of its execution.

[4]We use 'reaction' and 'reactive cycle' interchangeably.

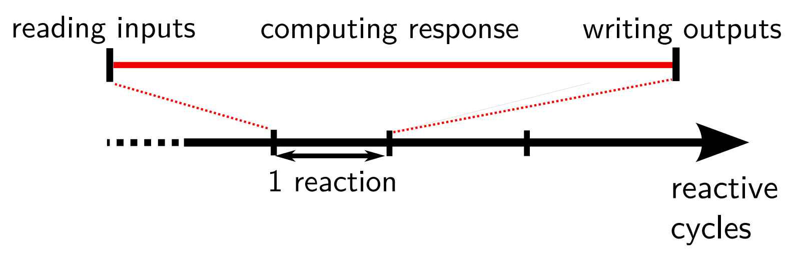


Figure 6.22: Reactive Cycles

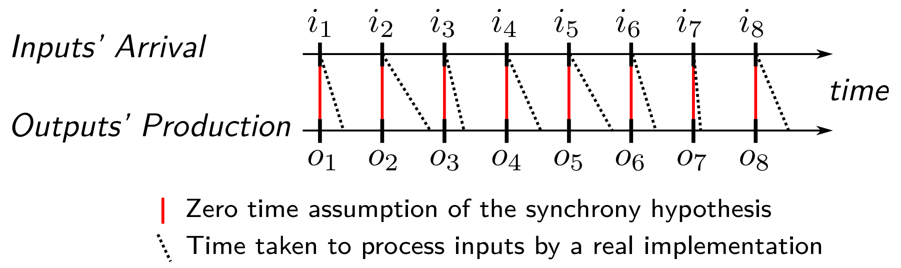

Inputs' Arrival
Outputs' Production
time
i_1 i_2 i_3 i_4 i_5 i_6 i_7 i_8
o_1 o_2 o_3 o_4 o_5 o_6 o_7 o_8
Zero time assumption of the synchrony hypothesis
Time taken to process inputs by a real implementation

Figure 6.23: Feasible Implementation of the Synchronous Paradigm

The synchronous paradigm [17] is an abstraction of reactive system execution that aims to design deterministic and verifiable safety critical systems. It is based on the idealised assumption – the so-called synchrony hypothesis – that each system's reaction takes zero time. It follows that (i) environment remains constant during each reaction and (ii) component execution as well as inter-component data transfer are simultaneous and instantaneous. Such an assumption certainly cannot be achieved in real systems, because any implementation of such systems takes non-zero time to compute. However, if a system implementation guarantees that each reaction of the system can be processed before the inputs of the next reactive cycle arrive from the environment (see Figure 6.23), the synchronous hypothesis is feasible since it holds with respect to the environment. Any design properties based on the synchrony hypothesis therefore still hold. In particular, such designs are easily composable and guaranteed to exhibit deterministic behaviour by construction.

6.5.5 Reactive Execution Cycle

The execution semantics of our component model respects the conceptual view of reactive system execution as a series of reactions, and adopts the synchronous hypothesis to help us enforce deterministic system execution. In this section, we present the basic idea of our approach – a reactive execution cycle.

System execution is defined as a series of execution cycles, each of which corresponds to one reactive cycle. Figure 6.24 illustrates the correspondence. An execution cycle starts by source components emitting values (the system reads inputs), continues by components processing the values (the system computes its response) and finishes by sink components consuming the previously computed results (the system writes its outputs).

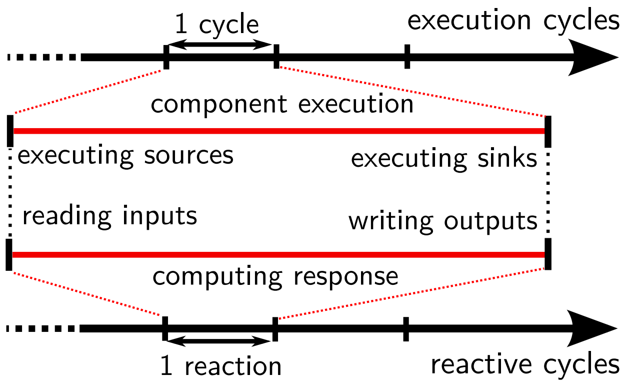
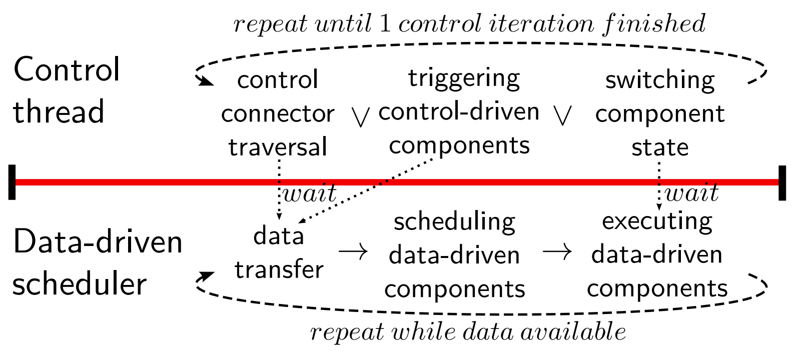


Figure 6.24: Alignment of Reactive and Execution Cycles

Figure 6.25 offers a more detailed description of what happens during an execution cycle. After the source components emit the system's input data, the data-driven scheduler transfers the data from output data ports across data connectors to input data ports; it then schedules the data-driven components with all their inputs present for execution, and it executes them. Component execution leads to new data being produced, which triggers further data transfer and scheduling (see the lower dashed arrow in Figure 6.25).

At the same time, the control thread starts its control loop iteration (see the upper dashed arrow in the figure): traversing the structure of control connectors, triggering the execution of control-driven components and switching the execution states of control-switched data-driven components; it repeats these steps until it completes one traversal of the connector hierarchy. If the loop connector is the top-level control connector, the traversal of the connector hierarchy corresponds to one iteration of the loop, and it is repeated every execution cycle; if another control connector is at the top, the control connector hierarchy is traversed only once in the first execution cycle.

To establish determinism within an execution cycle, we follow the ideas of the synchronous hypothesis. However, we cannot directly use existing realisations of the hypothesis, because they focus exclusively on control-driven languages, such



repeat until 1 control iteration finished
Control thread
control connector traversal
∨
triggering control-driven components
∨
switching component state
wait
wait
Data-driven scheduler
data transfer
→
scheduling data-driven components
→
executing data-driven components
repeat while data available

Figure 6.25: A Reactive Execution Cycle

as Esterel [19], or data-driven languages, such as Lustre [51]. The basic principle is to assume – as in synchronous data flow languages [51] – that data flow and data-driven computation takes zero time, which effectively prioritises data flow over control flow. As a result, the race conditions between the schedulers can always be resolved by giving priority to the data-driven scheduler over the control thread.

This is illustrated in Figure 6.25 by the arrows that denote waiting of the control thread for the data-driven scheduler during control connector traversal (waiting for connectors' data inputs), before a component is triggered (waiting for all inputs to arrive) and before disabling a control-switched data-driven component (waiting for the component to process all available inputs). These rules are described more precisely in the following section.

6.5.6 Synchronisation Rules

In this section, we introduce the rules imposed on system execution by the reactive execution cycle and by prioritising data flow over control flow. These rules help to avoid the undesired non-determinism stemming from the lack of synchronisation between the two schedulers, hence synchronisation rules. We explain the rules and also illustrate with an example how they help achieve deterministic system behaviour.

Apart from conforming to the domain-specific system execution, the execution cycle also serves as a synchronisation point between the control thread and the data-driven scheduler: at the end of a cycle, the control thread has finished one control iteration and the data-driven scheduler processed all data (the initial output of sources and all the outputs of subsequent computations). The execution cycle imposes the following synchronisation rules:

- If the top-most control connector is a loop, the control thread makes one traversal of the control connector hierarchy (one iteration of the loop) per execution cycle; otherwise, the traversal is made once in the first execution cycle. *(SR1)*
- The data-driven scheduler makes source components emit system inputs once at the beginning of each execution cycle. *(SR2)*
- The data-driven scheduler continues to execute data-driven components and transfer data via data connectors, while there exists a data-driven component whose inputs are available or can be supplied via data connectors. *(SR3)*

Although these rules cannot enforce deterministic system execution (situations identified in Section 6.5.3 still cause non-determinism), they simplify the problem: once we establish deterministic behaviour inside an execution cycle, the whole system execution becomes deterministic. To achieve that, we prioritise data flow over control flow in the situations in which the two flows interact. The following two synchronisation rules capture this idea:

- The control thread can skip an execution of a control-driven component during triggering only if at least one of its inputs is not computable in this cycle. *(SR4)*
- The control thread can disable a control-switched data-driven component only if at least one of its inputs is not computable in this cycle. *(SR5)*

To understand the rules, we need to explain the notion of 'an input not computable in this cycle'.

'An input not computable in this cycle'
The assumption of instantaneous data flow and data-driven computation implies that data, once source components emit them, are immediately transferred to their target components and immediately processed by data-driven computations, whose outputs are in turn immediately distributed to other components, etc. As a result, when the control thread reaches a component (control-driven or control-switched), all its inputs that have been transferred or computed by

data-driven computations based on the outputs of source components should be present. Informally, an input not computable in this cycle is a datum that cannot arrive to a component's input port before the control thread reaches the component, assuming instantaneous data flow and data-driven computation.

We can define this notion formally using a deterministic decision procedure. Let us define the binary relation *produces* between all entities with ports in a system (components and data coordinators) and all input ports that are connected via a data connector to an output port of an entity with ports. We define the *production set* of an input port i as a set of all entities with ports that can either directly send a value to the input port i or can send an input to another entity which can transitively send a value to the port i; we compute $PS(i) = \textsc{ProductionSet}(i)$ using the following recursive function[5]:

```
function $\textsc{ProductionSet}(i)$
    $E \leftarrow$ the set of all entities with ports in the system
    $PS \leftarrow \{e \in E | (e, i) \in \mathit{produces}\}$
    for all $x \in PS$ do
        for all $\mathit{inputPort} \in$ input ports of $x$ do
            $PS \leftarrow PS \cup \textsc{ProductionSet}(\mathit{inputPort})$
        end for
    end for
    return $PS$
end function
```

An input for a component port i of a component C is not computable in this cycle at the time when control flow reaches C if i is empty, $PS(i)$ contains no data-driven entities that could be scheduled for execution and no data channel with a target port in $PS(i) \cup C$ can write data from its buffer to the target port. Using this procedure, the race conditions between several concurrent actions during component triggering or switching a component's execution state can be resolved in a deterministic fashion, as described below.

Deterministic Triggering of Control-driven Components

When the control thread reaches a control-driven component that has at least

[5]For simplicity, this version of the procedure does not terminate if there is a data flow loop in the system; however, the code can be modified to prevent this by passing the elements of the production set found so far as an additional parameter and by avoiding recursive calls that would not add new elements to the set.

one empty input port, it must decide whether to wait for the component's inputs and execute the component or to skip the execution. The racing concurrent activities that can lead to non-determinism are: (i) supplying input data to the component by the data-driven scheduler and (ii) triggering of the component by the control thread. To achieve deterministic triggering of the component, the control thread's decision must not be based on the order in which the racing activities occur; instead, it needs to be based on a deterministic decision procedure.

The synchronisation rule for triggering components ensures that the control thread only skips the execution of a control-driven component if at least one of its inputs is not computable in this execution cycle. Effectively, this makes the control thread wait until the production set of each empty input port of the considered component contains no data-driven components that can be scheduled for execution, and all relevant data channels have transferred their data.

Deterministic State Switching of Control-switched Data-driven Components

When the control thread reaches a control-switched data-driven component to switch its execution state, there may be three racing concurrent activities: (i) supplying input data to a control-switched data-driven component, (ii) its execution by the data-driven scheduler, and (iii) switching its state by the control thread; the current execution state of the control-switched component has to be considered as well.

All orderings of the three activities for the two states make up twelve possibilities, some of which are invalid because execution always follows supplying of inputs or because the data-driven scheduler only executes enabled components (see Table 6.6).

The analysis of the valid possibilities shows that the synchronisation rule for control-switched data-driven components ensures a deterministic resolution of the order of the racing activities. If the component is enabled, the switching of the component's state can occur only after processing all inputs, when there are no further inputs computable in this cycle. Again, this forces the control thread to wait until there is at least one empty input port, whose production set contains no data-driven entities that could be scheduled for execution, which effectively means that activities (i) and (ii) take precedence over the activity (iii). If the component is disabled, no additional synchronisation is necessary: after its state

Current State	**Activities' Ordering**	**Behaviour**
enabled	IES	process all inputs computable in this cycle, disable
	ISE	process all inputs computable in this cycle, disable
	ESI	impossible: E follows I
	EIS	impossible: E follows I
	SEI	impossible: E follows I
	SIE	process all inputs computable in this cycle, disable
disabled	IES	impossible: only enabled components are executed
	ISE	enable, start executing
	ESI	impossible: E follows I
	EIS	impossible: E follows I
	SEI	impossible: E follows I
	SIE	enable, start executing

I = supplying inputs; E = execution; S = switching the execution state

Table 6.6: Possible Orderings of Activities Operating on a Control-switched Data-driven Component

is switched to enabled, the component simply starts executing and processing any arriving inputs.

An Example of Applying Synchronisation Rules

To show how the synchronisation rules ensure deterministic system execution, we use the example system from Section 6.5.2, which exhibited non-deterministic behaviour, and we demonstrate its new deterministic behaviour achieved by applying the synchronisation rules.

Table 6.7 describes the execution of the system (see Figure 6.21 on page 116) on the sample input of length three. The execution is split into three execution

Sample Input = ((0, 2), (1, 4), (1, 9))

Execution cycle	Reaction Input	Reaction Output
1	(0, 2)	4
2	(1, 4)	2
3	(1, 9)	3

Table 6.7: System Execution on a Sample Input

cycles, each of which corresponds to one reactive cycle. In every execution cycle, the source component emits two values. The first one goes to Chooser, which computes the current mode (fed to itself by an NDR-1 channel to keep record of the state) and whether the current mode needs changing; it feeds the latter to the data guard, which forwards the new mode to the selector connector, if mode change is needed, and to the data switch, forwarding the second input to the component active in the current mode. The control thread makes one iteration of the loop every cycle; it uses the Chooser's output as a condition for the guard connector to decide whether the control signal should be sent to the selector to switch modes. If the condition holds, the selector disables the component responsible for the old mode and enables the component responsible for the new mode, which has already received data from the data switch. The enabled component's output is sent to the sink.

Notice how the synchronisation rules ensured deterministic execution. In particular, consider the control-switched data-driven component SQR, on which we demonstrated non-deterministic behaviour in Section 6.5.2. In the first cycle, it is initially enabled (see Figure 6.21), and it processes the first input, due to (SR2) and (SR3). In the second execution cycle, the system input (1, 4) makes the selector connector deactivate SQR: there is no input for SQR computable in this cycle, because none of the entities in the production set of the SQR's input port (Source, DS) sends it any data (SR5). There is no race of concurrent activities concerning SQR, because the component does not receive any inputs and therefore cannot be executed. It remains disabled in the third cycle. SQRT is disabled in the first cycle, it becomes enabled in the second cycle when it also processes the reaction's input, due to (SR2) and (SR3), and it remains enabled in the third cycle.

6.5.7 Limitations

In this section, we list the main constraints brought about by the execution semantics:

The cycle of data-driven component execution may not terminate.
For instance, if there is a directed cycle of data connectors connecting data-driven components, the execution of one component in the cycle causes its successor component to be executed, etc. The model contains a means for preventing this problem: e.g., data switches can route data out of such a cycle. Ultimately, the responsibility of avoiding non-terminating execution cycles lies on the system

designer.[6]

The execution is an *infinite* series of execution cycles.

The execution semantics has no means for detecting the end of a system execution. It would not be sufficient to query source components whether they can deliver more data since there exist other data sources (pure control-driven producers) and, what is more, the system can compute without external data – changing its own state (which can be, e.g., reflected via actuators in the system's physical environment).

One control loop iteration has to complete within one execution cycle.

Developers should avoid the control thread waiting for the input data of some control connectors that cannot arrive in the current execution cycle, as it would result in stalled system execution. Such a situation can be easily avoided; it, however, puts an additional constraint on system construction.

Further, a more high level, discussion of the limitations of our model's execution semantics can be found in Section 11.4.2.

6.6 Formal Definition of Execution Semantics Using Coloured Petri Nets

We have formalised the execution semantics of our component model using Coloured Petri Nets (CPN). For any system constructed in our component model, we can derive a CPN model whose behaviour conforms to the execution semantics. The benefits of such a formalisation are:

- a precise definition of the execution semantics,
- the ability to verify some properties of a CPN model, such as whether it behaves deterministically.

We have chosen Coloured Petri Nets for formalising our model's execution semantics for a number of reasons. Firstly, CPN have comprehensible graphical notation, unlike well-known process calculi (e.g., Communicating Sequential

[6]Model checking of formalised system models, such as Coloured Petri Net models described in Section 6.6, may help designers establish this property.

Processes, Calculus of Communicating Systems, π-calculus); additionally, the resulting CPN models can be mapped to the graphical representation of system architectures in our component model. Secondly, Petri Nets have been successfully used to formalise related approaches, component models (X-MAN [70] and Palladio [16]) and workflow languages [1]; our formalisation can be seen as an extension of the work by Lau et al. [70]. Finally, there exist many tools for creating and analysing CPN models (e.g., CPNTools [96]).

In this section, we give an overview of the formalisation. Section 6.6.1 introduces the CPN notation. Section 6.6.2 formulates the basic idea of the formalisation, of composing CPN models corresponding to component model elements. This is further detailed in Sections 6.6.3, 6.6.4 and 6.6.5, which describe the principles of constructing CPN models of component models elements, CPN representation of synchronisation rules and their composition to form CPN models of systems, respectively. In Section 6.6.6, we illustrate that it is possible to formally verify whether the resulting CPN models behave deterministically.

6.6.1 Coloured Petri Nets

In this section, we give a brief summary of the CPN notation and of the concepts necessary for the reader to follow the rest of this section.

Coloured Petri Nets [60] is a visual modelling language for the specification of concurrent systems – their computation, communication and synchronisation.

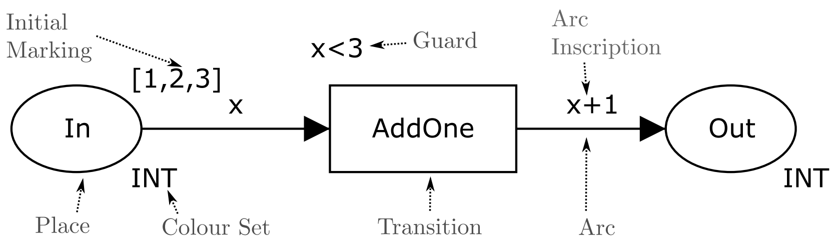


Figure 6.26: Overview of Coloured Petri Nets Notation

A CPN model comprises static net structure – places, transitions, arcs and declarations – and the dynamically changing state of the net, formed by token colours (values) moving between places through transitions. Figure 6.26 depicts a simple CPN model that illustrates how the basic elements connect together: arcs link places and transitions. A place has a name, an initial marking (token colours that appear in the place at the initial state) and a colour set. An arc connects a place to a transition and has an inscription specifying a multi-set of token colours. A transition has a name and, optionally, a condition called guard. Declarations define colour sets (types), functions and variables used in arc inscriptions and transition guards.

A transition represents actions, called binding elements, that change the state of a CPN model by removing token colours from the transition's input places (for which there exists a connected arc coming towards the transition) and by adding token colours to the transition's output places (for which there exists a connected arc going out of the transition). The token colours removed from or added to each place are determined by an inscription on the adjacent arc. A binding element is enabled if there are enough token colours in input places and the transition's guard is *true*. Once enabled, a binding element can occur, which moves the token colours, as described above.

The state of a net evolves in steps consisting of non-empty multi-sets of enabled binding elements that occur simultaneously[7]. Since there are no priorities associated with transitions, any choice of a non-empty subset of non-conflicting enabled binding elements makes a valid step.

For example, in the initial state of the CPN model from Figure 6.26, there are two enabled binding elements associated with the transition AddOne: one binds x to the token colour 1, and another binds x to the token colour 2. Any one or both of them can constitute the next step in the net's evolution. So, it may take one or two steps to transition from the initial state to the state with no enabled binding elements, as shown below:

Place	Token Colours		Place	Token Colours
In	$\{1, 2, 3\}$	$\longrightarrow$	In	$\{3\}$
Out	$\emptyset$		Out	$\{2, 3\}$

Further in the section, we structure our CPN models hierarchically using the constructs called substitution transitions. A substitution transition is a transition that encapsulates an instance of a sub-net, called page. Input and output places of the substitution transition (socket places) are unified with places in the sub-net of the same colour-set (port places) so that the associated socket-port pairs always contain the same marking. The sub-net composition is explained in more detail in Section 6.6.5.

[7]The binding elements must not be in conflict; that is, there must be enough token colours in input places for all of the binding elements.

6.6.2 Basic Idea of the Formalisation

To formalise our component model using CPNs, we have

1. constructed CPN models for all component model elements in a way that allows their further composition (by defining them as reusable CPN subnets), and

2. created a CPN model for enforcing the execution according to the reactive execution cycle and synchronisation rules.

The basic idea of creating a CPN model of a system constructed using our component model is to compose the CPN models corresponding to component model elements according to the structure of composition of the system and to further compose the resulting CPN model with the CPN model enforcing the synchronisation rules.

To model control flow and data flow, we use different colour sets. To model computation, we use transitions' ability to inspect and change token colours. To enforce synchronisation rules, we rely on synchronisation places and global places for detecting the inactivity of data-driven elements.

6.6.3 CPN Representation of Component Model Elements

In this section, we describe the principles of modelling component model elements using CPN. We also show some example CPN models to illustrate our discussion. For simplicity, we do not yet consider synchronisation rules; these are discussed in Section 6.6.4. The interested reader can find more details in Appendix A.1.

Components

CPN models of components express: (i) their interface (data and control ports), (ii) their computation, and (iii) the way their execution is triggered.

For example, Figure 6.27 shows the CPN model of a control-driven component with the input data port x and the output data port y that realises the function $y = 3 * x$.

Each data port is represented as a pair of places: one for data values (whose colour set corresponds to the data type of the port) and one for restricting the port's capacity to one value (uses 'colour-less' tokens of the type UNIT). Directionality of a port is determined by the direction of the connected arcs. For

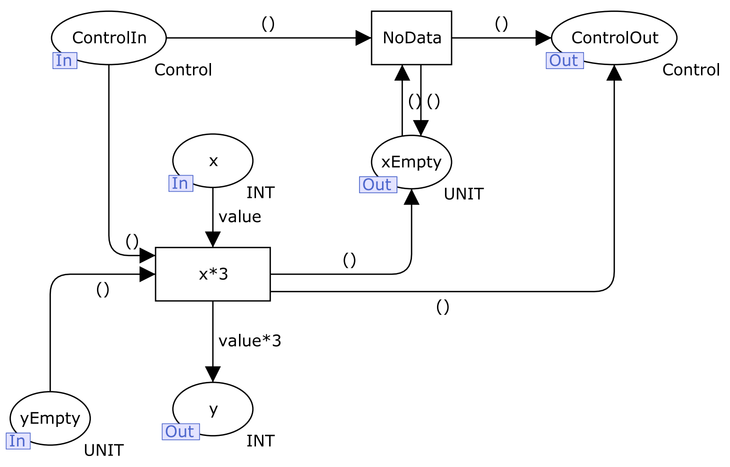


Figure 6.27: CPN Model of a Control-driven Component

example, the output data port y is represented as two places: y and yEmpty in Figure 6.27.

Each control port is also modelled using two places: one for incoming control flow and one for outgoing control flow; both have the token colour Control. There is no need for extra places enforcing the capacity of control places since there is always just one control token in the system (which models one control thread). For instance, see ControlIn and ControlOut representing the control port in Figure 6.27.

Computation – transformation from input data (and state) to output data (and state) – can only be represented as a transition in CPN models. All input and output data ports (i.e., pairs of the places that represent them) are connected to the transition, whose associated data binding element becomes enabled when all inputs arrive and output ports are empty (and thus writeable). When the binding element occurs, the token colours representing input data and the token colours denoting the emptiness of output ports are removed; at the same time, new token colours are added to the output data places and to the places denoting the emptiness of input ports. The computation in our example control-driven component is represented by the x*3 transition connected to the four places corresponding to the ports x and y.

The way a component's execution is triggered depends on its type. The control-driven component in our example is triggered by the arrival of the control token if it also has all data inputs ready. This is modelled by the control input place ControlIn being an input place of the computation transition. When all data inputs and the input control place fill, the transition can occur, which fills output data places, updates the capacity places and also emits a control token to the output control place. To skip computation, the model has an additional transition (in general, there is one such a transition for each data input), NoData, which can occur when control arrives and the input port is empty (i.e., xEmpty has a token).

Data Connectors

CPN models of data channels represent buffers and data transfer between data ports; CPN models of data coordinators represent data flow routing. Their interfaces comprise pairs of places corresponding to the data ports between which they route data.

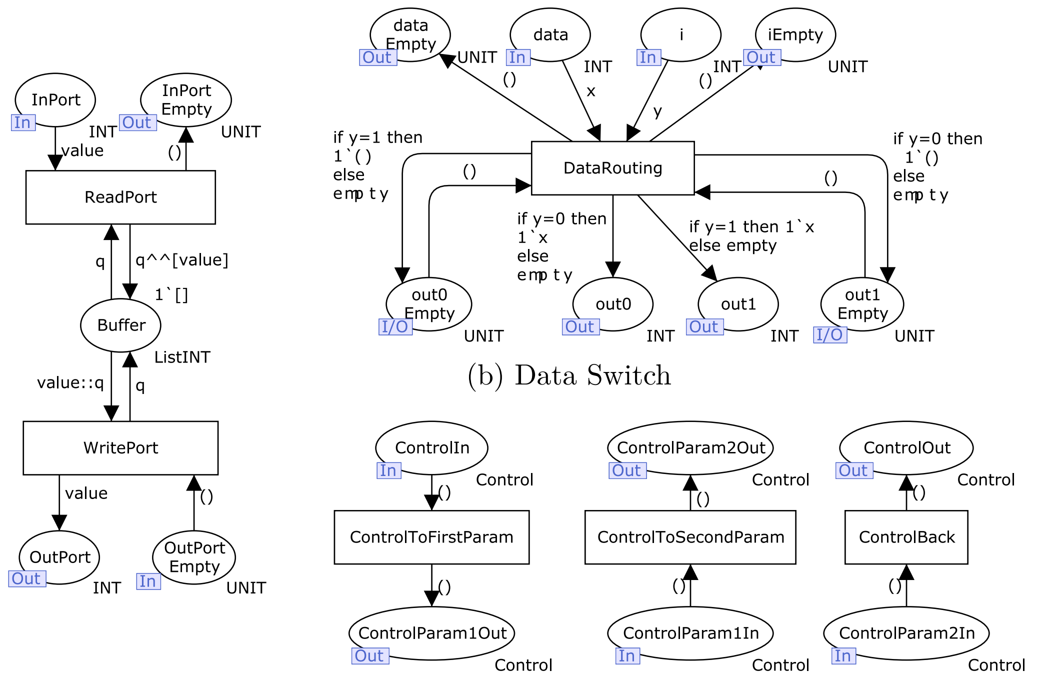
Data channels, such as the unbounded FIFO in Figure 6.28a, are modelled using two pairs of places representing source and target data ports, a place representing the channel's buffer and two transitions – one for reading the source port and another for writing to the target port. Our example FIFO channel transfers data between two integer data ports. The place representing the channel's buffer, Buffer, contains always one token colour which is a list of integers. The ReadPort transition adds the new value at the end of the list; WritePort removes the first value from the list.

Data coordinators have even simpler CPN models: they only comprise pairs of places, corresponding to their input and output ports, and a single transition which routes data between ports according to the inscriptions on its outgoing arcs. Figure 6.28b illustrates this on a model of a data switch.

Control Connectors

CPN models of control connectors represent control flow routing. Their interfaces comprise pairs of places corresponding to control ports, control parameters and, possibly, input data ports.

For example, a binary sequencer (see Figure 6.28c) is modelled by three pairs of places, corresponding to its control port and two control parameters. Its sequencing behaviour is realised by means of three transitions. Note that the model is partitioned into several unconnected parts; the connected graph of control flow of a system is formed during CPN model composition (Section 6.6.5).



(a) Unbounded FIFO

(b) Data Switch

(c) Sequencer

Figure 6.28: CPN Models of Connectors

6.6.4 CPN Representation of Synchronisation Rules

CPN models discussed in Section 6.6.3 lack (for simplicity) synchronisation between the concurrent flows of tokens representing control and data, and they thus behave non-deterministically. To achieve deterministic behaviour of CPN models representing systems constructed in our component model, we have translated the synchronisation rules identified in Section 6.5.6 into CPN. In this section, we explain the basic principles of representing synchronisation rules in CPN. The interested reader can find more details in Appendix A.2.

The resulting formalisation comprises (i) a single synchronisation CPN sub-net, which realises the reactive execution cycle and provides necessary synchronisation mechanisms, and (ii) modifications of the CPN models of individual component model elements presented in Section 6.6.3 to use the provided synchronisation mechanisms. In this section, we give a brief overview of both of these aspects.

Reactive Execution Cycle

The synchronisation CPN sub-net partitions system execution into cycles. Figure 6.29 shows a simplified schema of the part of the CPN sub-net that models this. The execution cycle begins with one token colour in the place CycleStart-

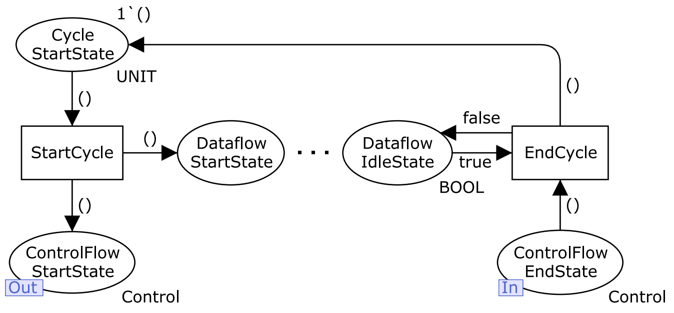


Figure 6.29: Schema of the CPN Synchronisation Sub-net for Execution Cycles

State. The StartCycle transition starts an iteration of the control loop and data flow by placing tokens to ControlFlowStartState and DataFlowStartState, respectively. The control flow iteration is modelled by the control token that leaves the synchronisation sub-net, traverses the control connectors' hierarchy and ends up in ControlFlowEndState. Likewise, the end of data-driven computation and data transfer in a cycle is denoted by the presence of the token colour *true* in DataflowIdleState. At this point, the EndCycle transition moves the token back to

CycleStartState to start a new cycle.

Detecting Idle Data Flow

The middle part of the synchronisation sub-net (omitted in Figure 6.29 for simplicity) deals with data-driven computation and data transfer. It thus represents the data-driven scheduler. The key synchronisation mechanism there is the detection of idle data flow. This corresponds to the system state in which no data-driven component model entity (including data connectors) can execute due to missing data inputs; in other words, all data emitted by source components in that cycle have been processed. The mechanism is used (i) for detecting the end of an execution cycle and (ii) for modelling the synchronisation rules related to prioritising data flow over control flow (see (SR4) and (SR5) in Section 6.5.6).

The basic idea is to keep track for each data-driven entity whether it can be scheduled for execution, i.e, whether it has enough inputs to compute (or transfer data in the case of data connectors). If none of the data-driven entities in a system can be scheduled for execution, the system has reached the data flow idle state.

This is realised in CPN by every data-driven entity having a so-called synchronisation place with a single token colour, indicating whether the entity can be scheduled for execution (*busy*) or not (*idle*). At the beginning of an execution cycle, the synchronisation sub-net distributes *busy* token colours to all data-driven entities. Data-driven entities are responsible for updating the token colours in their synchronisation places from *busy* to *idle*, while the synchronisation sub-net monitors these places and enters the data flow idle state when they are all *idle*. The mechanism allows data-driven entities to acquire the *busy* token colour repeatedly during an execution cycle to execute multiple times. This is represented by the place Restart defined within the synchronisation sub-net but globally accessible from the CPN models of data-driven entities. For completeness, Appendix A.2 shows an example of a complete synchronisation sub-net.

The impact of this solution on the CPN models of component model elements is described below.

Modification of CPN Models of Component Model Elements

As a result of the introduction of the synchronisation sub-net, the CPN models of component model elements from Section 6.6.3 need changing to incorporate the additional synchronisation mechanism.

Data-driven entities need to incorporate the mechanism for the data flow idle state detection. Therefore, their CPN models have an extra synchronisation place, and they also include the Restart place from the synchronisation sub-net. They wait for the *busy* token colour to compute and change it to *idle* after the computation finishes. Additionally, if the execution can result in scheduling another data-driven entity for execution, they set the marking in Restart to $\{true\}$. They have additional transitions to change the token colour from *busy* to *idle* in the case they cannot compute because of missing inputs.

The CPN models of control-driven components and control-switched data-driven components need modifying to realise the synchronisation rules that prioritise data flow over control flow. In particular, they need to wait for any inputs computable in the current cycle before they skip the computation during triggering or switch the execution state to disabled. We have realised this by making them wait for the data flow idle state. This does not exactly implement the decision procedure from Section 6.5.6, because they wait for more data-driven entities than the synchronisation rules require. However, it does not affect correctness, only the efficiency, which does not matter for our purposes of formalisation.

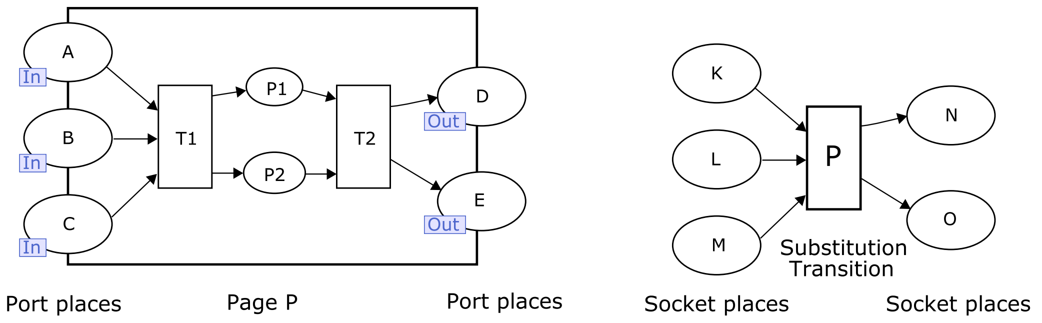
6.6.5 Composition of CPN Models

In this section, we describe the mechanism for composing CPN models of component model elements to form the CPN models of systems. The composition of CPN models is driven by the structure of composition defined by system architecture. To realise composition in CPN, we use their mechanism for hierarchical, modular model construction.

Hierarchical CPN Model Composition

Each CPN model in Section 6.6.3 forms a CPN sub-net (*page*) with an interface comprising of places tagged with In, Out or I/O labels (*port places*). A page can be instantiated in another CPN model as a transition (*substitution transition*); the places connected to the transition (*socket places*) correspond to the page's port places. A socket place and a port place can be linked if they have the same colour set, and the socket place connects to the substitution transition via an arc with the direction corresponding to the type of the port place (towards the transition for an In port place, and away from the transition for an Out port place). The two corresponding places have always the same marking. Figure 6.30

illustrates the terminology: the port places A, B, C, D and E correspond to the socket places K, L, M, N and O, respectively.



(a) Page (b) Substitution Transition

Figure 6.30: Hierarchical CPN Model Composition

Pages are composed by instantiating them as substitution transitions in the same model and by sharing some of the socket places among pages so that one socket place corresponds to several port places in different pages.

Composing CPN Models of a System

Pages modelling our component model elements are composed in the same manner; they only need to respect system architecture to avoid some invalid configurations. For example, two CPN models of components must not be composed directly but via a CPN of a data channel. Appendix A.3 shows a resulting CPN model of the sample system from Figure 6.31.

Additionally, we identified several configurations for which the page composition is not enough: multiple outgoing channels from a single data port and multiple control parameters connected to a single control port. To represent the behaviour required in our component model, the composition requires adding several extra transitions and places in the system model.

Finally, a CPN model of a system is composed with the synchronisation subnet specialised for that system. The specialisation amounts to adjusting the number of synchronisation places to equal the number of data-driven elements in the system.

6.6.6 Verifying Deterministic System Execution

One of the advantages of formalising our execution semantics in CPN is the ability to verify certain system properties by using existing tools. In this section,

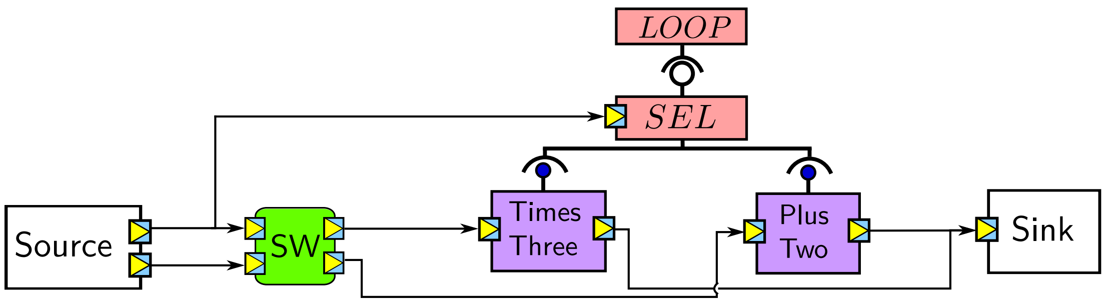


Figure 6.31: Architecture of a Sample System

we illustrate how model checking can verify deterministic behaviour of a system.

We have constructed CPN models of several simple systems and experimented with existing methods for simulation and verification. In particular, we have used CPNTools [96] to perform an exhaustive state space search of several CPN system models to confirm their deterministic behaviour.

For example, see the following excerpt of analysis results. The tool checked the state space of the CPN model of the system from Figure 6.31 with a fixed input of the length two. That is, it completely searched through the state space corresponding to two execution cycles.

```
State Space
        Nodes: 59856     Arcs: 363943     Secs: 825     Status: Full


Home Markings:     Dead Markings:     Dead Transition Instances:
[59856]            [59856]            TRComponentXPlus2'NoData 1
                                      TRComponentXTimes3'NoData 1
```

Apart from the size of the state space and search time, the output shows several markings (states of the net) with special properties. A home marking is a state reachable from every state in the state space; a dead marking is a state with no enabled transitions. In the output, we see one home marking, named 59856, which is at the same time a dead marking. The closer inspection reveals that the marking 59856 corresponds to the final state of the system (that is why it is also dead) with correctly computed outputs. The system behaves deterministically, because there is just one such marking; i.e., the CPN model eventually transitions to the only final state from all its possible states.

Chapter 7

Composite Connectors

Our component model has been designed to fulfil the characteristics suitable for representing interaction patterns. Chapter 6 shows how the model separates the interaction and computation specification, and allows explicit and separate modelling of control flow and data flow in system architecture. In this chapter, we introduce the mechanism for composing connectors in our component model. The products of connector composition – composite connectors – are the component model abstractions that represent interaction patterns in our approach [110].

In Section 7.1, we introduce composite connectors as the means for defining interaction patterns in our model. Section 7.2 defines the structure of composite connectors: their interface, constituent elements and their ways of composition. Section 7.3 defines the run-time behaviour of composite connectors. Section 7.4 concludes the chapter by discussing the representation of stateful interaction patterns and the reusability of composite connectors.

7.1 Overview

In our component model, composite connectors are the abstraction for representing interaction patterns. They define interaction in terms of incoming and outgoing flows of control and data. The interaction defined by a composite connector is determined by the structure of its composition; it is derived from the simpler interactions of its constituent connectors.

Composite connectors have a first-class architectural representation distinct from that of components. Their role in system architecture is to compose components interacting according to the interaction pattern they represent. They have well-defined interfaces, which enables them to be reused in different systems via a repository.

7.1.1 An Illustrative Example

To illustrate the role of composite connectors in our approach, we refactor the architecture of the window controller system depicted in Figure 6.2 on page 93 so that the interaction part of the system will be represented using a composite connector.

The system controls the movement of a car window by issuing commands to the window motor. The SensorProcessor component either prevents the motor from moving, if the window cannot move further in the direction requested by the user, or triggers one of the two control-driven components to move the window in the respective direction.

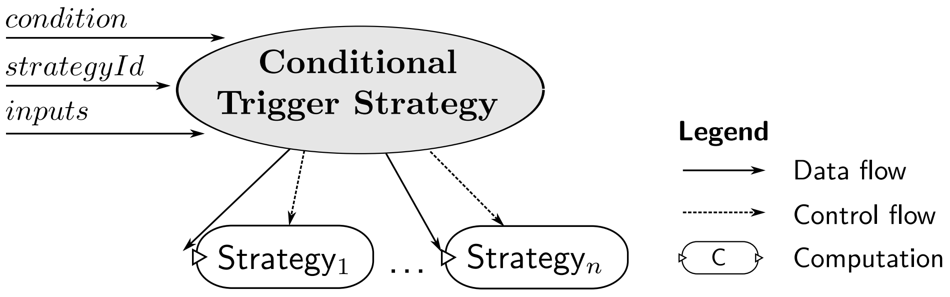


Figure 7.1: The Conditional Trigger Strategy Interaction Pattern

We can generalise the interaction between the three components in Figure 6.2 into an interaction pattern. Let us call it Conditional Trigger Strategy. The pattern (see its schema in Figure 7.1) delegates processing of input data (*inputs* in the figure) to one of a number of available computations (Strategy$_1$, ..., Strategy$_n$) by routing the data to the computation selected by the *strategyId* index and by triggering the computation. The delegation is conditional: it only happens if $condition = true$.

The interaction pattern can be represented as a composite connector in our model and deployed into the window controller architecture instead of the control and data connectors in Figure 6.2. The resulting architecture is depicted in Figure 7.2.

The Conditional Trigger Strategy composite connector encapsulates the interaction defined by the pattern in a single, reusable architectural entity, interfacing

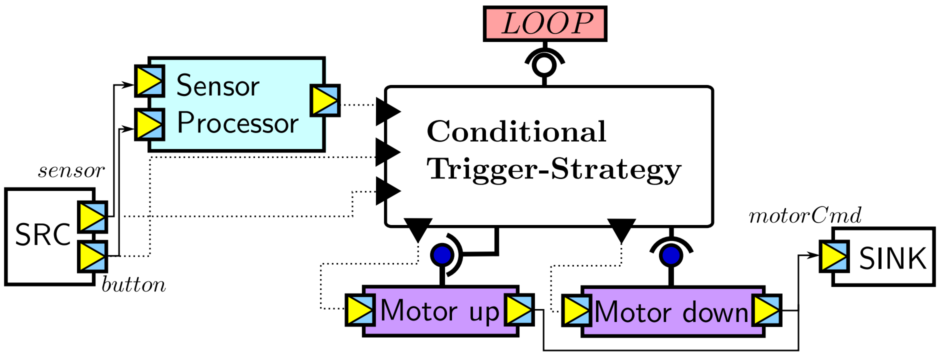


Figure 7.2: Window Controller System with a Composite Connector

participating components. The connected components Motor up and Motor down correspond to the Strategy$_1$ and Strategy$_2$ participants of the interaction pattern. The data inputs of the pattern (*condition*, *strategyId* and *inputs*) come from SRC and SensorProcessor. The behaviour of the system remains unchanged.

The aim of this chapter is to define the interface, possible composition structure and behaviour of composite connectors in our component model.

7.2 Structure

In this section, we describe the structure of composite connectors. We start with their interface (Section 7.2.1) and then discuss how they are composed (Section 7.2.2).

7.2.1 Interface

The interface of a composite connector specifies points of interaction between the connector and entities it coordinates; it is used for composing the composite connector with its coordinatees in a system architecture. The interface comprises (see the schema in Figure 7.3): at most one control port, zero or more control parameters, and zero or more required data ports.

The control port and parameters are the same entities that form part of control connectors' interfaces: a control port is an entry point for the control flow coming from a superior coordinator; control parameters are exit points for the control flow coordinating subordinate entities. Required data ports are dedicated to data flow: required output ports are entry points of data flowing to a composite connector, whereas required input ports are exit points for data flowing out of a

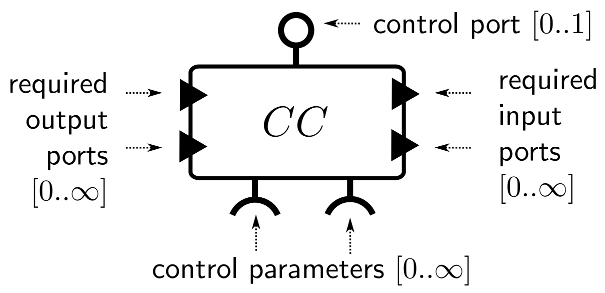
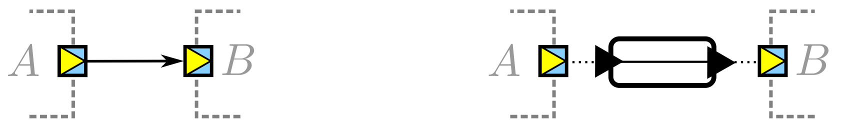

control port [0..1]
required output ports [0..∞]
CC
required input ports [0..∞]
control parameters [0..∞]

Figure 7.3: Schema of a Composite Connector's Interface

composite connector.

Required data ports differ from data ports that form part of the interface of components or data coordinators (hence different graphical notation). Data ports are places that store components' inputs or outputs; required data ports rather correspond to placeholders for data ports, such as ends of data channels. During connector instantiation, they are identified with data ports, but on their own they do not contain any value. For example, a data channel could be modelled using two required data ports, one output and one input (see Figure 7.4b as opposed to the data channel representation, shown in Figure 7.4a, presented in Chapter 6); conversely, the data flow interface of a composite connector is a generalisation of a data channel, with multiple input and output required ports.



(a) A Link between Component Ports (b) A Composite Connector

Figure 7.4: Data Channel Representations

A required data port specifies a constraint on a data port that can be identified with it. The constraint determines (i) the data type and (ii) directionality of any such data port.

7.2.2 Composition

Because we represent interaction patterns in terms of control flow and data flow, we use the basic elements of control flow and data flow from our component model – control and data connectors – to build up composite connectors. At the same time, interaction patterns can be complex; we therefore need a way to compose composite connectors in a hierarchical fashion to represent such complex patterns.

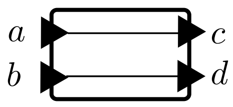
In this section, we describe how we construct composite connectors (i) by composing data connectors, (ii) by composing control connectors, (iii) by composing data and control connectors, and (iv) by composing composite connectors.

Composing Data Connectors

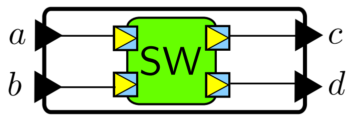
Data channels (see Section 6.3) define a unidirectional data flow between an output data port and an input data port; data coordinators define some dynamic data flow routing behaviour. Composite connectors generalise both: they can define data flow between many data ports, having more inputs and outputs, with complex routing behaviour.

To build more complex data flows, we can compose data connectors in two ways: (i) by aggregating data channels, and (ii) by means of data coordinators.

The first method aggregates data channels so that they become contained inside a composite connector. Each of the aggregated channels connects a required output port to a required input port of their parent connector. The resulting data flow of the composite connector is a union of data flows defined by the channels. For example, see the composite connector created by aggregating two FIFO channels in Figure 7.5a, which defines data flows from the required output ports a and b to the required input ports c and d.



(a) Aggregation



(b) Using a Data Coordinator

Figure 7.5: Composing Data Connectors

The second way involves using a data coordinator, such as a data switch, to compose several data channels, each of which connects a required port of the parent composite connector to a port of the data coordinator. The resulting data flow is a dynamic data routing pattern (defined by the data coordinator) of data flows with particular buffering and access policies (defined by the composed channels). Figure 7.5b shows an example of a composite connector comprising four FIFO channels composed by a data switch; the resulting connector directs the data flow coming through its source port b either to the sink c or d, depending on the values from the source port a.

Composing Control Connectors

Each control connector defines a basic control flow pattern coordinating its control parameters. These control patterns can be composed to form complex coordination hierarchies, as described in Section 6.4. We can use the same composition mechanism to construct composite connectors; we just need to specify their interfaces and the relation to composed control connectors.

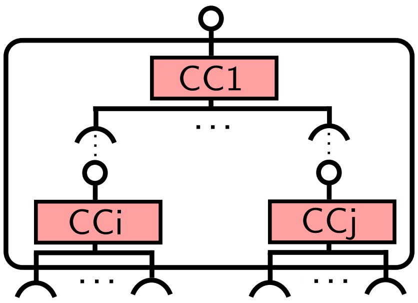


Figure 7.6: Composing Control Connectors

Given a hierarchy of control connectors composed by connecting their control ports and control parameters (see Figure 7.6), we need to delegate an unconnected control port and control parameters to become part of the interface of the composite connector: the control port of the top-most connector in the hierarchy (with no parent) will be delegated to become the control port of the composite connector, as will be the unconnected control parameters of constituent connectors.

Composing Data and Control Connectors

To represent coordination patterns, composite connectors should comprise both control flow and data flow. Hence, they can contain both data connectors and control connectors composed either by aggregating a control connector and a data connector or by connecting a data connector to the data input port of a control connector.

Aggregation is the way of composition in which a control connector and a data connector do not interact. They just happen to be contained in the same composite connector because of their participation in one interaction pattern. They form part of the composite connector's control and data flow. For example, see the FIFO channel, going from *data* to the lower port of the data switch, being aggregated with the selector connector within the Trigger-Strategy connector (Figure 7.7).

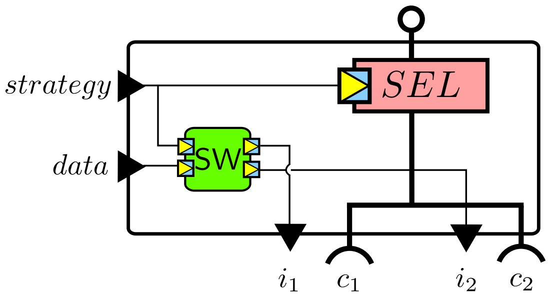


Figure 7.7: The Trigger-Strategy Composite Connector

The latter type of composition makes data flow and control flow interact by a data connector feeding data to a control connector that needs the data for its control routing decisions. In Figure 7.7, we can see a FIFO channel feeding the data from the required output port *strategy* to the data port of the selector connector.

The Trigger-Strategy composite connector (Figure 7.7) realises an interaction pattern that coordinates triggering of two control-driven components, taking the same input. Only one component is triggered via control parameters c_1, c_2 at a time based on *strategy*, after the data switch supplies its input from *data*. The two components represent two different strategies for processing the *data*, hence the name of the connector. Intuitively, we can see that this interaction pattern forms the part of the Conditional Trigger Strategy pattern (introduced in Section 7.1.1); their exact relation can be formulated using the notion of composite connector composition, discussed below.

Composing Composite Connectors

Complex interaction patterns often consist of other simpler patterns that are useful in their own right. To enable the construction of such composite patterns as well as to handle the complexity of connector design, our composite connectors can be composed in a hierarchical fashion.

An existing composite connector can thus be used in the definition of a new composite connector by composing with other entities. The composition can be realised via its control port and parameters or via identifying its required port with a data port of some entity inside the new composite connector[1]. Alternatively, an unconnected interface element of a composite sub-connector can be

[1]The only entities with data ports that can be composed within a composite connector are data coordinators and control connectors with inputs.

propagated to form part of the interface of its parent composite connector.

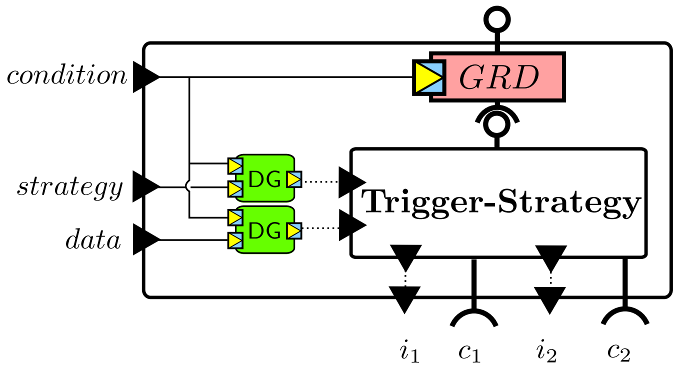


Figure 7.8: The ‘Conditional Trigger-Strategy’ Composite Connector

For example, Figure 7.8 shows the control port of Trigger-Strategy composed with the control parameter of the guard connector and its two required output data ports composed with data guards. The required input data ports of Trigger-Strategy are propagated to become the required data ports of the parent connector, which means that the constraints (type and directionality) on data ports of the inner connector are now enforced by the parent connector. Likewise, control parameters c_1 and c_2 are propagated to the interface of the parent connector. The resulting behaviour of Conditional Trigger-Strategy only triggers components via c_1, c_2 if $\mathit{condition} = \mathit{true}$.

7.3 Execution Semantics

In this section, we define the run-time behaviour of composite connectors to be able to fully define the behaviour of the systems in which they are instantiated.

A composite connector consumes flows of data and control via its input interface elements (required output data ports and a control port) and enforces a particular interaction pattern by producing flows of data and control via its output interface elements (required input data ports and control parameters). To describe the observable behaviour of a composite connector therefore amounts to describing the relationships between input and output flows of the connector's interface elements.

In Section 7.3.1, we describe the run-time behaviour of two example connectors introduced earlier in this chapter, Trigger-Strategy and Conditional Trigger-Strategy. Section 7.3.3 defines the behaviour of composite connectors by their recursive transformation to basic connectors, whose behaviour has been defined in Chapter 6.

7.3.1 Examples

Let us detail the behaviour of the composite connectors Trigger-Strategy and Conditional Trigger-Strategy, whose structure we have seen earlier in this chapter (Figures 7.7 and 7.8).

The behaviour of state-less interaction patterns – patterns for which the relationships between their input and output flows are always the same, irrespective of the internal state of an ongoing interaction – can be described using a table, such as Table 7.1 for Trigger-Strategy. The table captures the dependencies of the control and data flowing through output interface elements on the control and data flowing through input interface elements.

Output Element	**Emits**	**When ...**		
		strategy	*data*	control port
c_1	control	$= 0$		receives control
i_1	data	$= 0$	is present	
c_2	control	$= 1$		receives control
i_2	data	$= 1$	is present	

Table 7.1: Behaviour of Trigger-Strategy

The control parameter c_1 of Trigger-Strategy emits a control signal when the *strategy* required port contains 0 and the control port receives control. Indeed, the selector connector inside the composite connector (see Figure 7.7) only routes control to c_1 if both of the conditions hold. The required input port i_1 produces a datum if there is a datum at the *data* required port and *strategy* contains 0, which makes the internal data switch route the datum from *data* to i_1. The flows output via c_2 and i_2 have similar dependencies on input flows: *strategy* just has to contain 1 instead of 0.

Table 7.2, capturing the behaviour of Conditional Trigger-Strategy, differs from Table 7.1 by having an additional column representing the dependency of all output elements on the added required output port *condition*, which has to be *true* in order for the output elements to emit control or data.

A composite connector's output flow can appear as soon as the input flows on

Output Element	**Emits**	**When ...**			
		condition	*strategy*	*data*	control port
c_1	control	$= true$	$= 0$		receives control
i_1	data	$= true$	$= 0$	is present	
c_2	control	$= true$	$= 1$		receives control
i_2	data	$= true$	$= 1$	is present	
any	nothing	$= false$			

Table 7.2: Behaviour of Conditional Trigger-Strategy

which it depends arrive. For instance, Table 7.1 shows that the output data flows i_1, i_2 of Trigger-Strategy can deliver data inputs to a coordinated control-driven component as soon as the input flows *strategy* and *data* arrive, without waiting for control. That is, there is no additional synchronisation between input and output flows, other than that between the flows that depend on each other.

As a result, composite connectors generally comprise some independent data flow routing and control flow routing parts, i.e., parts which do not interact with each other (composed by aggregation), as exemplified by Trigger-Strategy. The execution semantics prevents any race conditions by prioritising data flow (any data computable in the current execution cycle arrive before control, see Section 6.5).

To achieve the correct behaviour of our example patterns, their users must ensure that input data arrive as many times as control signals within an execution cycle. If control arrives when there is a missing data input that is not computable in the current execution cycle, it waits in the connector indefinitely, which results in stalling the program execution; if there are more data inputs than incoming control signals, the data accumulate in the buffers of data channels, which may lead to the coordinated components processing old inputs (coming from previous execution cycles). Note that this issue is unrelated to the lack of synchronisation of input and output flows; it forms part of the pattern's intended usage and should be properly documented.

7.3.2 Composite Connectors' Run-time Behaviour

Composite connectors are built up from other composite connectors and basic connectors whose behaviour is well-defined (see Section 6.5). To define the behaviour of composite connectors, we describe how we compose the behaviour of their constituents.

Firstly, the behaviour of composite connectors, composed of basic data and control connectors only, is equivalent to the behaviour of the composition of these basic connectors without being encapsulated within a composite connector (see Sections 6.4.1 and 6.5). This is because composite connectors, unlike components, do not impose any additional synchronisation on their input and output flows.

Composite connectors do not impose any additional synchronisation on their input or output control and data flows, because it would be too restrictive. Unlike component execution, which transforms component inputs and can therefore happen only after the inputs arrive, the behaviour of a composite connector may be more complex; different data and control flows comprising the interface of a composite connector can be active at different times to model some interaction patterns, such as a dialogue. In addition, if synchronisation between some flows is required, the component model already contains the means for control flow and data flow synchronisation: not only the afore-mentioned component execution, but also the data flow synchronisation imposed by data coordinators and the synchronisation of control flow and data flow imposed by guards and selectors.

The behaviour of composite connectors composed of other composite connectors can be defined recursively by decomposing the connector composition hierarchies from their leaves up to their roots. To be more precise, we define in the following section a behaviour-defining transformation, called semantic decomposition, from systems containing composite connectors to behaviourally equivalent systems without composite connectors.

7.3.3 Semantic Decomposition

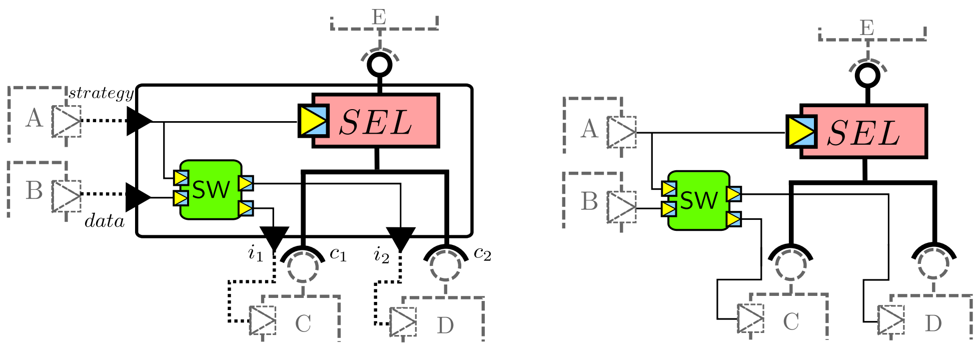
Semantic decomposition is a transformation that eliminates all occurrences of composite connector instances in a given system and replaces them with constituent control and data connectors, thereby defining the behaviour of the original system as equivalent to the behaviour of the resulting system. It is important to realise that the transformation does not, by any means, undermine the first-class status of composite connectors in architecture as its sole purpose is to define behaviour.

It can be defined as a function taking an old system as its parameter and producing the new system with composite connector instances eliminated:

$$semanticDecomposition : System \rightarrow System.$$

The transformation iterates through the component model entities comprising the given system and copies the entities other than composite connector instances to the resulting system.

When it encounters a composite connector composed of basic connectors only, it essentially removes a composite connector's interface elements and directly replaces the connector instance with its constituent sub-connectors. Figure 7.9 illustrates the transformation with an example of a Trigger-Strategy instance. Figure 7.9a shows the instantiated connector together with entities it is connected to (the dashed rectangles A–E). To decompose the composite connector, we remove required data ports by identifying them with the component ports to which they are connected. We also remove control port and parameters from the composite connector's interface and connect their counterparts to the unconnected control ports and parameters inside the composite connector.[2] Figure 7.9b depicts the result of the decomposition for our example connector.



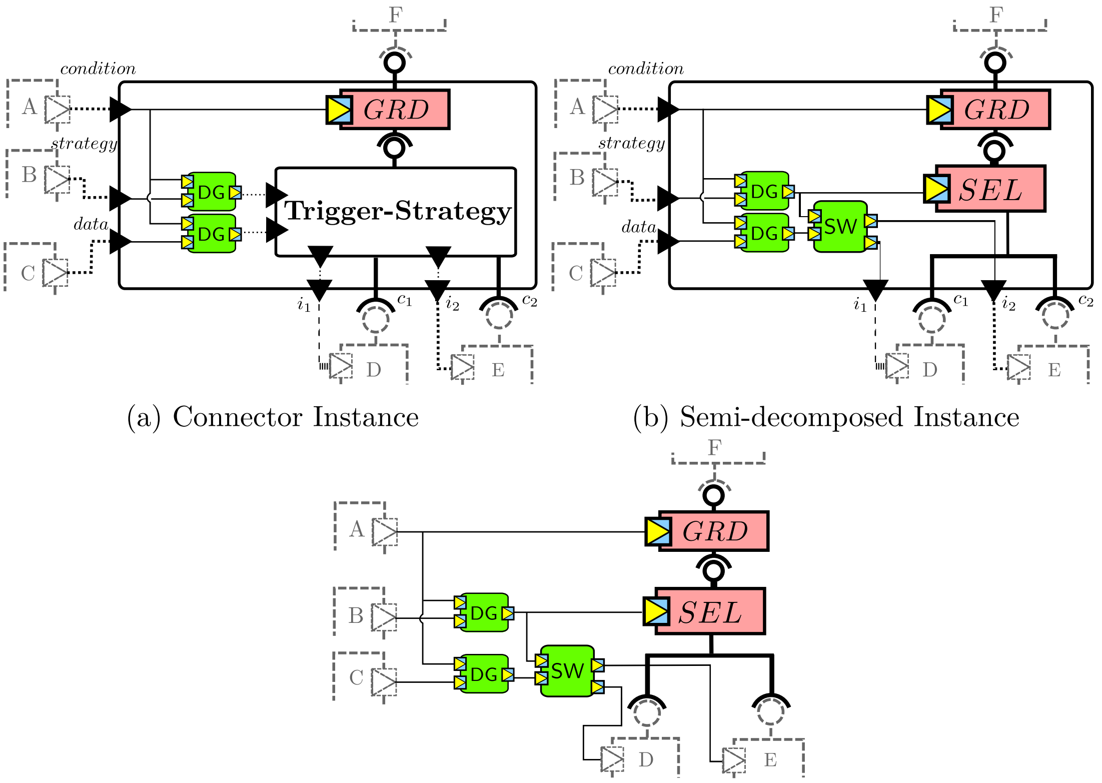
(a) Connector Instance (b) Decomposed Instance

Figure 7.9: Semantic Decomposition of Trigger-Strategy

For a composite connector comprised of other composite connectors, the transformation needs to be applied recursively, starting at the leaves of the connector's composition hierarchy and traversing up towards the top-level instance. Figure 7.10 illustrates this with an instance of Conditional Trigger-Strategy. Figure 7.10a shows the instantiated connector together with entities it is connected

[2]This is trivially achieved in our graphical notation by deleting the composite connector's boundary. In fact, however, it is completely analogous to what happens with required data ports.

to. Figure 7.10b shows the result of semantic decomposition applied to the constituent Trigger-Strategy instance. The second application of semantic decomposition results in a design completely free of composite connectors (Figure 7.10c). Notice that the connector instance in Figure 7.10c corresponds to the connector composition of the Window Controller System's architecture in Figure 6.2 on page 93, because they represent the same interaction.



(a) Connector Instance (b) Semi-decomposed Instance

(c) Decomposed Instance

Figure 7.10: Semantic Decomposition of Conditional Trigger-Strategy

7.4 Discussion

In this section, we discuss composite connectors' capability to model stateful interactions and their reuse potential.

7.4.1 State

In this section, we discuss how a special class of interactions – stateful interactions – can be represented using composite connectors in our component model.

Some interaction patterns exhibit stateful behaviour: the interaction (particular control and data exchanges between interacting entities) changes as the interaction progresses; each such change can be associated with the change of the interaction's state. Conceptually, a state corresponds to a tuple of values of some internal variables that determine the current behaviour of the interaction and whose changes denote the state change of the interaction.

For example, an interaction with several distinct phases in each of which data and control flow differently is stateful. Consider a dialogue between two entities: in every odd phase, the first entity speaks/writes while the second listens/reads, and their roles reverse in every even phase. The state of the dialogue can be represented by a simple boolean variable, whose value flips every time the speaker changes.

Composite connectors in our model do not provide explicit state modelling, which can be found in automata-based behaviour models, such as BIP [13]; nor do they support modelling of the state by means of state variables inside of a composite connector. Each state change would require computation to change the values of the state variables, and that would break the strict separation between computation (carried out by components) and interaction (carried out by connectors) in our component model.

To represent a stateful interaction pattern, one can externalise the state change to be performed by a connected component. The values of state variables then determine control and data flow routing performed by the connector representing the pattern. For example, Figure 7.11 illustrates this method on a composite connector realising a dialogue, a simple request-response interaction pattern between two entities.

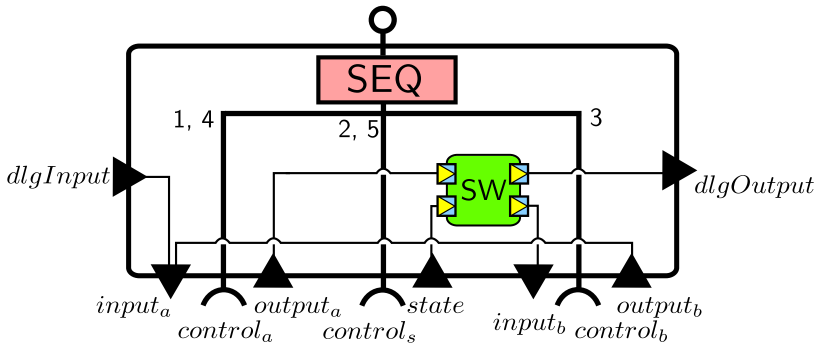


Figure 7.11: The 'ABA' Dialogue Composite Connector

Let us call the two interacting entities A and B. The entities are triggered in the following sequence: A→B→A. Data follow control flow in the same sequence. The interaction has two states that determine whether A's output is routed to B or out of the connector at the end of the interaction. The connector externalises the state keeping and state modification to another component, S, which outputs the current state used by the connector for correct data routing. The interaction starts by supplying the input data coming from *dlgInput* to A via $input_a$ and by triggering A. Then S is triggered to supply the current state, based on which A's output is routed via the data switch to B. B is triggered next and its result is fed back to A (via $output_b$ and $input_a$), which processes it during its next triggering. The second triggering of S yields a new value of the state, which this time routes the A's output out of the connector via *dlgOutput*.

However, state externalisation creates a tight coupling between the state-changing component and the connector. In the above example, any deviation of the component S's behaviour from the expected one (outputting the alternating sequence of 0s and 1s) would break the routing behaviour of the connector. This limitation could be resolved by a suitable state representation within the connector and state-based control and data routing, see Section 11.4.2.

7.4.2 Reusability

Reusability of interaction patterns has been one of the goals of introducing composite connectors to our component model. In this section, we assess the reusability potential of composite connectors and identify directions for its further improvement, which are realised in the following chapter.

The separate specification of interaction (by composite connectors) and computation (by components) allows us to reuse these two aspects independently, which leads to an increased reuse potential for composite connectors compared to entities that mix both aspects in their specification, such as components in component models based on endogenous composition mechanisms (see Section 4.4.4).

Furthermore, composite connectors lack any direct dependencies on coordinated components' types since their interfaces only prescribe incoming and outgoing data flows and control flows. This loose coupling between coordinators and coordinatees[3] helps increase connectors' reusability as it widens the scope of possible contexts in which they can be used.

[3]Nevertheless, there are cases, such as the aforementioned state-changing components, in which coordinated components are expected to exhibit certain behaviour.

However, the interface of composite connectors described in this chapter exhibits unnecessary rigidity; in particular

- required data ports constrain the connected ports to have a specific type,
- required data ports correspond to a single data flow,
- the number of a composite connector's interface elements is fixed.

To understand why the above constraints are undesirable, let us consider the Trigger-Strategy connector described earlier in the chapter (see Figure 7.7). Both of its required output ports, *strategy* and *data*, specify a single data flow of values of a specific type (the integer type in this case). For the data coming from *strategy*, it is indeed necessary, because the selector connector determines to which of the control parameters c_1, c_2 it should forward control according to those data values. On the other hand, no sub-connector interprets the data entering the connector via *data*: the data switch only routes them to i_1 or i_2 without interpreting their values, which makes the specification of the required data type of *data* unnecessary.

Additionally, because one required data port transports a single flow of values, *data* limits the applicability of the connector to components with a single integer input, rendering the connector unusable for components with other input data type or with multiple inputs.

Finally, the Trigger-Strategy connector is limited to coordinating two control-driven components, while the interaction pattern it represents leaves the number of possible coordinated components open. To coordinate more components would require the creation of a new composite connector with a different interface (and with only slightly different composition structure).

The rigidity of composite connectors' design narrows down the number of contexts in which they can be instantiated, thereby lowering their reuse potential. To overcome these issues, we need to treat composite connectors stored in a repository more like templates, parametrisable during their instantiation in particular reuse contexts, to achieve higher variability. This extension to further increase reuse potential of composite connectors is the subject of the next chapter.

Chapter 8

Composite Connector Templates

An abstraction for interaction patterns should be a reusable template, stored in a repository, and instantiated in different contexts each time the interaction pattern it represents is needed. However, composite connectors presented so far rather correspond to pattern instances than to pattern templates.

In this chapter, we define a new form of composite connectors – composite connector templates – designed to represent interaction patterns' solution templates. They are stored in a repository for reuse, and they are instantiated by parametrisation into many different composite connector instances that all conform to the same interaction pattern.

In Section 8.1, we distinguish between connector templates and connector instances by describing their different roles in the connector life cycle. The chapter is focused on connector templates: we describe their interface (Section 8.2), their composition structure (Section 8.3) and their instantiation process (Section 8.4). In Section 8.5, we discuss their impact on connector reuse and some alternatives to the presented parametrisation mechanisms.

8.1 Composite Connector Life Cycle

In this section, we distinguish between composite connector templates and composite connector instances by examining the life cycle of composite connectors.

Since composite connectors in our model are reusable artefacts, their life cycle comprises two distinct phases: one in which they are being developed, an instance of development for reuse, and one in which they are being reused, an instance of development with reuse. In accordance with Lau and Wang [75], we refer to these as *design phase* and *deployment phase*, respectively. The boundary between the phases is formed by the repository (see Figure 8.1), which (i) stores composite connectors deposited at the end of their design phase; and (ii) retrieves the stored connectors for reuse in the deployment phase.

A composite connector's form is different in each of the phases as it caters for different functions and plays a different role. Consequently, we distinguish three conceptually different entities associated with composite connectors: a composite connector template, a composite connector design instance and a composite connector deployed instance. Their role in the composite connector life cycle is depicted in Figure 8.1 and described below.

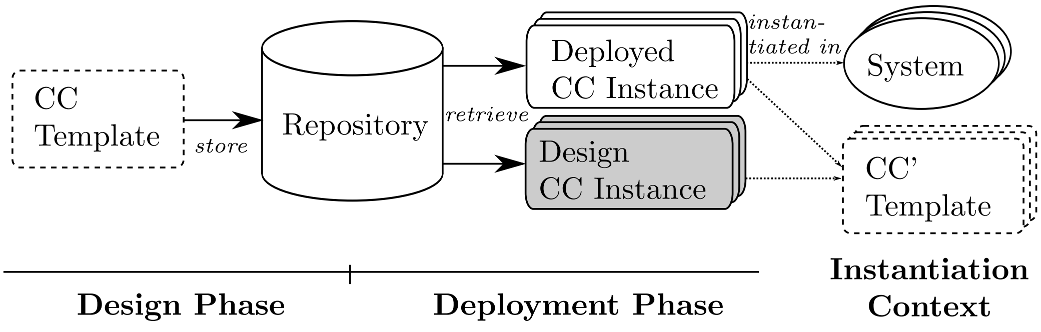


Figure 8.1: Composite Connector Life Cycle

In the design phase, a composite connector is constructed to represent a particular interaction pattern. The product of the design phase is a *connector template*, which is stored in the repository to be reused. A connector template defines the interaction enforced by the pattern, but it also represents different variants prescribed by the pattern's solution template to support different instantiation contexts. That is, the interface and composition structure of a connector template are parametrisable: some of their aspects are expressed as variables whose values need to be fixed in the deployment phase for reuse in a particular context.

In the deployment phase, a connector is retrieved from the repository and takes the form of an instance of the connector template. Since an instance can be used for two different purposes, we distinguish two different kinds of connector instances: deployed instances and design instances.

A deployed connector instance is used to enforce interaction among a particular set of connected components or connectors in the context of its deployment, which can be a system or a connector template under construction. Unlike connector templates, the interface and composition structure of deployed connector instances are fixed since each instance represents only one particular variant of the many variants defined by the template. The instantiation amounts to selecting specific values for the parameters defined by the connector template. For example, Conditional Trigger-Strategy deployed in the Window Controller system in Figure 7.2 on page 141 is a deployed connector instance.

A design connector instance can only be deployed into the context of another connector template that reuses the interaction pattern realised by the instantiated connector. Such instantiation occurs in the design phase of the constructed connector, hence the instance's name. A design instance does not immediately coordinate any components or connectors as a deployed instance; instead, it is a means of composition of the instantiated connector template and the template of the connector under construction. Thus, unlike deployed instances, design instances may keep the parameters defined by their template free (not fixed). Later, in the deployment phase of the connector under construction, the process of fixing a particular variant of its deployed instance will also fix a particular variant of all design instances contained in its template.

8.2 Interface

In this section, we give an overview of what comprises the interface of composite connector templates.

A connector template is a reusable asset, stored in a repository, representing an interaction pattern. It does not interact with other entities in a system; it is the role of its instances to compose other component model entities in order to enforce the interaction pattern represented by the template. The interface of a connector template is used to derive interfaces of its instances.

The interface of connector templates comprises required data ports, a control port, and control parameters to represent input and output flows of control and data; additionally, it may contain multiports and roles. Multiports enable a number of data flows to be merged and transported along a single data path through a composite connector; the number of data flows merged by a multiport can differ among different instances of a connector template. Roles are named groups of other interface elements that can be multiplied a different number of times in the interfaces of different template instances. They allow the interface and composition structure of different instances of the same connector template to vary.

Table 8.1 enumerates possible interface elements of connector templates, their properties and types of those properties. They are described further in this section.

Entity	**Property**	**Property Type**
Required Port	Required type	Data type id
	Required directionality	{input, output}
	SameTypeAs	Required port
Required Multiport	Required directionality	{input, output}
	SourceMultiport	Required multiport
Control Port		
Control Parameter		
Role	Name	Text
	Ports	Set of required ports, multiports and control parameters
	Multiplicity	Integer interval
	Role data	Set of required ports

Table 8.1: Entities Comprising Connector Templates' Interfaces

To illustrate the interface specification of a connector template, Figure 8.2 shows the interface of the connector template representing the Trigger Strategy interaction pattern, whose deployed instance is depicted in Figure 7.7 on page 145. The interface comprises one control port and control parameter, one required output port of the integer type (*strategy*), used for choosing the Strategy component to be triggered and sent inputs, and a pair of required multiports for transporting inputs to the selected component. The multiport d transports the same data flows that are connected to the *data* multiport, which is signified by the SourceMultiport property. Additionally, the template defines the role Strategy, which aggregates interface elements c and d, communicating with Strategy components. The role's multiplicity $[2, +\infty)$ determines that these interface elements should be multiplied in the interface of an instance of this template at least twice. The deployed instance of this connector template in Figure 7.7 on page 145 has the multiplicity of this role set to two.

In the following subsections, we go through different kinds of connector template interface elements in detail.

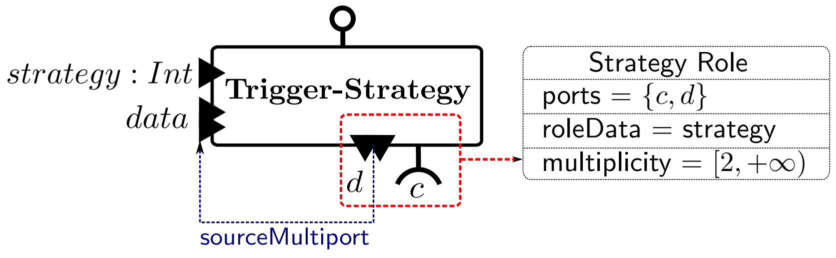


Figure 8.2: Interface of the Trigger-Strategy Connector Template

8.2.1 Required Data Ports

Required data ports in a connector template's interface specify what data can flow in and out of the template's instances since they become part of instances' interface. A required data port specifies the directionality of the data flow (input or output). It also defines the required data type of the flow. This can be either specified directly by naming the data type or indirectly by referring to another required data port that transports the data of the same type (via the SameTypeAs property in Table 8.1).

The different ways of specifying required data types are suitable for different kinds of data flowing through connector instances. We can distinguish between two kinds of data entering a composite connector – decision data and passing data. Decision data are used by a connector to route control or other data through the connector. Ultimately, they are consumed by a control connector with a data input (selector or guard) or a data coordinator, and their value determines the routing decision. Passing data, on the other hand, only flow through a connector, routed in accordance with the connector's semantics and particular values of decision data. Unlike decision data, passing data eventually leave the connector. For example, the data flowing through required output port *strategy* in Trigger Strategy's interface in Figure 8.2 are decision data, because they are used by the connector to choose the Strategy component and they do not leave the connector. The data flowing through the connector's multiports *data* and d are passing data.

A connector instance needs to know the exact types of its decision data since it interprets the data, it does not need so much information about passing data as it only transports them, without interpreting their values. Thus, the data type of decision data should be specified directly by name, whereas the data type of required input and output ports transporting passing data can be specified indirectly by the ports referring to each other via the SameTypeAs property.

The indirect way of data type specification increases the reuse potential of connector templates since the type specification is not fixed in the template; instead, it can vary among the template's instances while still maintaining type safety (see Section 8.4.3).

8.2.2 Required Multiports

Required multiports are connector template interface elements that, like required data ports, specify entry and exit places for data flows in the template's instances. However, they can be used to represent passing data only. They allow a variable number of data flows to be merged and transported along a single data path[1] defined in a connector template. The merged data flows (identified by their source data ports) become template parameters to be fixed during instantiation.

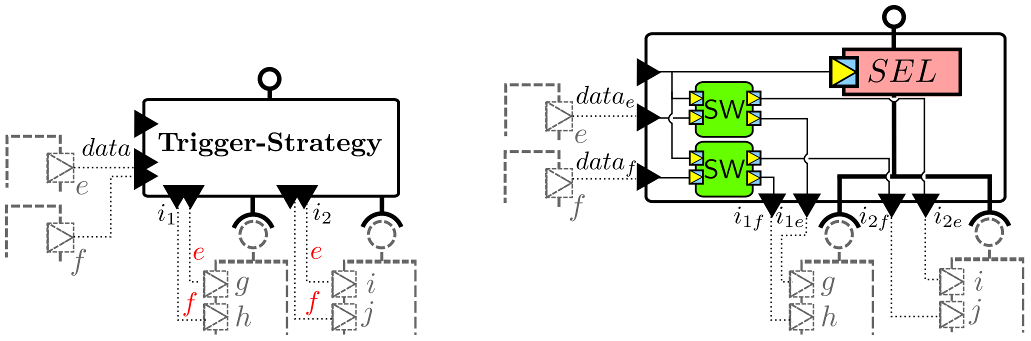
In a connector template, multiports do not prescribe any particular type; instead, required input multiports are associated with the corresponding required output multiports that feed them the passing data using the SourceMultiport association. Figure 8.2 illustrates this with Trigger-Strategy: the required input multiport d is associated with the required output multiport *data*, which signifies that these two multiports transport the same set of data flows in each instance of the template.

In deployed connector instances, multiports can be connected to multiple ports. A required output multiport can be connected to multiple output data ports to merge (multiplex) their data flows; the corresponding required input multiport is connected to the same number of input data ports, splitting (demultiplexing) the merged data flow and additionally directing each of the constituent data flows to a particular input port (selected by the designer).

Figure 8.3a schematically shows the multiport binding of a deployed instance of Trigger-Strategy. The required output multiport *data* merges two data flows coming from output ports e, f, which is denoted by connecting e and f with *data*. The binding of a corresponding (associated via sourceMultiport) required input multiport i_1 not only consists of specification of the target component input ports (connected ports g and h), but it also specifies which data flow should each target port receive: g receives data from e, and h from f.

[1]By a data path, we mean a sequence of data connectors transporting a data flow.

More formally, the binding of a required output multiport m_o to several output data ports $o_1, ..., o_k$ corresponds to a set of pairs $\{(m_o, o_1), ..., (m_o, o_k)\}$; and the binding of a required input multiport m_i being fed data from m_o (i.e. $(m_i, m_o) \in sourceMultiport$) to input ports $i_1, ..., i_k$ corresponds to a set $\{(m_i, i_1, o_{d(1)}), \ldots, (m_i, i_k, o_{d(k)})\}$, where $d : \{1, ..., k\} \rightarrow \{1, ..., k\}$ is a permutation of indices representing the designer's selection of which data flow (identified by the index of its source output port) should be routed to which target input port. The d mapping has the following semantics: $d(a) = b$ means that the data flow originating in o_a should be routed by the connector to the input port i_b.



(a) Deployed Instance with Multiports (b) Equivalent Instance without Multiports

Figure 8.3: Multiports in Trigger-Strategy Instances

It should be noted that we have not defined multiports as part of deployed composite connector instances in Chapter 7[2]: Figure 8.3a thus shows an invalid composite connector instance. However, we can map composite connectors with multiports into a subset of our component model for which the behaviour is already defined. Such a semantics-defining mapping (similar to the semantic decomposition of composite connectors in Section 7.3) takes a connector template and multiport binding information associated with a connector instance, and produces a composition of data and control connectors that defines the behaviour of that particular instance.

Figure 8.3b shows the result of the mapping of the Trigger-Strategy instance in Figure 8.3a to a connector definition without multiports with equivalent behaviour. Multiports have been decomposed into several required ports according to the number of connected ports (two in this case); each required port is responsible for transporting only one of the data flows multiplexed by its corresponding

[2]We have done so for simplicity; in our prototype implementation, multiports are valid interface elements of deployed connector instances.

multiport. Moreover, the data paths inside the connector template have been duplicated as well so that they now transport only streams of values of basic data types.

8.2.3 Control Ports and Parameters

Control ports and parameters in a connector template's interface specify how control flows in and out of the template's instances since they become part of instances' interface. There is at most one control port and zero or more control parameters per composite connector template. Unlike data flow, control flow is untyped in our model, which diminishes the opportunities for parametrisation. As a result, no additional properties of these interface elements can be specified, and they retain their semantics described in Section 7.2.1.

8.2.4 Roles

The mechanism of roles parametrises certain structural aspects of connector templates; it enables creating a variety of instances of the same template with different interfaces and internal composition structure, which increases the reuse potential of composite connectors in our model. The mechanism is inspired by the roles in pattern solution templates (hence the name), which specify minimal expected behaviour of entities that can participate in a pattern (see Section 3.3).

In the context of composite connector templates, we define a role as a group of required data ports and control parameters. A role forms an interface between instances of the connector and the role's participants; that is, a role expresses the constraint on participants by defining interface elements through which participants interact with the connector. Crucially, a variable number of participants can be assigned to play a given role during a connector's instantiation, which allows us to have composite connectors with variable interfaces.

Additionally, a role has a name and multiplicity, an integer interval specifying how many times the role can be instantiated (multiplied) within a connector instance. Every participant needs to be distinguished so that the connector can route control or data to it, based on some values of the connector's decision data. Therefore, each role also specifies which required output ports (corresponding to incoming decision data) identify participants of that role (see Table 8.1).

To instantiate a connector template, the developer needs to specify participants for each role contained in the connector template. A participant information comprises the role that the participant plays, and the decision data values identifying the participant. The interface of a connector instance contains as many copies of required ports and control parameters as there are participants of the role that aggregates them.

For example, the interaction pattern realised by Trigger-Strategy allows choosing between possibly many strategies. Each strategy corresponds to a control-driven component that receives input data and is triggered by the connector. To express the above constraint on participants in the interface of Trigger-Strategy, the Strategy role (see Figure 8.2) aggregates part of the connector's interface dedicated for communication with strategy components – the required input port d and control parameter c. Its multiplicity determines that each connector instance should have at least two participants, each of which is identified by a value of the integer type (*strategy*'s required type).

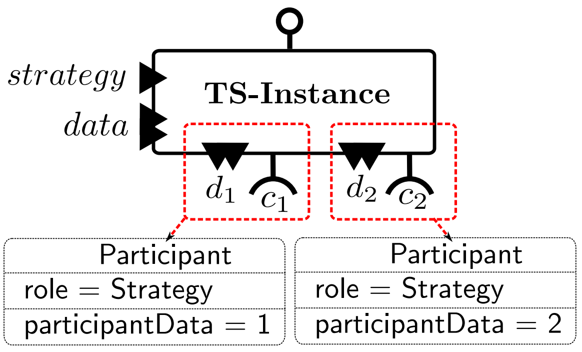


Figure 8.4: A Trigger-Strategy Instance's Interface

Figure 8.4 shows the interface of an instance of the template from Figure 8.2. During instantiation, two participants of the Strategy role have been specified: one to be selected when $strategy = 1$ and another one to be selected when $strategy = 2$. As a result, the ports aggregated by the Strategy role have been duplicated in the connector's interface for each of the role's participant.

8.3 Composition Structure

A connector template comprises: (i) an interface that defines how its instances interact with other model entities (see Section 8.2), and (ii) composition of control and data connectors that realise a particular interaction pattern. Unlike the composition structure of deployed instances described in Section 7.2.2, connector templates parametrise their composition structure so that it can be instantiated differently in different deployment contexts to support variant behaviours of connector instances.

To parametrise the composition structure of connector templates, we use the mechanism of roles (see Section 8.2.4). When a connector template is instantiated, the specification of participants for each role not only guides the instance's interface creation but also the transformation of the template's composition structure into the composition structure of the instance.

A connector template is composed of variable and fixed constituent elements, according to whether they are affected by the instantiation transformation. Fixed elements are not associated with any role; variable elements must be associated with some role defined in the template's interface. There exist two ways in which an element can vary during instantiation: either it can be duplicated multiple times or its 'arity' (the number of ports in its interface and its internal structure) changes. A connector template comprises design connector instances for the parts that change their arity during instantiation and deployed connector instances for the parts whose structure does not change (whether they are multiplied during instantiation or not).

All basic connectors (Section 6.3) and composite connectors (Chapter 7) described in earlier chapters correspond to deployed connector instances and can be composed within connector templates in the way presented in Section 7.2.2. In Section 8.3.1, we extend the concept of design connector instances to data coordinators and control connectors (not only composite connectors as described in Section 8.1), and we discuss how to compose connector templates from both kinds of connector instances. In Section 8.3.2, we explain how to compose connector templates from existing connector templates. Finally, Section 8.3.3 lists some structural constraints on the design of connector templates, given by the instantiation mechanism.

8.3.1 Design Instances of Basic Connectors

Design instances of data coordinators and control connectors possess most of the characteristics of composite connector design instances (see Section 8.1): (i) they are instantiated in a connector template; (ii) they may keep some parameters unassigned and propagate them as parameters of the template they are instantiated in; and (iii) all their unassigned parameters are fixed during instantiation of the template that contains them (i.e., they effectively become deployed instances). Unlike design composite connector instances, their parameters are not defined in their template, since there is none, but they have only one such parameter, defined directly in the component model: arity.

Arity denotes the number of specific interface elements (ports or control parameters) of a connector (see Table 8.2 for details). If the arity of a design instance is left unassigned in the definition of a connector template, the design instance can vary among the template's instances, which helps to realise restructuring part of the instantiation transformation.

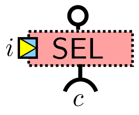
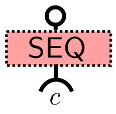
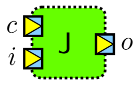
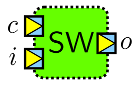
Graphical Notation	**Connector Type**	**Parametrised by**
	Selector	Number of control parameters (c) & decision data values associated with each of them
	Sequencer	Number of control parameters (c) & their sequencing indices
	Data Join	Number of input data ports (i) & decision data values associated with each of them
	Data Switch	Number of output data ports (o) & decision data values associated with each of them

Table 8.2: Summary of Design Instances of Basic Connectors

Table 8.2 gives an overview of design instances of basic connectors. Design instances of selector and sequencer are parametrised by the number of their control parameters and by the additional information to determine the exact control flow (the values for selecting a particular control parameter in the case of selectors and sequencing indices for control parameters in the case of sequencers). Design instances of data join and switch are parametrised by the number of their input and output data ports, respectively, as well as with values of decision data used by a data coordinator to select a respective input or output port. We do not define design instances of data channels since they are treated differently by the template instantiation procedure (see Section 8.4)

Design connector instances in a connector template are associated with a role from the template's interface. The participant information supplied during instantiation of the template is used to set the arity of the template's design instances (effectively converting them to deployed instances).

Design connector instances are composed by the same means as their deployed counterparts (Section 7.2.2) since they have the same interface elements – data ports, control ports and control parameters.

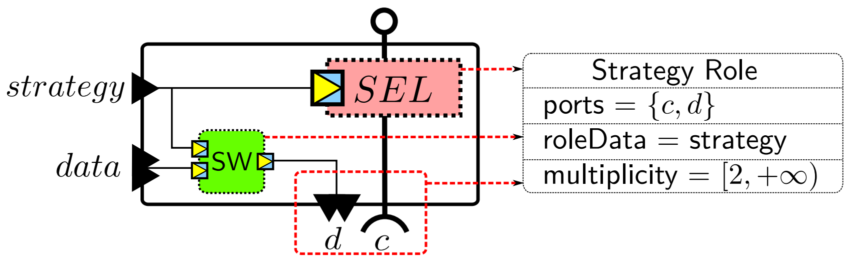


Figure 8.5: Trigger-Strategy Template with Variability

Example

To illustrate composition of design connector instances within a connector template, we show the connector template of Trigger-Strategy in Figure 8.5. The template has the same interface as in Figure 8.2, with the role Strategy defining the interface for participating Strategy components. It is composed of design instances of data switch and selector, defining control and data flow routing within the template, and of four data channels, defining data flow within the template. Both design instances are associated with the Strategy role.

8.3.2 Composing Connector Templates

To reuse interaction patterns stored in a repository as connector templates in the definition of a new connector template, one needs to create a new connector template by composing *instances* of the reused templates. That is, connector instances are a means for connector template composition. An existing connector template can be instantiated into the context of a template under construction as either a deployed connector instance or a design connector instance.

Deployed composite connector instances have their interface and composition structure fixed. The composition of these instances follows the principles described in Section 7.2.2. For example, Conditional Trigger-Strategy template in

Figure 7.8 composes a deployed instance of Trigger-Strategy.

On the other hand, design connector instances still have some of their template parameters – such as a port's data type or interface elements associated with an uninstantiated role – unassigned. Fixed parts of a design instance's interface can be composed with other entities inside the new template or propagated to the template's interface in the same way as interfaces of deployed instances. Variable parts of its interface have some restrictions on which other elements inside the new template they can be connected to; however, their unassigned variability parameters have to be always propagated to the interface level of the new template.

Example

To illustrate the composition of a design connector instance within a connector template, we show the connector template of Conditional Trigger-Strategy (introduced in Section 7.1.1) in Figure 8.6. It contains a design instance of the Trigger-Strategy connector template from Figure 8.5. The Trigger-Strategy template defines two kinds of variability parameters: multiports and roles. Because none of its parameters has been fixed during instantiation of the design instance, they propagate to the Conditional Trigger-Strategy template. The Strategy role is directly propagated by connecting its associated ports to the corresponding ports of the new template (c, d). Additionally, one of the ports aggregated by the role is a multiport. This is propagated to the interface of the new template as well. The propagation entails establishing the sourceMultiport relation between d and *data*.

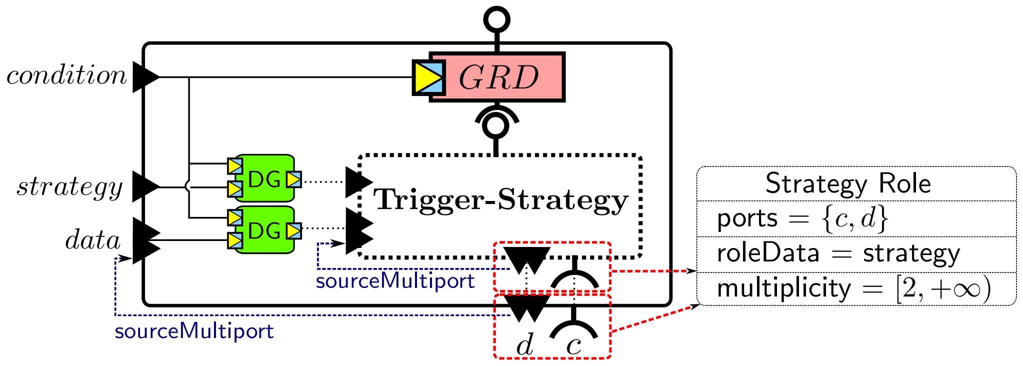


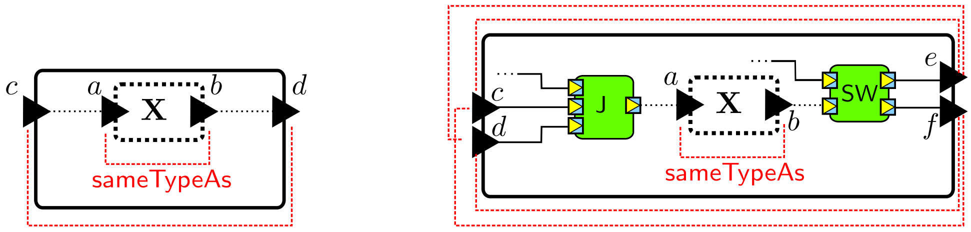
Figure 8.6: The Conditional Trigger-Strategy Template

Let us consider the three kinds of template parameters defined in Section 8.2

– generic types, multiports and roles – and discuss how they can be propagated via design connector instances to the interface of the new template.

Generic Types

They are represented by the sameTypeAs relation between pairs of required data ports in a connector template. In the simplest case, a pair associated with the relationship in the interface of a design instance is connected to a corresponding pair of ports in the interface of the new template, as shown in Figure 8.7a, which results in the propagation of the relationship. More formally: if a, b are required ports of a design connector instance, $(a, b) \in \mathit{sameTypeAs}$, c, d are required ports of the new template, a is connected to c and b is connected to d, then the interface of the new template contains the constraint $(c, d) \in \mathit{sameTypeAs}$.



(a) Direct Propagation (b) Indirect Propagation

Figure 8.7: Propagating Generic Types from Design Instances

In general, the ports of a design instance associated via sameTypeAs can be connected to the ports of other connectors comprising the new template as long as they are not treated as decision data. In this case (illustrated by Figure 8.7b), there exists a set A of required output ports of the new template ($\{c, d\}$) that can (directly or indirectly) supply data to the required output port of the associated pair (a), as well as a set B of required input ports of the new template ($\{e, f\}$) that can (directly or indirectly) receive data from the required input port of the associated pair (b). The sameTypeAs relation between the required input and output ports of the design instance is propagated to the level of the new template interface by creating the sameTypeAs association between all elements of Cartesian product $A \times B$.

Multiports

Pairs of multiports in a connector template are associated using another binary relation, sourceMultiport. The relation propagates from required multiports of design instances to required multiports of the new template, similarly to sameTypeAs. In particular, the case of direct propagation (analogous to Figure 8.7a) is the same; see the multiport propagation in Figure 8.6. The case of indirect propagation is simpler due to different semantics of the two relations. Because each required input multiport must have exactly one corresponding source multiport, required input multiports connected to the design instance's required input multiport b (the set B) can only have one required output multiport in the template interface as their source multiport. Consequently, the required output multiport a in the design instance's interface can be connected to only one corresponding multiport in the templates interface (the singleton set A). The relation is then propagated to the pairs comprising $B \times A$.

Roles

Roles that have not been instantiated in the interface of a design instance propagate to the interface of the new template. The required data ports and control parameters comprising such a role may be either connected directly to the corresponding interface elements of the new connector template, or indirectly via a control or data path comprising deployed instances[3].

Uninstantiated roles can propagate to the interface of the new template unchanged (see Figure 8.6); alternatively, their constituting ports can become aggregated to form a subset of a new role defined at the interface level of the new template. The latter case amounts to composition of roles – a role in a parent connector template can compose roles of several of its composite sub-connectors[4]. Participant information for the new role will be used to construct participant information for the composed roles during instantiation of the new template.

Figure 8.8 illustrates the role composition. Connector templates X and Y define roles of the same name. The templates are composed by means of their design instances in the new template (the right-most one in the figure). Instead of being directly propagated to the interface level of the new template, the roles are being composed by a new role XY simply by being connected to the interface elements associated with that role. Participants of the role XY will be used to construct participants for the roles X and Y; therefore, both X and Y will be instantiated using the same number of participants. It is the designer's responsibility to ensure that multiplicity and role data of a new composite role conform

[3]Section 8.3.3 explains the limitation forbidding the nesting of design instances.

[4]Role composition is not yet supported in our prototype tool.

to the constraints of its composed roles.

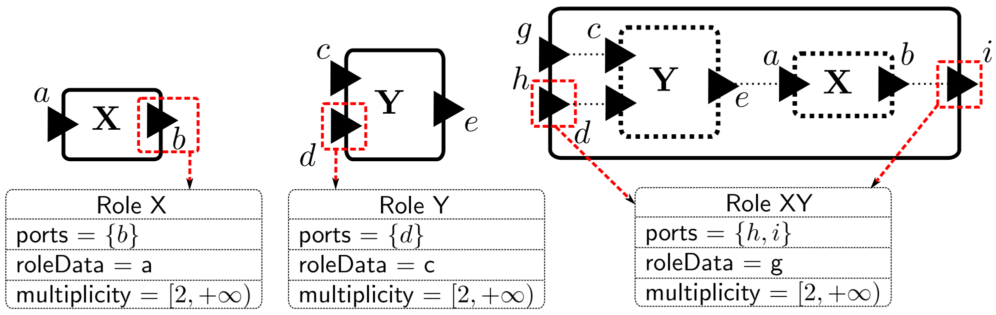


Figure 8.8: Role Composition

8.3.3 Constraints

To complete the discussion on the structure of connector templates, we summarise the rules that designers need to comply with for the instantiation process to be able to create template instances correctly. These restrictions stem from the properties of the instantiation process (Section 8.4), and specifically its part dealing with creating composition structure of a new instance. Their list follows:

- Design instances of data coordinators or control connectors have to be associated with the role whose instantiation fixes their arity.
- Design instances have to be (indirectly) connected to the interface elements of its associated role to propagate their variability parameters to the interface level of the new template.
- Deployed instances of data coordinators and control connectors have to be associated with a role in order to be duplicated during instantiation.
- There can be at most one design instance on every data or control path in a connector template.

Let us explain the meaning of the last rule. A data path corresponds to a sequence of adjacent data connector instances or composite connector instances, starting from a required output port and ending at a required input port; a control path is a sequence of adjacent control connector instances or composite connector instances, starting from the control port and ending at a control parameter of the template. The rule prevents, e.g., a situation in which a design selector instance

S has a design sequencer instance T as its child. Assume that S is associated with role a instantiated with s participants, T is associated with role b instantiated with t participants, and that T's parameter is propagated to the interface of its containing template. In our scenario, providing s participants to the role b would result in creating $s \times t$ copies of T's control parameter, instead of expected s copies, which violates the principles of interface instantiation of a connector template from Section 8.2. The rule prevents these cases.

8.4 Instantiation

Instantiation of a connector template is the process of (i) creating a connector instance from the template based on instantiation information and (ii) connecting the instance to other component model entities to integrate the interaction pattern, realised by the connector template, into the context of its instantiation.

The former corresponds to a transformation that, given instantiation information and a connector template, produces an instance of the template – its interface and composition structure. The latter corresponds to the process of composing the instance with other model elements, carried out by the designer in a modelling tool and assisted by automated connection constraint checking.

In this section, we give a complete overview of the instantiation process of connector templates. We first illustrate the process with examples in Section 8.4.1. Further, we describe the three processes that comprise instantiation: (i) creating the interface (Section 8.4.2), (ii) connecting the interface elements to other elements and associated constraint checking (Section 8.4.3), and (iii) creating the internal composition structure (Section 8.4.4).

8.4.1 Overview

In this section, we give an overview of how the instantiation transforms the interface and composition structure of a template to the interface and composition structure of its instances. We show examples of deployed instances of connector templates to illustrate the instantiation process.

To instantiate a connector template as a deployed instance, the designer needs to specify participant information for every role defined by the template. Each participant specifies its role and so-called participant data, a decision data value that identifies the participant as a target for routing control and data (see Section 8.2.4). The number of participants of a certain role must fall within the multiplicity interval defined by the role.

This information is used to generate the instance's interface from the template's interface. The instance's interface contains all interface elements of its template. The interface elements not associated with any role in the template appear once in the instance's interface; the elements associated with a role are multiplied as many as times as there are participants for that role specified during instantiation.

The composition structure of the instance is generated from the template's composition structure in a similar manner. Connector instances in the template not associated with any role are copied to the instance's structure. The connector instances associated with some role are transformed by instantiation depending on their type: deployed instances are duplicated according to the number of participants of their role; design instances are recursively transformed so that their interface and composition structure is fixed based on the associated participant information. The transformation also traces data and control paths and duplicates them if necessary.

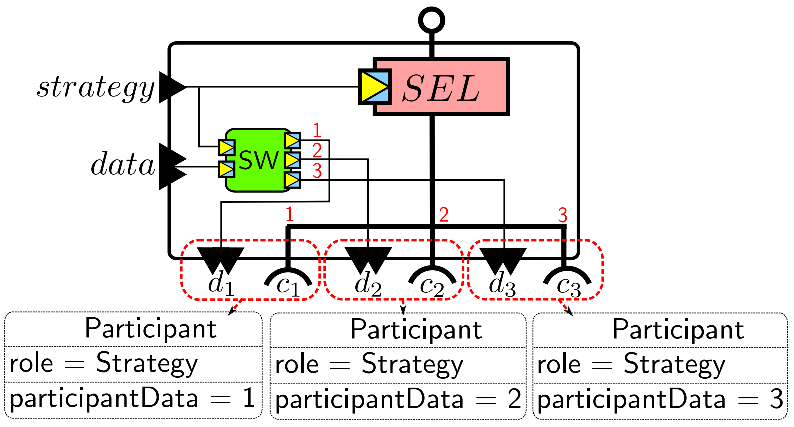


Figure 8.9: A Trigger-Strategy Instance

Figure 8.9 shows the result of instantiating the Trigger-Strategy template (see Figure 8.5) with three participants of the Strategy role. The interface of the instance changed accordingly: ports c, d from the template are multiplied thrice. In the instance's composition structure, design instances from the template have been set the arity equal to the number of participants of the Strategy role and have thus become deployed instances. Each of the selector's control parameters has been annotated with the participant data of the respective participant so that the selector's control routing behaviour is fully defined. Likewise, the output ports of the data switch have been annotated with the decision data values of corresponding participants.

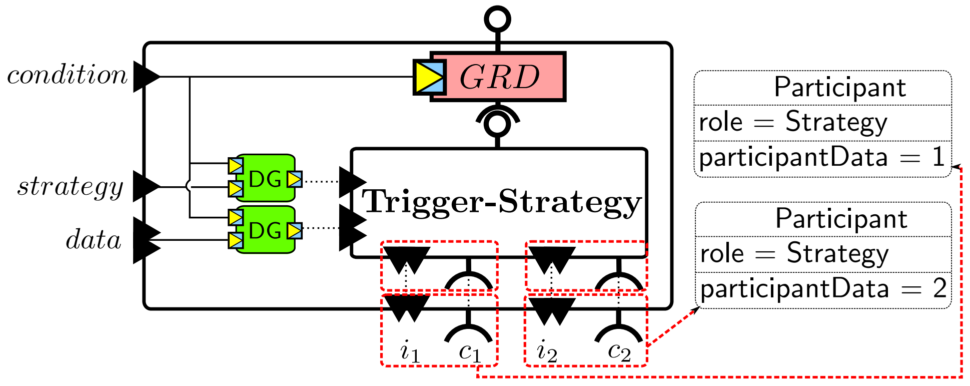


Figure 8.10: A Conditional Trigger-Strategy Instance

To illustrate the recursive nature of composition structure generation, Figure 8.10 shows a deployed instance of Conditional Trigger-Strategy (see the template in Figure 8.6). The Strategy role has been instantiated with two participants. The design Trigger-Strategy instance has become a deployed instance – with its interface and composition structure recursively transformed – after all of its variability parameters have been fixed using the instantiation information of its parent template.

To instantiate a connector template as a design instance, not all roles have to be provided with participants. Parts of the interface of a design instance associated with such uninitialised roles are identical to the corresponding parts of the template's interface; parts of the interface associated with initialised roles (with participants) are instantiated as in the case of deployed instances. The uninitialised roles are propagated to the interface of the parent connector template. The composition structure of design instances is not generated during their instantiation, because not enough information is available at that time. Instead, their composition structure is generated when their parent connector template is being instantiated as a deployed instance, as exemplified by the Trigger-Strategy instance in Figure 8.10.

8.4.2 Interface Creation

In this section, we specify a procedure that creates the interface of a connector template instance based on the participant information supplied by the designer. The procedure is executed after a connector template is chosen to be instantiated from the repository. The pseudocode of the procedure follows:

```
procedure CreateInstanceInterface($c$, $participants$)
    $template \leftarrow$ template of $c$
    for all $element \in$ interface elements of $template$ do
        $role \leftarrow$ role associated with $element$
        $roleParticipants \leftarrow \{p \in participants | p.role = role\}$
        if $roleParticipants = \emptyset$ then                ▷ No role, or an unitialised one.
            Add a copy of $element$ to $c$'s interface
        else                                                  ▷ An initialised role.
            for all $p \in roleParticipants$ do
                Add a copy of $element$ to $c$'s interface
            end for
        end if
    end for
end procedure
```

The procedure iterates through the elements of the interface of *c*'s template; each *element* is either copied to the interface of the new instance, if there is no role associated with it or the designer has decided – when creating a design instance – not to initialise the associated role with any participant, or duplicated as many times as there are participants of its associated role.

8.4.3 Connecting Instance Interface

To complete the integration of the interaction pattern represented by a connector template into the context of its instantiation, the designer needs to connect the instance's interface elements to other model elements' interfaces. This activity is supported by the connection constraint checking process, which triggers its checking routine whenever an instance's interface elements are connected to or disconnected from other model elements. The routine checks whether the connected interface elements comply with the constraints imposed by the connector template – checking data types and directionality of ports; the result (a boolean) can be used to prevent a connection from being created or it can be displayed to the designer as an error.

The rules for connecting interface elements of a composite connector instance are as follows:

- A required data port can be connected to one or more[5] component data ports of the right type and directionality, or to a required data port of a connector template with the same constraint (if instantiated within the template).

- A required output multiport can be connected to many component data ports, or to a required output multiport of a connector template (if instantiated within the template).

- A required input multiport can be connected to the same number of component data ports (of the same types and opposite directionality) as its corresponding output multiport, or to a required input multiport of a connector template (if instantiated within the template).

- A control port can be connected to one or more control parameters, or to a control port of a connector template (if instantiated within the template).

- A control parameter can be connected to one control port, or to a control parameter of a connector template (if instantiated within the template).

Type checking of required data ports amounts to checking that the data type of the connected port equals the required data type of the required port if the type is specified directly. If generic type specification is used, the data types of the ports connected to an instance's ports in the same equivalence class of the sameTypeAs relation should be the same.

8.4.4 Composition Structure Creation

In this section, we describe the transformation that generates the composition structure of a deployed connector instance from the connector template and instantiation information to fully define the connector's behaviour at run-time.

[5]The semantics of multiple component ports connected to a single required data port depends on the directionality: a required output port consumes the data from one *non-deterministically chosen* port of the connected ports; a required input port *copies data to all* connected ports.

We formulate it as a procedure CreateComposition that takes a deployed connector instance *dc* and the instance's participants as its inputs and creates the internal composition structure of the instance. We assume that the interface of the instance has already been created using the CreateInstanceInterface. The following pseudocode gives a high-level description of the procedure:

```
 1: procedure CreateComposition($dc, participants$)
 2:     $t \leftarrow$ template of $dc$
 3:     $map \leftarrow \emptyset$                    ▷ Maps template elements to their instance copies
 4:     $partMap \leftarrow \emptyset$    ▷ Maps instance elements to a corresponding participant or $\emptyset$

 5:     $deployedNoRole \leftarrow$ deployed instances of control connectors, data coordinators
                          and composite connectors, with unassigned role
 6:     for all $x \in deployedNoRole$ do
 7:         copy $x$ from $t$ to $dc$, update $map$ and $partMap$ accordingly
 8:     end for

 9:     $roles \leftarrow$ roles defined by $t$
10:     for all $role \in roles$ do
11:         $roleParticipants \leftarrow \{a \in participants | a.role = role\}$
12:         $p \leftarrow |roleParticipants|$
13:         for all $y \in$ deployed instances in $t$ of any type, associated with $role$ do
14:             copy $y$ from $t$ to $dc$ $p$ times, update $map$ and $partMap$ accordingly
15:         end for
16:         $designNoCC \leftarrow$ design instances in $t$ of non-composite connectors,
                          associated with $role$
17:         for all $z \in designNoCC$ do
18:             deploy $z$ with arity $p$ from $t$ to $dc$, update $map$ and $partMap$ accordingly
19:         end for
20:     end for

21:     for all $c \in$ design instances in $t$ of composite connectors do
22:         $d \leftarrow$ create a deployed instance of $c$ in $dc$, update $map$ and $partMap$ ac-
    cordingly
23:         $ct \leftarrow$ template of $c$
24:         $dp \leftarrow$ gather participants for roles defined by $ct$ from $participants$
25:         CreateInstanceInterface($d, dp$)
26:         CreateComposition($d, dp$)      ▷ Recursive call to create composition of $d$
```

```
27:     **end for**

28:     **for all** $a \in$ data channels, control parameters, and identity connections in $t$ **do**
29:         $sources \leftarrow map($the source of $a)$    ▷ A non-empty set of elements from $dc$
30:         $targets \leftarrow map($the target of $a)$
31:         **for all** $(s, t) \in sources \times targets$ **do**
32:             **if** $(partMap(s) = \emptyset \vee partMap(t) = \emptyset) \vee partMap(s) = partMap(t)$ **then**
33:                 Create a copy of $a$ in $dc$ from $s$ to $t$
34:             **end if**
35:         **end for**
36:     **end for**
37: **end procedure**
```

The procedure first copies deployed connector instances (other than data channels) not associated with any role (and thus not multiplied in the instance) from the template to the instance (lines 5-8). It then copies deployed instances of connectors (other than data channels) associated with some role as many times as there are participants of the role (lines 10-15). Design instances of non-composite connectors (other than data channels) are converted to deployed instances, their arity is set to the number of participants of their associated role (lines 17-19).

Further, design composite connector instances are transformed (lines 20-26): each design instance is converted to a deployed one, with its interface initialised by the procedure described earlier in this section (CreateInstanceInterface); the recursive call (line 26) creates the composition structure of the new deployed connector instance.

Finally, all connections (comprising data channels, control parameters and identity connections) from the template are created in the instance (lines 28-36). Since some of the entities have been copied to the instance multiple times, there can be multiple connections created for a single connection in the template. The procedure iterates through each connection in the template and duplicates the connection between every copy of the connection's source and every copy of the connection's target if they correspond to the same participant or at least one of them is not associated with any role. This condition correctly deals with all possible combinations of design or deployed instances, with or without roles, under the following assumptions: (i) a connection between two design instances is forbidden, and (ii) a connection cannot connect entities associated with two different roles.

In the pseudocode, *map* and *partMap*, which are being built throughout the execution of the procedure, help us determine what instance elements correspond to a template element and what participant corresponds to an instance element, respectively.

8.5 Discussion

In this section, we discuss the impact of connector templates on connector reuse in Section 8.5.1 and possible alternatives to the realisation of template parametrisation mechanisms in Section 8.5.2.

8.5.1 Reuse

Connector templates further increase the reuse potential of connectors in our component model by parametrising their various aspects. In particular, they address the shortcomings of composite connector instances in representing interaction patterns identified in Section 7.4.2:

- they allow some required data ports to have generic data types, while still maintaining type safety of instance composition,
- they introduce an interface element type (multiports) representing a set of data flows and
- they allow connector instances to vary their interface and composition structure (and thus behaviour) by means of the role mechanism.

The structural variability of reusable abstractions is not common in component-based software development. Components, the dominant reuse abstractions in component-based software development, are mostly viewed as black boxes. They comprise a public interface and hidden implementation, which defines their behaviour and is essentially a piece of software, often provided in the compiled binary form, written in some programming language. Because of the black box nature of their implementation, component interface and implementation are fixed and cannot vary across their instances.

Unlike black-box components, whose behaviour is defined by opaque implementation, our composite connectors are fully defined as compositions of simpler component model elements. This enables their flexible instantiation, which can even affect their compositional structure. As a result, connectors can be instantiated in a greater variety of contexts, i.e., they have greater reuse potential than corresponding black-box components.

8.5.2 Alternative Parametrisation Mechanisms

In this section, we discuss possible alternative realisations of the template parametrisation mechanisms of generic types, multiports and structural variability.

Generic Data Types

The proposed mechanism is similar to the type parametrisation mechanism found in programming languages such as Java or C++, in which the definition of a class and its members may contain type variables that are fixed during class instantiation (at run-time) and that represent a similar type constraint. Indeed, it would be possible to use the type variable mechanism. However, there is no difference in expressiveness, provided that we allow the sameTypeAs relation between any two required ports, and the approach using model associations has a simpler meta-model realisation.

Multiports

A possible alternative solution to the problem of multiplexing several data flows would be to extend the type system to include composite data types, such as arrays or records. That would also lead to extending the set of model elements by operators for multiplexing and demultiplexing of simple streams to complex ones and back. However, using statically typed arrays or records in a connector template definition would not be enough: arrays require homogeneous constituent elements, while records have a fixed number of elements, which would limit the reuse of connector templates in the same way as the solution without multiports.

During binding multiports to data ports, the multiport mechanism effectively dynamically creates a composite type with elements of varied types addressable by names (of originating data ports), in a type-safe manner. The closest alternative, feasible in static type systems, would be to represent a multiport in a template using a type variable, which would be assigned a composite type in the template's instance.

Structural Variability

The presented solution to the variable composition structure of composite connectors relies on a single instantiation transformation, common to all connector templates. This solution does not make connector template design overly complex, and it allowed us to parametrise the structure of most connector templates on which we evaluated our method. However, it is by no means the only solution.

One alternative would be to have such a transformation procedure specific to each template: the designer of a template would deposit both a template and its instantiation procedure in the repository. The degree of variability of the template would depend on its instantiation procedure, which could be customised to the represented interaction pattern. Certainly, this could lead to more flexible instantiation. On the other hand, since instantiation would be template-specific, the template would become an obsolete (constant) parameter, which could lead – in the extreme case – to designers only storing instantiation procedures in the repository. If the transformation was written in some generic transformation (or even programming) language, our aim of designing the behaviour of interaction patterns by composing component model elements would be defeated.

The presented solution to the structural variability of templates and its aforementioned variant can be seen as two extremes of a wider spectrum; the former being common to all templates, the latter being template-specific. Another option would be to create a domain-specific language for this purpose, with constructs reusable across many connector templates, but with the ability to be attached and customised to a particular template. However, to develop useful constructs of such a language systematically would require substantial research involving the analysis of a large number of patterns, which is out of the scope of this thesis.

Chapter 9

Model-driven Implementation

To evaluate and experiment with our approach, we have implemented a prototype tool. It allows designers to create components and composite connectors, deposit them in a repository and deploy them to compose systems conforming to our component model.

This chapter illustrates how we used various model-driven techniques to implement our prototype; in particular, instantiation of composite connectors is a novel application of model transformations in the context of component-based software development.

In Section 9.1, we give an overview of the prototype tool from the designer's perspective. Section 9.2 briefly describes the underlying meta-model. The use of model transformations in the prototype is explained in Section 9.3. Section 9.4 describes how the execution semantics is implemented in the simulator. The chapter concludes with the discussion of alternative ways of defining model transformations and of some performance issues in Section 9.5.

9.1 Prototype Tool Overview

The prototype tool allows one to design and simulate systems conforming to our component model. Designers can design and implement components and connectors, reuse them by means of repositories to compose systems, and validate system behaviour by simulation. The tool supports the connector life cycle (connector templates and two kinds of their instances) and variability parameters of connector templates described in Chapter 8.

The main development activities are supported by the following modules:

component editor, connector template editor, repository, system architecture editor and simulator. The modules are integrated to form one design environment, centred around the repository module (see Figure 9.1[1]). We used the Eclipse Platform as an integration platform; individual modules were realised as (sets of) plug-ins extending the core platform.

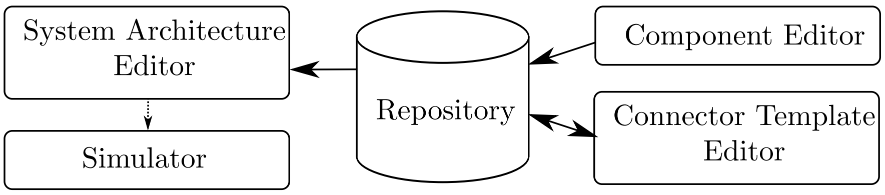
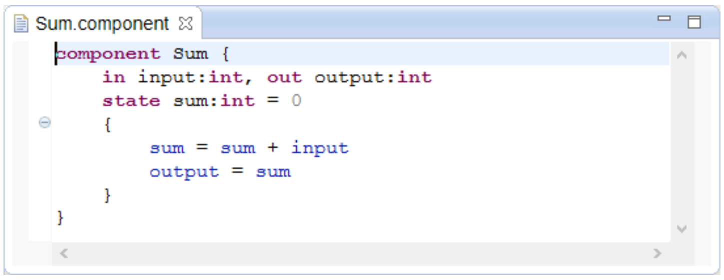


Figure 9.1: Modules Comprising The Prototype Tool

In the remainder of this section, we describe the function of individual modules from the designer's perspective.

Component Editor

The component editor allows the definition of components by means of a textual, domain-specific language. The language can specify both a component's interface, in terms of ports, and behaviour, by defining arithmetic expressions for computing values of output ports from the values of input ports and state variables[2]. An example of a component defined in the editor is shown in Figure 9.2.



```
component Sum {
    in input:int, out output:int
    state sum:int = 0
    {
        sum = sum + input
        output = sum
    }
}
```

Figure 9.2: Component Editor

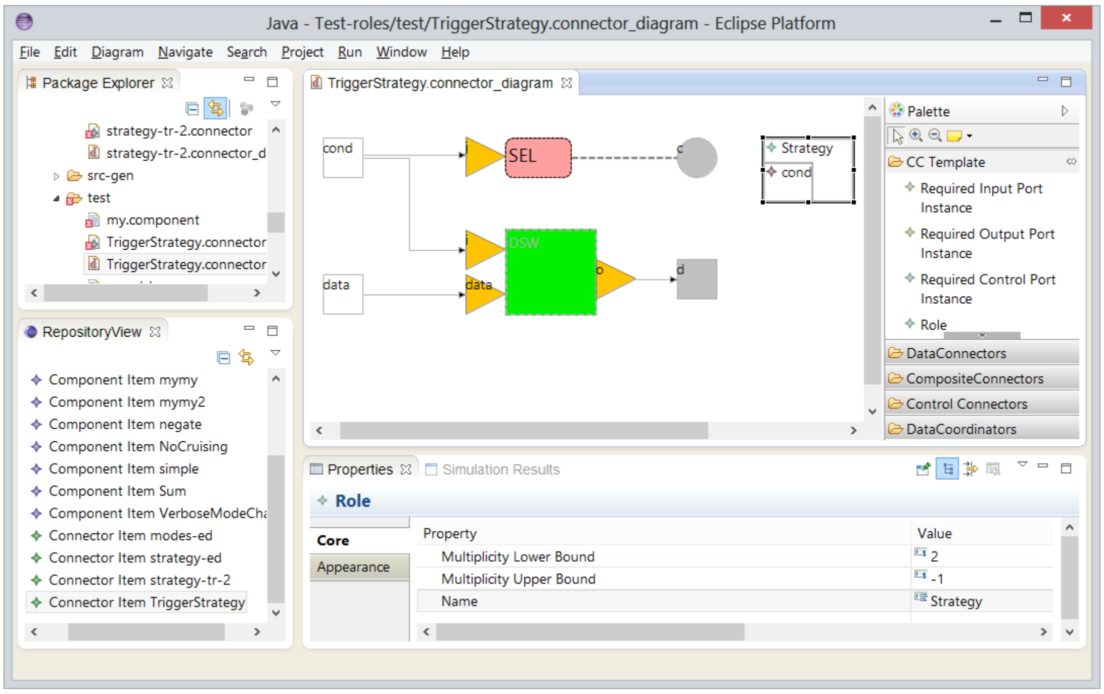
Connector Template Editor

This graphical editor serves to define connector templates – their interface

[1]The solid arrows in the figure denote components and connectors being deposited in, and retrieved from, the repository.

[2]If more complex behaviour is required, designers can develop components as Java classes implementing a particular interface.

and internal composition structure. For example, the Trigger-Strategy template in the editor is shown in Figure 9.3. The editor has a palette of interface elements and component model elements that can be composed within a composite connector (on the right in the figure); it is also integrated with the repository, from which existing connector templates can be instantiated into the currently designed connector template (see the bottom left view in the figure).


Java - Test-roles/test/TriggerStrategy.connector_diagram - Eclipse Platform
File Edit Diagram Navigate Search Project Run Window Help
Package Explorer
strategy-tr-2.connector
strategy-tr-2.connector_d
src-gen
test
my.component
TriggerStrategy.connector
TriggerStrategy.connector
RepositoryView
Component Item mymy
Component Item mymy2
Component Item negate
Component Item NoCruising
Component Item simple
Component Item Sum
Component Item VerboseModeCha
Connector Item modes-ed
Connector Item strategy-ed
Connector Item strategy-tr-2
Connector Item TriggerStrategy
TriggerStrategy.connector_diagram
cond
SEL
c
Strategy
cond
i
DSW
o
d
data
data
Palette
CC Template
Required Input Port Instance
Required Output Port Instance
Required Control Port Instance
Role
DataConnectors
CompositeConnectors
Control Connectors
DataCoordinators
Properties
Simulation Results
Role
Core
Appearance

Property	Value
Multiplicity Lower Bound	2
Multiplicity Upper Bound	-1
Name	Strategy

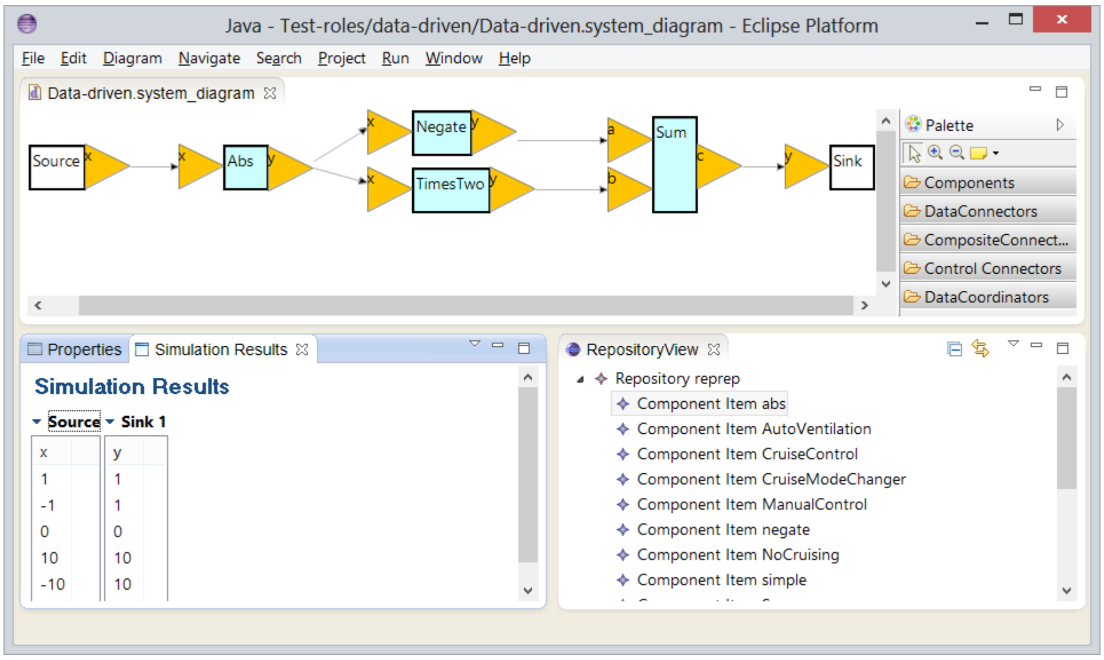
Figure 9.3: Connector Template Editor

Repository

The repository module manages available repositories that contain components and connector templates. Users can explore the contents of the repositories, deposit components or connector templates in them, and instantiate selected entities into the connector template editor and the system editor. The repositories view is shown on the left side of Figure 9.3.

System Architecture Editor

This graphical editor enables the designer to construct systems in our model by composing deployed instances of component model elements using data and control connectors. Visually, it is similar to the connector editor: it has the palette of component model elements that can be composed within a system, and it is also integrated with the repository module, from which existing connector templates and components can be instantiated into the currently designed system. Figure 9.4 shows a purely data-driven system in the editor.



Simulation Results

Source Sink 1

x	y
1	1
-1	1
0	0
10	10
-10	10

Repository reprep
- Component Item abs
- Component Item AutoVentilation
- Component Item CruiseControl
- Component Item CruiseModeChanger
- Component Item ManualControl
- Component Item negate
- Component Item NoCruising
- Component Item simple

Figure 9.4: System Architecture Editor

Simulator

The simulator executes a well-defined system, constructed in the system architecture editor, in accordance with the execution semantics defined in Section 6.5. After the simulation terminates, it displays the report showing inputs and outputs generated by the simulated system (see the bottom-left tab in Figure 9.4).

To develop the plug-ins, we have used model-driven frameworks from the Eclipse Modelling Project [119]: Eclipse Modelling Framework (EMF) for meta-modelling (see Section 9.2), Graphical Modelling Framework for building graphical editors based on EMF meta-models (the connector template and system editors) and XText for building the component specification language and the associated component editor. We have also experimented with Henshin for defining model-to-model transformations using graph rewriting rules (Section 9.5.1).

9.2 Meta-Model

In this section, we describe the meta-model underlying the model-driven implementation of our prototype tool.

Meta-modelling is a key activity in model-driven software engineering; in general, it aims to create models of some aspects of a software system under development, which are used to generate (skeletons of) software artefacts comprising that software system. In our prototype tool, we use meta-modelling to define the structure of our component model, out of which the skeletons of implementation of various editors are automatically generated.

In our implementation, the meta-model is defined using the ECore language, which provides a number of concepts well-known from object-oriented structural modelling: classes, attributes and inter-class associations. The ECore meta-model forms one of the inputs to the following automated transformations:

- code generation of the meta-model entities (by EMF),
- code generation of the connector template graphical editor (by GMF),
- code generation of the system architecture graphical editor (by GMF) and
- code generation of the component editor (by XText).

Structurally, ECore meta-models form trees, class hierarchies defined by containment (by the composition association). They can have modular structure by being further subdivided into packages.

The meta-model representation of our component model in our prototype tool comprises three top-level packages: *component*, *connector* and *system*. The *component* package contains the structural descriptions of components and component instances, their interface elements and some basic data types shared by other two packages. The *connector* package is further subdivided into three packages – *cctemplate*, *design* and *deployment* – defining connector templates and basic connector types, their design and deployed instances, respectively. Finally, the *system* package defines the concept of a system, the root of the meta-model's class hierarchy.

Since the meta-model defines the structure of our component model in accordance with Chapters 6, 7 and 8, we do not discuss it in detail in this section. The interested reader can find the ECore representation of the meta-model for reference in Appendix B.

For illustration, we only show a small excerpt from the meta-model as a class diagram in Figure 9.5. It depicts the part of the meta-model defining the binding of a required port of a connector template's deployed instance.

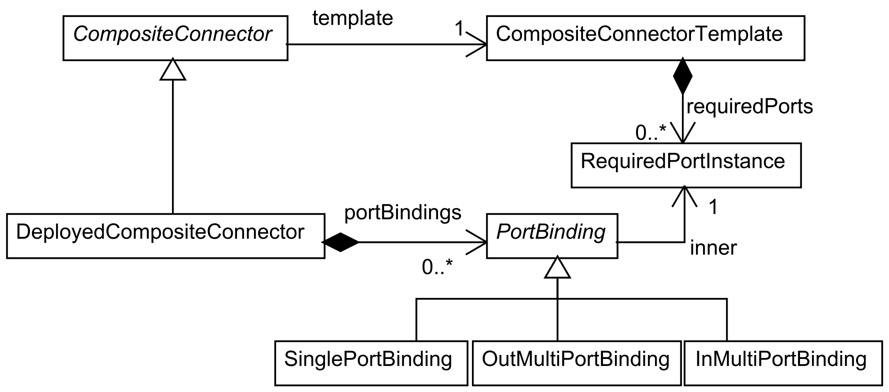


Figure 9.5: Port Binding in Deployed Composite Connector Instances

Deployed connector instances are modelled by the class DeployedComposite-Connector inheriting from CompositeConnector, the common ancestor of both kinds of instances that crucially contains a reference to the CompositeConnector-Template class. This meta-modelling pattern for representing commonalities between design and deployed instances of a component model entity via a common abstract class is used throughout the meta-model.

Notice that the meta-model differentiates between the interface of an instance and the interface of a connector template: whereas templates comprise required ports (RequiredPortInstance), their deployed instances have (the same number of) port bindings. Each port binding refers to its corresponding required port (via the inner association) and to the entity to which that required port is connected for this particular instance (not shown in Figure 9.5). The relation between port bindings and required ports is analogous to that of actual and formal parameters of a procedure. In the discussion of interfaces of composite connector templates in Chapter 8, we have ignored this difference for the sake of clarity, but the meta-model does not allow such imprecision.

The PortBinding class is abstract (its name is set in italics) and has three concrete sub-classes. This represents the fact that there are different kinds of required ports of connector templates, which require different binding information. SinglePortBinding holds the binding information of ordinary required ports; OutMultiPortBinding and InMultiPortBinding encapsulate binding information for multiports of respective directionality.

9.3 Application of Model Transformations

A model transformation is 'an automated process that takes one or more models as input and produces one or more target models as output, while following a set of transformation rules.' [102]; it is assumed that the models involved conform to some meta-models, which are used to define transformation rules. In model-driven engineering, model transformations can play many roles: they can maintain consistency between models, refine abstract specification models to more concrete implementation models, perform refactoring, implement migration between database schemas, etc.

In our prototype tool, we use model transformations

- to construct interfaces of instantiated connector templates,
- to construct composition structure of instantiated connector templates,
- to convert instances of composite connectors with multiports to their equivalents without multiports, and
- to decompose instances of composite connectors to composition of basic connectors.

All of the above transformations are endogenous; i.e., their input and output models conform to the same meta-model, that of our component model (Appendix B). The former two transformations help realise the variability of connector templates; the latter two transformations convert more high-level models to their low-level counterparts that exhibit the same behaviour so that the simulator operates on a simplified version of a system (see Section 9.4.1).

In our prototype tool, transformation rules comprising the above model transformations are implemented in a general programming language (Java), using the object-oriented representation of the meta-model generated by EMF to modify the transformations' input models and produce the output models.

Alternatively, model transformations can be implemented by means of specialised model transformation languages. We have experimented with such languages based on graph rewriting to define and to formalise model transformations in our prototype tool. We have shown that it is possible (see Section 9.5.1), but we did not use these languages to implement model transformations in our prototype due to the increased size and complexity of their specification compared to Java.

9.4 Simulator

The simulator simulates the execution of systems designed in the system architecture editor; it takes a system's architecture description and simulator configuration[3], and runs the system in accordance with the model's execution semantics. A simulator configuration (i) associates sources and sinks in the system specification with input and output files, and (ii) initialises the state of some component model elements, such as component execution states or the contents of some data channels. Once the user terminates the simulation, the simulator updates the Simulation Results view with the summary of values that were input to and output by the simulated system. The simulator also saves the system's outputs to the files associated with the sinks, and prints the trace of system execution for debugging purposes.

We use the term 'simulation' rather than 'execution' to stress that systems are not being compiled or deployed to any target execution (hardware or software) platform. The simulator interprets a system architecture description and executes the Java code of its constituent components in order to simulate the system's behaviour. Its aim is to validate the run-time behaviour of designed systems.

In this section, we describe what transformations a system representation undergoes before simulation (Section 9.4.1), and we give an overview of the implementation of the execution semantics in our prototype (Section 9.4.2).

9.4.1 System Preprocessing

A system architecture description is preprocessed in several steps before the system simulation starts. The flowchart in Figure 9.6 gives an overview of these steps. First, a system architecture undergoes a series of model transformations that simplify the system's description. Based on the description, the run-time representation of the system is constructed, which is then used by the interpreter to simulate the system's behaviour. Let us describe these steps in more detail.

Model Transformations Simplifying a System's Description

Before it is further processed by the simulator, a system architecture description undergoes three model transformations (Section 9.3), all of which are related to

[3]In the prototype, the configuration information is specified directly in the architecture editor.

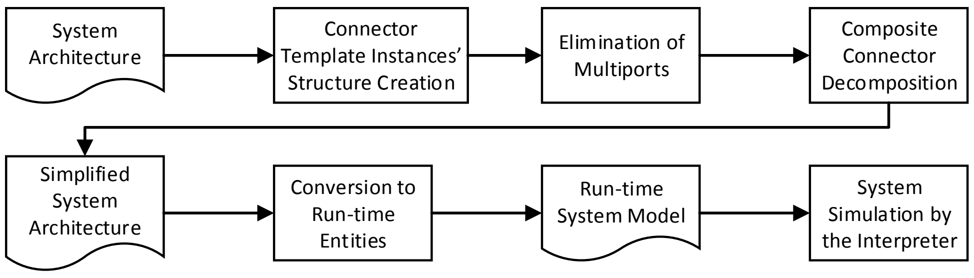

System Architecture
Connector Template Instances' Structure Creation
Elimination of Multiports
Composite Connector Decomposition
Simplified System Architecture
Conversion to Run-time Entities
Run-time System Model
System Simulation by the Interpreter

Figure 9.6: System Architecture Preprocessing for Simulation

composite connectors. The first one fully creates the composition structure of connector template instances (Section 8.4.4); the second one removes multiports and replaces them with sets of single ports (Section 8.2.2); the third one removes the instances of connector templates altogether by decomposing them into basic connectors (Section 7.3.3). The result of the simplifying transformations is a system description of simpler structure and equivalent behaviour, which simplifies the design of other parts of the simulator.

Run-time System Representation

The classes defined in the meta-model (Appendix B) represent the design-time structure of our component model. To interpret a system architecture at its run-time, the simulator contains a set of classes that extend their meta-model counterparts with (i) additional run-time state and (ii) the implementation of their run-time behaviour in accordance with the component model's semantics.

For instance, run-time components specify Java classes implementing their execution behaviour, run-time control-switched data-driven components additionally keep their execution state (enabled/disabled), data ports have associated state determining whether they are full or empty, and run-time data connectors implement the semantics specified by their design-time counterparts (FIFOs, non-destructive read channels).

Due to the simplifying model transformations, not all meta-model entities have their run-time counterparts.

Interpreter

The interpreter takes a run-time system model and simulates the system's behaviour by running the control thread and data flow scheduler on the system model in accordance with the execution semantics of our component model, as defined in Section 6.5. Section 9.4.2 describes the implementation of the execution semantics in more detail.

9.4.2 Implementation of Execution Semantics

The implementation of execution semantics in the prototype's interpreter module respects the principles laid out in Section 6.5: there are two schedulers, the control thread and data-driven scheduler, performing tasks summarised in Figure 6.25; the schedulers are synchronised by splitting the execution into a series of execution cycles and additional synchronisation rules ensuring deterministic system behaviour (Section 6.5.6). There are two main decisions to be taken to implement these principles: the threading model and the execution synchronisation mechanism.

Threading Model

The threading model determines the number of threads and allocates to each thread the tasks to carry out. A possible straightforward implementation of the control thread and data-driven scheduler (Figure 6.25) is to dedicate one thread to each of them. However, that does not exploit the potential of data-driven computation to be parallelised.

In our prototype, we opted for a threading model schematically depicted in Figure 9.7: the control thread is realised by one thread, but the data-driven scheduler is implemented by a pool of threads (of a constant configurable size, not necessarily three as in the figure). Each request for a data-driven computation, i.e., computation carried out by a data-driven component or a data coordinator, is submitted to the thread pool, managing a queue of computation requests, from which unoccupied threads get their computation jobs. Thus, a thread in the thread pool can sequentially perform computation for multiple component model elements within one execution cycle, as suggested by discrete segments of the squiggly lines representing threads in the figure. The synchronisation of execution is managed by yet another thread, the master scheduler, which re-starts the two schedulers at the beginning of each execution cycle and waits for both of them to finish at the end of an execution cycle; it also spawns and destroys the scheduler's threads at the beginning and at the end of the simulation, respectively.

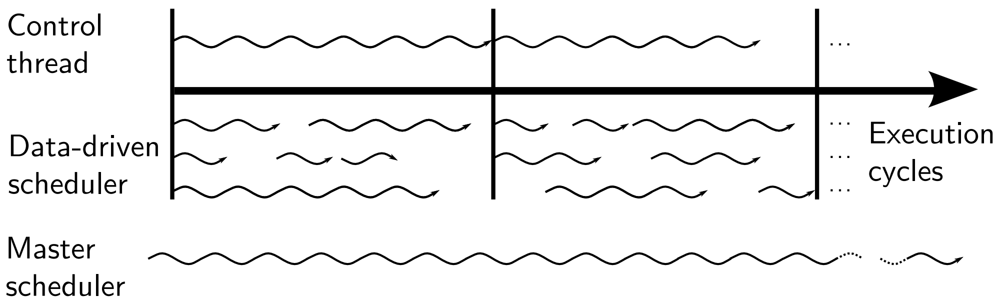


Figure 9.7: Threading Model Implemented in the Simulator

Execution Synchronisation Mechanism

There are several synchronisation mechanisms between the threads. They can be split into two groups: (i) the synchronisation between execution cycles and (ii) the synchronisation of the threads within an execution cycle. The former is enforced by the master scheduler; the latter amounts to the control thread waiting for missing data inputs of control connectors, of control-driven components before skipping their triggering and of control-switched data driven components before disabling them.

An important primitive used in the realisation of both kinds of synchronisation in our prototype is a global shared integer variable (with synchronised multi-thread accesses), *active*, that denotes the number of active data-driven computations (i.e., single invocations of data-driven components or data coordinators that have been submitted for execution to the thread pool of the data-driven scheduler but have not been executed yet). It is incremented each time before a computation request is submitted to the thread pool and decremented after a request has been processed. The master scheduler uses the variable to determine the inactivity of the data-flow scheduler at the end of an execution cycle, and the control thread uses it to detect whether a missing data input can be computed in the current execution cycle. The other aforementioned synchronisation mechanisms are implemented by active waiting (the master scheduler waiting for the control thread) and by passive waiting realised by means of condition variables (the control thread waiting for control connectors' inputs).

The following pseudocode illustrates the working of the master scheduler. It is defined as a function that takes a run-time model of a *system* to be simulated and returns the number of simulated execution cycles.

```
 1: function MainScheduler($system$)
 2:     Create the control thread $ct$ and data-driven scheduler's thread pool $tp$.
 3:     $cycle \leftarrow 0$
 4:     while the simulation not terminated by the user do
 5:         $active \leftarrow 0$                         ▷ Active data-driven computations
 6:         Submit computation requests to $tp$ for all data-driven entities
                   in $system$ with all inputs present.
 7:         Wake $ct$.                   ▷ $ct$ and $tp$ execute the current cycle now ...
 8:         Wait until $active = 0$ and $ct$ has finished one iteration.
 9:         $cycle \leftarrow cycle + 1$
10:     end while
11:     Kill $ct$ and $tp$.
12:     return $cycle$
13: end function
```

The initialisation of the simulation (lines 2-3) is followed by the main simulation loop (lines 4-10) that needs to be terminated by the user (the execution semantics defines non-terminating systems). Each cycle of the loop corresponds to an execution cycle. Each cycle starts by setting the synchronised global variable *active* to zero. Initial scheduling of data-driven computation and waking the control thread follow. At that moment, the control thread and the threads of the data-driven scheduler perform all the computation comprising the execution cycle. The master scheduler waits for both of them to finish, and it then proceeds to another execution cycle (denoted by incrementing the *cycle* variable at line 9). After the simulation has been terminated, the threads are killed and the simulation ends (lines 11-12).

9.5 Discussion

In this section, we discuss alternative ways of defining model transformations in Section 9.5.1, and we note some performance issues in Section 9.5.2.

9.5.1 Model Transformations as Graph Rewriting Systems

In this section, we introduce an alternative way of defining model transformations as graph rewriting systems and discuss the outcomes of our experiment with implementing model transformations in our prototype using this formalism.

In our prototype, we have implemented model transformations in a general, object-oriented programming language (Java). Alternatively, model transformations can be implemented in languages based on graph rewriting. Not only do graph rewriting techniques define model transformations, but they also provide a framework to formalise them since they themselves have formal semantics. In particular, we investigated Henshin [8], because it is well integrated into the EMF ecosystem.

A graph rewriting system comprises a set of rules; each rule has the form of $L \rightarrow R$, where L and R are sub-graphs, and represents an atomic modification of an input graph G in which a sub-graph of G isomorphic with L is replaced by R. A graph transformation corresponds to a series of applications of graph rewriting rules. In the context of model transformations, graphs comprise nodes with attributes (corresponding to meta-model classes) and edges (corresponding to inter-class associations); rewriting rules can also express modification of node attributes (hence the term attributed graph grammars).

Henshin is an implementation of such an attributed graph grammar. In addition, it has a graphical syntax for rules. Each rule is represented as an annotated class diagram that combines elements of both sides of the rule; each class and association in the diagram is annotated by one of several kinds of annotations – «preserve», «create», «delete», etc. An annotation determines on which side of the rule a particular element is: elements annotated with «preserve» are in both L and R (and thus preserved by an application of the rule), elements annotated with «create» are only in R (i.e., created by an application of the rule) and elements annotated with «delete» are only in L (removed by an application of the rule).

Having analysed the pseudocode of the transformation creating composition structure of connector template instances (Section 8.4.4), we identified several kinds of Henshin rules that correspond to atomic modifications performed by the transformation: (i) rules that copy deployed data coordinators, control connectors and connector template instances with no assigned role to the new template, (ii) rules that multiply deployed instances associated with a role in the new template, (iii) rules that deploy design instances of data coordinators, control connectors and composite components, and (iv) rules that create data channels and control parameters in the new template. We have defined a rule of each kind in Henshin to evaluate its feasibility for defining model transformations in our prototype (see the examples of transformation rules in Appendix C).

Compared to the imperative pseudocode, the coordination of these rules' application may be simpler; in particular, there is no need for recursion, which is used in the imperative implementation of our transformation. This is because the matching of sub-graphs on the left hand side of rules contains an implicit loop, which traverses an input graph in search of an isomorphic sub-graph. Henshin additionally provides a rich set of control flow primitives (such as sequencing, branching and looping), called transformation units, determining the order in which rules are applied.

On the other hand, the number and complexity of rules in the Henshin specification of a model transformation grows substantially with the size of the underlying meta-model. For instance, we had to construct many structurally similar rules for the cases in which an abstract meta-model class has many concrete sub-classes. As a result, we have decided not to fully implement model transformations in our prototype tool using Henshin.

9.5.2 Simulation Performance

In this section, we discuss some performance considerations of the simulator's prototype implementation. Although the primary goal of the simulator is to validate the behaviour of designed systems and its performance is thus of a secondary importance, we have made some observations useful for future implementation of the run-time environment for systems designed in our component model.

Firstly, the choice of the execution semantics of a component model influences the performance of systems designed in that model significantly. In our model, the data-driven execution semantics of some model elements has a positive impact on system performance since data-driven computation has a huge potential for parallelisation: once a data-driven entity has all its inputs – which is the minimum synchronisation requirement any computation can have – it can be executed. On the other hand, any synchronisation impacts system performance negatively, because it implies waiting. The control-driven and data-driven nature of our component model exhibits some inherent degree of synchronisation, which further increases due to the additional requirement of deterministic system behaviour and the realisation of execution as a series of execution cycles.

Another factor determining a system's performance is the quality of the implementation of the run-time environment itself. In implementing our prototype, we have made some decisions that trade off possible performance gains for simplicity. For instance, waiting for missing data inputs that are *computable within the current execution cycle* (see Section 6.5.5) before skipping triggering or disabling a component has instead been implemented by waiting for all data-driven computation to cease ($active = 0$). Although the implemented variant yields correct system behaviour[4], it may result in worse performance than waiting for inputs coming from entities comprising the production set of a given input port (a set of entities that can directly or indirectly provide a value to an input port, computed by the PRODUCTIONSET procedure from Section 6.5.6).

[4]A similar approach has been used in the formalisation of the execution semantics in Section 6.6

Chapter 10

Case Study: Mode Switching Pattern for Reactive Systems

In this chapter, we demonstrate how interaction patterns can be represented as composite connectors and how they can be used to develop systems in our component model.

We have chosen reactive control systems as our evaluation domain for its close fit with the idea of interaction patterns separate from the definition of computation. In particular, we focus on reactive control systems with modes – systems whose operation is at any one time driven by one of several strategies that can be switched between dynamically at run-time.

In Section 10.1, we define the Mode Switching interaction pattern, introduced in Section 3.4, as a connector template in our component model. In Section 10.2, we use the template to develop a particular reactive control system – climate control in a car.

10.1 Defining Mode Switching Pattern as a Composite Connector

In this section, we develop a connector template realising the Mode Switching interaction pattern described in Section 3.4. The template's interface will be based on the pattern's participants, while the pattern's behaviour will determine the template's composition structure.

10.1.1 Interface

Roles define entities that interact together in a pattern. In a sense, they define a pattern's *interface*. It is not surprising then that we use the roles of the Mode Switching interaction pattern to construct the interface of a connector template realising the pattern: for every role, we specify required data ports and control parameters through which the participants of that role interact with the connector template. To make the specified interface elements as general as possible (and thus maximise the reuse of the template), we use the parametrisation mechanisms defined in Chapter 8 – generic types, multiports and roles.

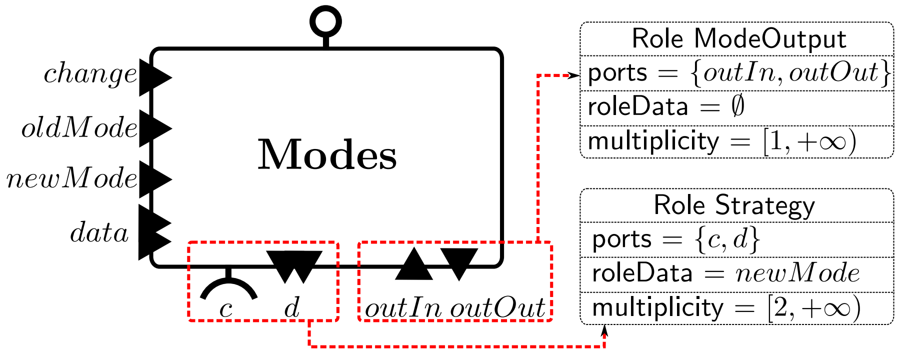


Figure 10.1: Interface of the Modes Connector Template

The interface of the connector template, called Modes, is depicted in Figure 10.1, and Table 10.1 details its elements. We structure its description according to the Mode Switching pattern's roles (see Section 3.4.3):

Role	**Interface Element**			
	Name	**Type**	**Data Type**	**Role**
Modes Provider	*change*	req. out.	boolean	–
	oldMode	req. out.	int	–
	newMode	req. out.	int	–
Inputs Provider	*data*	out. multi.	–	–
Mode	*c*	control par.	–	Strategy
	d	in. multi.	–	Strategy
	outIn	req. out.	–	ModeOutputs
Outputs Consumer	*outOut*	req. in.	–	ModeOutputs

Table 10.1: Interface Elements of the Modes Connector Template

Modes Provider

The Modes Provider participant interfaces with the connector through three required output ports: *change*, *oldMode* and *newMode*. The first one, of the boolean type, indicates whether the current mode has just been changed; the second and third ones, both of the integer type, input the old and the new modes, respectively. One can see that *change* is true iff $oldMode = newMode$; however, because composite connectors do not perform computation, which includes expression evaluation, our connector relies on an external component (the Modes Provider) to perform the evaluation instead. Likewise, the actual determining of the new mode is a function of the current mode and system inputs, and, as computation, it has been delegated to an external component, communicating through *newMode*.

Inputs Provider

The data inputs for the current Mode component are transferred through the required output multiport *data*. This allows us to parametrise the input interface of Mode components in the template, since particular data ports (and their types) will be specified during the instantiation of the template.

Mode

The interface for Mode components comprises three interface elements: the control parameter c, the required input multiport d and the required input port *outIn*. c and d are responsible for delivering inputs to a Mode component: a control signal for activation or deactivation through c and data inputs for the active Mode component via d. d delivers a set of data flows fed to the connector via the *data* multiport (i.e., *data* is the source multiport of d). Both c and d are aggregated in a role, called Strategy, and can thus be multiplied during the instantiation of Modes. The number of participants of the role should be set to the number of Mode components (which equals the number of system modes), so that each Mode component is connected to its own copy of c and d.

The required output port *outIn* collects outputs of the active Mode component. It is aggregated (together with *outOut*) in another role, called *ModeOutputs*. The number of the role's participants should match the number of output data ports of Mode components. Unlike Mode components' inputs, their outputs are collected by a set of required output ports shared by all Mode components. Because there is at most one active Mode component at a time, outputs' order cannot be destroyed by this non-deterministic configuration (several component output ports being connected to a single required port).

As a result, Mode components do not have to have the same interface: each Mode component can select a subset of data flows transported by d and output its results to a subset of the copies of $outIn$.[1] Additionally, Mode components can have other inputs and outputs, not dealt with by the Modes connector.

Outputs Consumer

The required input port *outOut* forms the interface for component(s) collecting outputs of the active Mode component. As mentioned above, it is associated with the ModeOutputs role and will be therefore copied during instantiation. The number of copies will be equal to the size of the set union of all output data ports of all Modes components. The data types of the copies of the port have to equal the data types of the ports connected to the corresponding $outIn$ port, which is expressed by the template's additional constraint $sameTypeAs(outIn, outOut)$ (see Section 8.2.1 for the definition of the *sameTypeAs* association).

10.1.2 Composition Structure

The composition structure of the Modes connector template realises the Mode Switching pattern's behaviour (Section 3.4). The composition structure of Modes is hierarchical since the template reuses the Trigger-Strategy connector template (Figure 8.5). Trigger-Strategy can be reused since the Modes 's behaviour dealing with redirecting inputs and sending control signals to Mode components bears similarity with the behaviour of Trigger-Strategy. However, Trigger-Strategy cannot be reused as is, it needs to be adapted to suit the context of the Modes connector template. We adapt it by composition within another connector template, called Activation-Strategy, which is in turn composed within the Modes connector template.

Figure 10.2 shows the composition hierarchy of the Modes connector template. In our description, we proceed in a top-down manner, starting with the composition structure of the Modes connector template.

[1]Notice that, if we used multiports for collecting Mode components' outputs, they could not be shared by all Mode components but would have to be copied for each of them, making the connector's interface more complex.

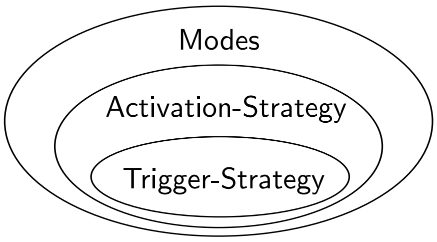


Figure 10.2: The Composition Hierarchy of the Modes Connector Template

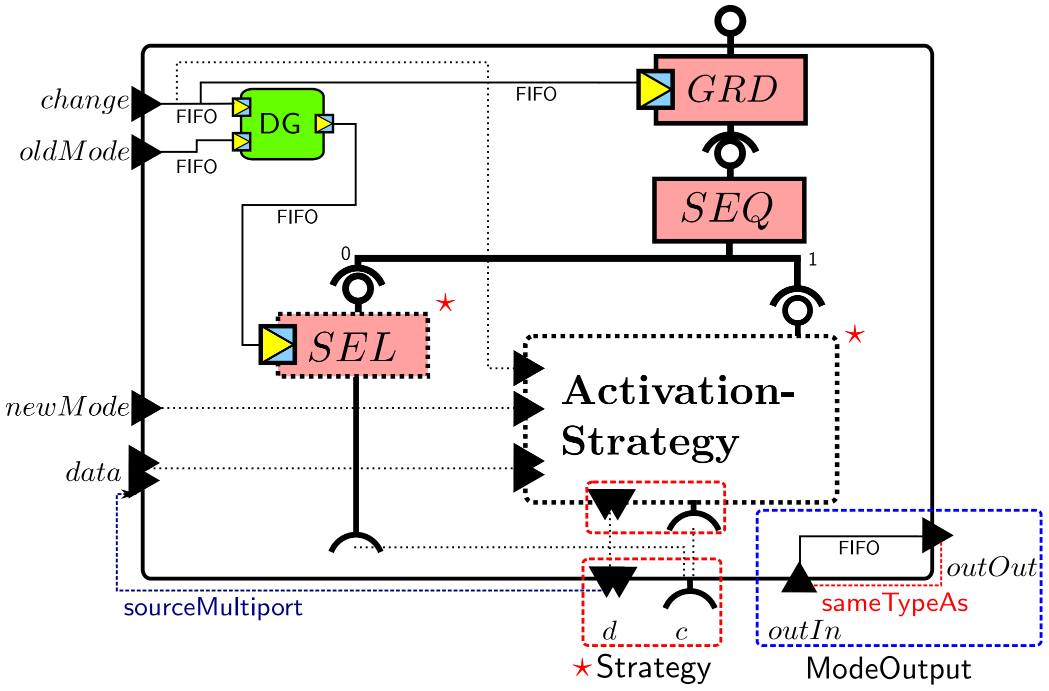


Figure 10.3: The Modes Connector Template

The Modes Connector Template

The control flow part of the Modes connector template (Figure 10.3) switches the currently active Mode component only if the current mode changes. That is realised by the guard control connector at the top of the template's control hierarchy, connected to the *change* required port. If the mode change occurs, the sequencer carries out the two activities comprising mode switching: firstly, the selector connector chooses the Mode component corresponding to the old Mode to be disabled; secondly, the Mode component corresponding to the new mode is enabled.

The selector is instantiated as a design instance, associated with the Strategy role (indicated by the star symbol). Consequently, its arity will be fixed at the template's instantiation time and set to the number of participants of the Strategy role, which equals the number of Mode components composed by the Modes connector.

Enabling of the new Mode component is delegated to the design instance of Activation-Strategy. As the selector, the Activation-Strategy instance is associated with the Strategy role. Its two interface elements associated with the role, delegated to the interface level of the Modes template, are thus copied for each Mode component. The control parameter c sends control to enable the Mode component corresponding to the new mode.

FIFO data channels and data coordinators route the incoming decision data (*change*, $oldMode$ and $newMode$) to the control and data routing connectors. Although the primary concern is to deliver the decision data to their targets, it is also important to decide when the data should not be delivered. For instance, the data guard ensures that the design selector only receives its data input if it is going to be triggered by the sequencer (i.e., when *change* = *true*), and thus prevents the selector from reading an old $oldMode$ value that would have otherwise been accumulated in the adjacent FIFO data channel during the execution cycles in which *change* = $false$.

The Active-Strategy instance routes data flows – aggregated by multiports – from the required output multiport *data* to the required input multiport d of the active Mode component. The outputs of the active Mode component are collected by the FIFO data channel transferring data from $outIn$ to *outOut*. To transfer each output of the Mode component, the channel will be duplicated during instantiation as many times as the ports it connects.

The Activation-Strategy Connector Template

This template adapts Trigger-Strategy to the context of the Modes connector. Trigger-Strategy coordinates a set of control-driven components: in every cycle, it triggers one of them (selected by the value of the *strategy* required port) and routes the data flows aggregated by the *data* multiport to the selected component. In the Modes connector template, Activation-Strategy is responsible for sending the control signal to a control-switched data-driven component only if the mode change occurs. However, since this is ensured by the top-level guard connector in Modes, the control flow structure of Trigger-Strategy (Figure 8.5) can be reused unchanged. Likewise, the routing of incoming data flows from *data* to d (of the component selected by *strategy*) suits the aim of delivering inputs to the active Mode component in Modes. As a result, the interface elements of Trigger-Strategy responsible for this routing behaviour (c, d and *data*) and the associated Strategy role are directly delegated to the interface level of Activation-Strategy (see Figure 10.4 and Table 10.2).

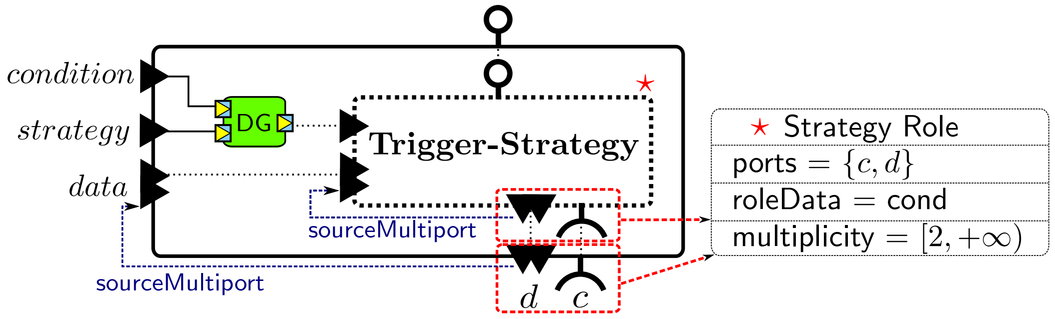


Figure 10.4: The Activation-Strategy Connector Template

Name	Type	Data Type	Role
condition	required output port	boolean	–
strategy	required output port	int	–
data	required output multiport	–	–
c	control parameter	–	Strategy
d	required input multiport	–	Strategy

Table 10.2: Interface Elements of the Activation-Strategy Connector Template

Trigger-Strategy is designed so that the decision data determining the selected component should be passed to the required port *strategy* as many times as the selected control-driven component is being triggered (typically, once per execution cycle). This constraint must be fulfilled by every context into which Trigger-Strategy is deployed. However, if we deployed Trigger-Strategy directly into the context of the Modes connector template, the constraint would be violated: Trigger-Strategy would receive control signal only if the mode changes, but the strategy selection data would come more frequently, even if the mode had not changed. This would result in the *strategy* selection values being accumulated in the buffer of the FIFO data channel within the Trigger-Strategy instance, and it would lead to choosing the newly active Mode component based on old decision data. To prevent this discrepancy in frequencies of incoming control signals and the corresponding decision data, the data guard in Activation-Strategy only lets the decision data enter the Trigger-Strategy instance if the mode has changed ($condition = true$).

On the other hand, the aggregated data flow from *data* must be delivered to the active Mode component at all times, regardless of the coming control signal. Consequently, the direct delegation between *data* and its counterpart in Trigger-Strategy is correct. Note that deploying Conditional Trigger-Strategy (Figure 8.6) instead of Activation-Strategy within Modes would be incorrect for the same reason.

10.2 Climate Control System

In this section, we develop an example system to illustrate using the Modes connector in a system development. The example system is a climate control system for cars, which we adapted from the case study realised by Labbani et al. [69]

10.2.1 System Requirements

The climate control system is a reactive software system controlling the climate in a car. It regulates the temperature in a car by controlling the speed of a ventilation fan and the input power (and indirectly the temperature) of a car heater. The user operates the system through the control panel shown in Figure 10.5. The panel contains three buttons (*Mode*, *Up* and *Down*) and two displays showing two variables controlled by the system – the current value of the heater's input power (in %) and the current speed of the ventilation fan (in rotations per minute).

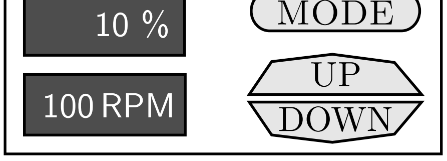


Figure 10.5: The Climate Controller Device

Apart from the inputs from the control panel, the system also senses the current temperature inside the car. Figure 10.6 gives an overview of all inputs and outputs of the system. The system takes four inputs: three booleans for the control panel's buttons (`true` when a corresponding button is pressed, `false` otherwise), and an integer representing the actual temperature in the car measured by a sensor. The system outputs two values: heater input power and ventilation fan speed. The system is sent inputs periodically and produces outputs in response, with the same frequency.

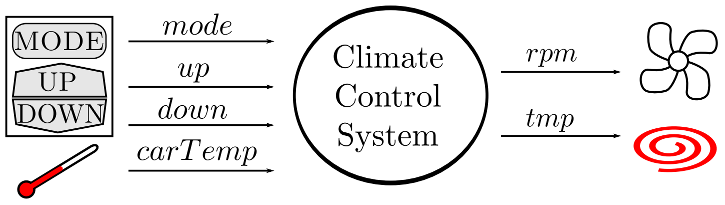


Figure 10.6: Inputs and Outputs of the Climate Control System

The system operates in one of three modes: Auto-Ventilation, Manual Temperature and Manual Ventilation. Once started, the system enters the Auto-Ventilation mode. The user changes modes cyclically in the order given by Figure 10.7 by pressing the Mode button. In each mode, the system's strategy for computing outputs and interpretation of inputs is different.

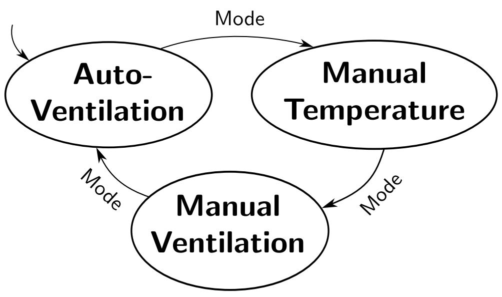


Figure 10.7: Modes of the Climate Control System

In the Manual Temperature mode, the system sets the input power of the heater according to the value set by the user via the control panel (the Up and Down buttons increase or decrease the set value by 1%, respectively) and keeps it regardless of the real temperature in the car. In the Manual Ventilation mode, the system only sets the ventilation fan's speed (set by the user in rpm using the Up and Down buttons). In the Auto-Ventilation mode, the system adjusts both controlled variables automatically to achieve and maintain the user-set temperature (set using the Up and Down buttons) inside the car – the higher the difference between the set and actual temperatures, the higher the heater's input power and fan speed. The functionality of the system's modes is summarised in

Table 10.3.

Mode	**User sets**		**System controls**	
	Variable	**Range**	**Heater**	**Fan**
Auto-Ventilation	cabin temperature in °C	$[10, 30]$	✓	✓
Manual Temperature	heater input power in %	$[0, 100]$	✓	×
Manual Ventilation	fan speed in rpm	$[0, 1000]$	×	✓

Table 10.3: Functionality of the Climate Control System's Modes

10.2.2 System Design

From the system requirements given in Section 10.2.1, it is clear that the climate control system is a reactive control system with multiple modes. We can therefore apply the Mode Switching interaction pattern to separate its mode switching logic from the computation carried out in different modes. In our component model, this amounts to deploying the Modes connector template into the system's architecture.

The Modes connector creates the basic coordination structure of the climate control system, into which we plug components that correspond to participants of the Mode Switching interaction pattern (Section 3.4). To design the system architecture, we first design for each of the pattern's participants their corresponding representation in our component model. We then instantiate the Modes connector template, connect it with the entities corresponding to participants and finalise the system architecture.

We define the following components to act as the Mode Switching pattern's participants:

Inputs Provider A component that feeds the Modes connector (and the whole system) with inputs can be represented in our component model as a source component. We therefore create a new source component, named SOURCE, with four output ports *mode*, *up*, *down* and *carTemp*, which correspond to system inputs (see Table 10.4 for the interface details of the components designed in this section).

Outputs Consumer Likewise, our component model provides sink components for collection of system outputs. We create two sink components – TMP

Component	**Port Name**	**Port Type**	**Data Type**
SOURCE	*mode*	output	boolean
	down	output	boolean
	up	output	boolean
	carTemp	output	integer
ModeChanger	*change*	input	boolean
	currentMode	input	integer
	outNewMode	output	integer
	oldMode	output	integer
AutoVentilation	*up*	input	boolean
	down	input	boolean
	carTemperature	input	integer
	tIn	input	integer
	tOut	output	integer
	rpm	output	integer
	tmp	output	integer
ManualTemperature ManualVentilation	*up*	input	boolean
	down	input	boolean
	value	input	integer
	output	output	integer
TMP	*tmp*	input	integer
RPM	*rpm*	input	integer

Table 10.4: Component Interfaces in the Climate Control System

and RPM – to represent the Outputs Consumer participant, because the system does not produce all its outputs in each mode. One sink component for both outputs would only consume them if both were available, which might result in some missing outputs at the end of system execution.

Modes Provider The component realising the Modes Provider participant provides the new and previous modes to the Modes connector. We implement it as the data-driven component ModeChanger. It takes the current mode and the information whether that mode is going to change (coming from the Mode button), and it computes the two modes. The component (see Figure 10.8) basically implements the state transition diagram in Figure 10.7, which defines the modes of the climate control system and transitions between the modes. ModeChanger is a state-less component, and will later require the addition of a buffered data channel to keep the current mode information.

```
component ManualControl {
  in up: boolean,
  in down: boolean,
  in value: int,
  out output: int
  {
    if (up) output = value + 1
    else if (down) output = value − 1
    else output = value
  }
}

component ModeChanger {
  in change: boolean,
  in currentMode: int,
  out oldMode: int,
  out outNewMode: int
  {
    outNewMode = if (change) {
          (currentMode + 1) % 3
      } else {
          currentMode
      }
    oldMode = currentMode
  }
}
```

```
component AutoVentilation {
  in up: boolean,
  in down: boolean,
  in carTemperature: int,
  in tIn: int,
  out rpm: int,
  out tOut: int,
  out tmp: int
  {
    if (up) tOut = tIn + 1
    else if (down) tOut = tIn − 1
    else tOut = tIn

    if (carTemperature < tOut) {
      rpm = 1000
      tmp = 100
    } else {
      rpm = 0
      tmp = 0
    }
  }
}
```

Figure 10.8: Definition of Components in the Climate Control System

Mode The system functionality in each mode is realised as a control-switched data-driven component of the same name. The AutoVentilation component is the most complex one: firstly, it keeps the user-set cabin temperature and updates it when the Up or Down button is pressed; secondly, it computes the heater input power *tmp* and fan speed *rpm* from the user-set temperature and the current cabin temperature *carTemperature*. The other two modes are realised by two instances (ManualTemperature and ManualVentilation) of the ManualControl component. They compute the new value of the variable they control, based on its old value and the state of the Up and Down buttons. All components are implemented as stateless (see Figure 10.8), i.e., without internal variables, and thus will have to have NDR-1 data channels added to some of their ports to keep the values of their controlled variables.

To instantiate the Modes connector template, we need to specify its instantiation information. It comprises the specification of the participants of the roles ModeOutput and Strategy, defined by the pattern. The ModeOutput role should be instantiated for every output data flow produced by Mode components. In the climate control system, the Mode component produces two such outputs, the heater temperature and fan speed. Consequently, we can instantiate this role using two participants; because the role does not define any role data, the instantiation information for this role is complete. The Strategy role is instantiated using three participants: one for each system mode. Figure 10.9 shows the Mode connector instance as well as all participants. Some of the required ports of the instance, such as *outIn* and *outOut*, have been renamed to suit the context of the climate control system.

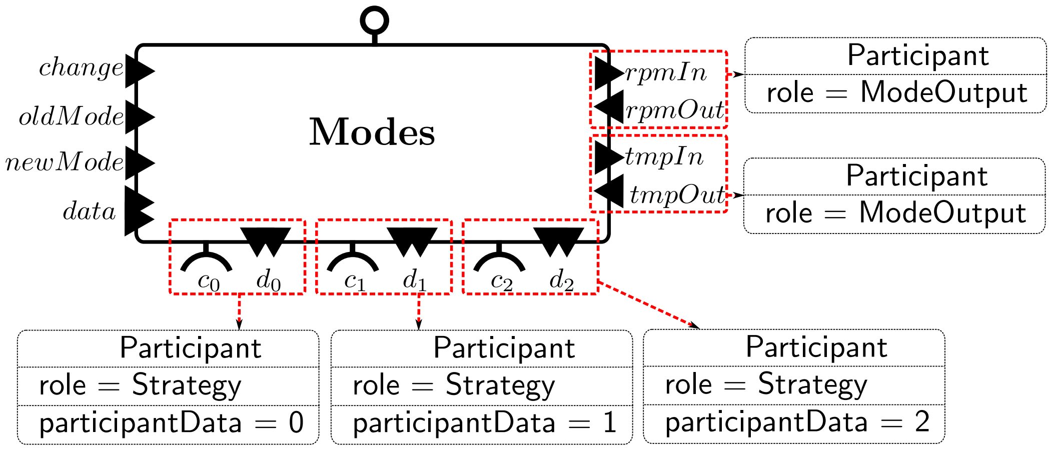


Figure 10.9: The Modes Connector Instance in the Climate Control System

The next step is to connect the interface elements of the Modes connector instance to the data ports of the components we have defined earlier. Here, the only interesting aspect is choosing which data ports should be aggregated by the *data* multiport. The general guidance is that it should be the data ports providing inputs to the Mode components. In the climate control system, these are at least the *up* and *down* ports of SOURCE, which are inputs to all three Mode components. However, we might also aggregate the data flow coming from *carTemp*, although only AutoVentilation consumes it. Ultimately, the choice is the responsibility of the designer.

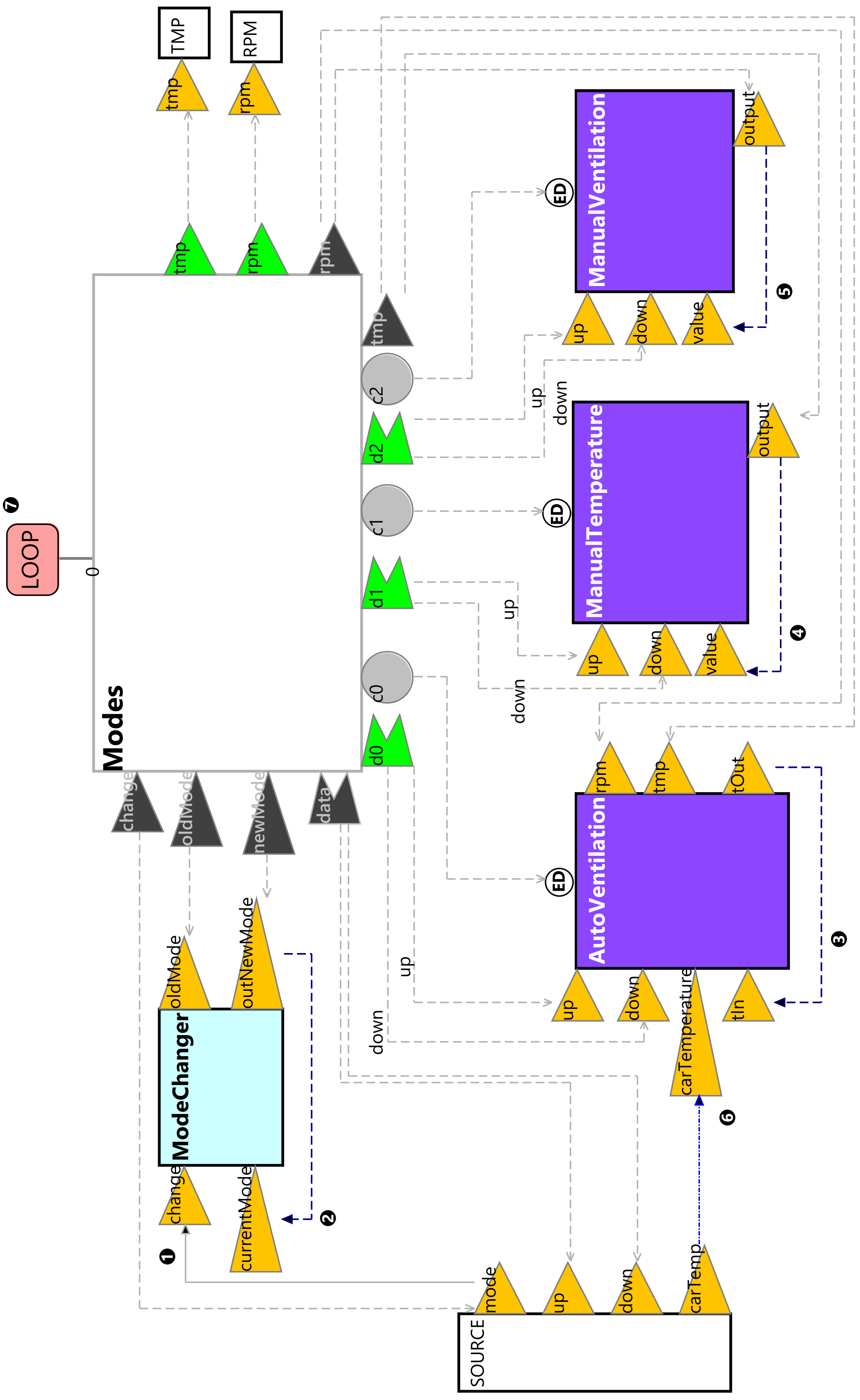


Figure 10.10: Architecture of the Climate Control System

Finally, the climate control system architecture (Figure 10.10 shows its graphical representation exported from our prototype tool) can be completed by adding the remaining data and control connectors. For the reader's convenience, the additions are numbered from ① to ⑦ and are tagged in Figure 10.10. Note that the graphical syntax in the tool slightly differs from the symbols introduced for component model elements earlier in the thesis; Figure 10.11 shows the meaning of individual symbols used in Figure 10.10.

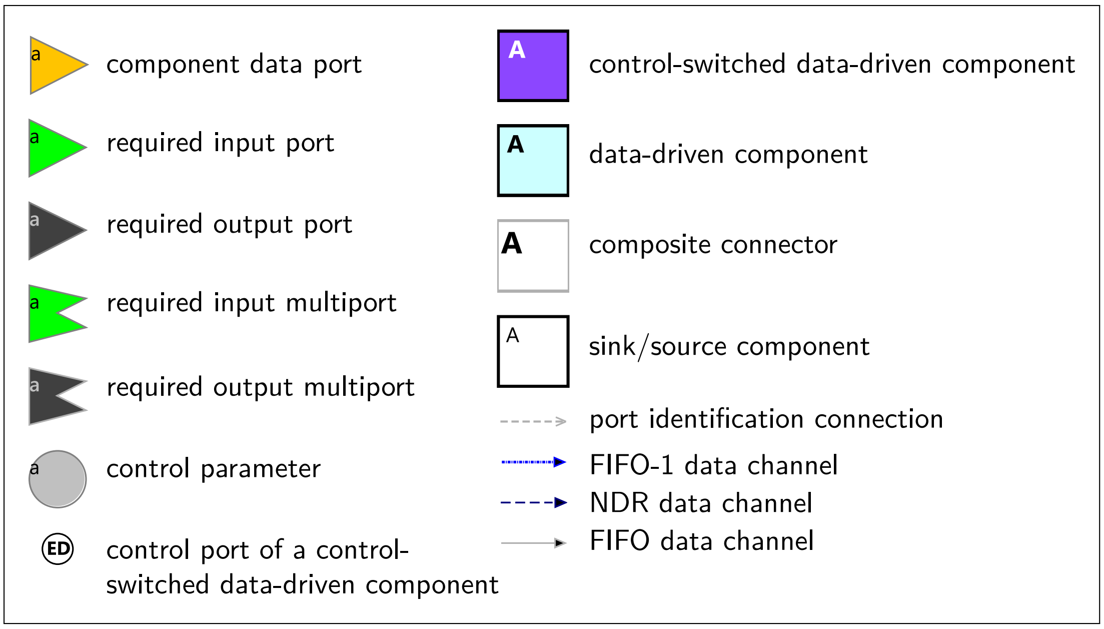


Figure 10.11: Key to the Graphical Syntax of the Prototype Tool

ModeChanger needs to be fed its inputs by data channels: a FIFO for the *change* input port ①, and an NDR-1 channel for the *currentMode* input port ②. The NDR-1 channel realises a common design pattern in our model. It connects a component's output port to its input port, and thus stores the result of the last component's execution to be fed as an input to the next component execution – it effectively adds state to a state-less component. The channel has to be initialised, so that there is a valid initial state. Thus, channel ② has the default value of 0, denoting the Auto-Ventilation mode. Likewise, NDR-1 data channels ③, ④ and ⑤ follow the same pattern: channel ③ keeps the internal car temperature, channel ④ keeps the heater's input power and channel ⑤ keeps the ventilation fan speed.

AutoVentilation gets the current car temperature from SOURCE via the FIFO-1 channel ⑥. The channel with capacity one was chosen so that AutoVentilation always processes the most up-to-date value coming from the source (old values cannot accumulate in the buffer when the system operates in one of the other two modes).

Lastly, the Loop control connector ⑦ has been added to the top of the control connector hierarchy and connected to the Modes connector instance to realise the system's looping behaviour. To complete the architecture description of the climate control system, Table 10.5 summarises the initial state of the stateful component model entities comprising the system.

Entity	**State**	**Value**
NDR-1 channel ②	Initial mode	0
NDR-1 channel ③	Initial car temperature	20
NDR-1 channel ④	Initial heater temperature	0
NDR-1 channel ⑤	Initial fan speed	0
AutoVentilation	Execution State	Enabled
ManualTemperature	Execution State	Disabled
ManualVentilation	Execution State	Disabled

Table 10.5: Initial States of the Climate Control System's Entities

10.2.3 System Validation

To validate the behaviour of a system designed in our prototype tool, users can compare the expected system behaviour to the behaviour simulated according to the execution semantics of our component model. In this section, we test the behaviour of the climate control system, constructed in the previous section, against the expected behaviour of a particular test scenario: we create the scenario, prepare the necessary inputs to the simulator and compare the simulated results for the scenario with the expected ones.

To test the climate control system's behaviour thoroughly, we design the test scenario that covers the functionality of the system in all its modes. Essentially, we need to list system inputs and corresponding system outputs for several reactive cycles. The inputs comprise the state of the Up, Down and Mode buttons and readings from the sensor measuring the temperature of the car's cabin; the outputs are the heater's input power and ventilation fan speed.

Inputs				**Internal States**				**Outputs**	
mode	*up*	*down*	*carTemp*	*newMode*	t_1	t_2	*s*	*rpm*	*tmp*
False	False	**True**	10	AV	19	0	0	1000	100
False	False	**True**	11	AV	18	0	0	1000	100
True	False	False	12	MT	18	0	0	–	0
False	**True**	False	13	MT	18	1	0	–	1
True	False	False	14	MV	18	1	0	0	–
False	**True**	False	15	MV	18	1	1	1	–
False	**True**	False	16	MV	18	1	2	2	–
True	False	False	17	AV	18	1	2	1000	100
False	False	False	18	AV	18	1	2	0	0
False	False	False	19	AV	18	1	2	0	0

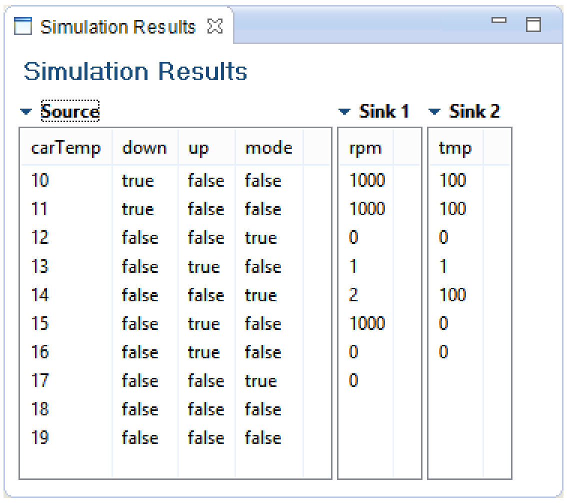
Table 10.6: A Test Scenario for the Climate Control System

In this test scenario, we model the following sequence of button presses (one per cycle): *Down* → *Down* → *Mode* → *Up* → *Mode* → *Up* → *Up* → *Mode*, and increasing values of the sensor measurements (starting at 10°C). Additionally, we include input data for two more cycles in which no buttons are pressed but the internal temperature changes. Table 10.6 shows the expected outputs of the climate control system for these inputs together with some internal states to help explain the output values. Each row corresponds to one execution cycle. The values in the Inputs column are data produced by SOURCE (hence the names). The Internal States column contains the current mode *newMode*, the user-set temperature of the car's cabin t_1, the user-set input power to the heater t_2 (in %) and the user-set fan speed s. The values in the Outputs column show the expected system outputs.

The simulator simulates the behaviour of a given system, based on the system's architecture and simulator configuration. The configuration comprises (i) the initial states of stateful entities from the system's architecture, (ii) the association of source and sink components with the files that the simulator uses for feeding inputs to and collecting outputs from the system, and (iii) the contents of the input files themselves.

The simulator produces three kinds of outputs: (i) it saves system outputs in the output files specified in the simulator configuration, (ii) it produces system execution trace for debugging purposes, and (iii) it displays a graphical window summarising the results of the simulation in terms of inputs and outputs.

Having created the input files corresponding to the test scenario defined in Table 10.6, we executed the simulation and got the summary with simulation results shown in Figure 10.12. Comparing the simulation results with the expected results (Table 10.6), we see that the system has produced the expected results, and we can therefore conclude that the system behaves correctly in this particular test scenario.



carTemp	down	up	mode	rpm	tmp
10	true	false	false	1000	100
11	true	false	false	1000	100
12	false	false	true	0	0
13	false	true	false	1	1
14	false	false	true	2	100
15	false	true	false	1000	0
16	false	true	false	0	0
17	false	false	true	0	
18	false	false	false		
19	false	false	false		

Figure 10.12: Outputs of the Simulation

Chapter 11

Discussion

In this chapter, we compare our approach with several strands of related approaches, and we give the overall evaluation of the conducted research against the research goals stated in Chapter 1.

In Section 11.1, we evaluate the benefits and drawbacks of the connector representation of interaction patterns in our approach with their component representations in current component models. In particular, we focus on component-based approaches in the reactive control system domain, the domain of our case study.

Composite connectors in our component model represent certain coordination behaviour; they thus form a coordination language. In Section 11.2, we compare composite connectors with relevant coordination languages to establish their unique characteristics.

Our approach aims to make some design patterns (interaction patterns) first-class artefacts in design and implementation, with a first-class architectural representation and well-defined behaviour. Section 11.3 analyses a number of existing approaches in a wider context of software engineering that also formalise design pattern solutions.

Finally, Section 11.4 gives an overall evaluation of the research conducted in this thesis with respect to our initial research goals. We also discuss the limitations of our approach.

11.1 Interaction Patterns in Component Models

In this section, we assess the benefits and drawbacks of the connector representation of interaction patterns in our approach by comparison with alternative interaction pattern representations in other component models.

To evaluate the selected component models, we have designed a system architecture for our case study, the climate control system (see Section 10.2), in each component model and examined how the Mode Switching interaction pattern can be represented there. The resulting architectures are shown in Appendix D.

Due to the focus of our case study, we have mostly selected component models used in the domain of control systems. In particular, we have chosen ProCom, Scade, Simulink and UML 2.0. UML 2.0 is not domain-specific but it has been included to represent a large group of control-driven component models with operation-based component interfaces using the request-response interaction style. Despite the limited scope, the comparison still covers various component model characteristics, as identified in the initial survey of existing component models in Section 5.3.1.[1]

In Sections 11.1.1-11.1.4, we discuss the interaction pattern representation in the compared component models. Section 11.1.5 summarises our findings.

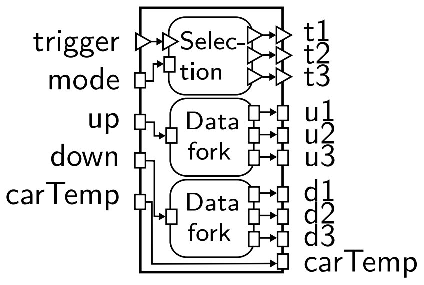
11.1.1 ProCom

ProCom (see Section 4.6.2) has connectors that can model the routing behaviour of interaction patterns in terms of control flow and data flow as our model. However, because ProCom lacks support for composite connectors, interaction-defining connections and connectors have to be composed within composite components together with other subcomponents (as Figure D.1 shows). Therefore, interaction patterns cannot be represented in ProCom as first-class entities distinct from components.

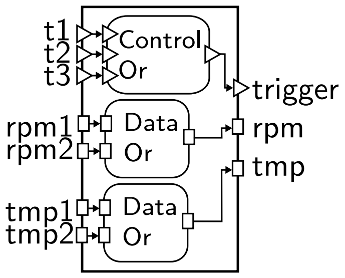
In some cases, the behaviour of an interaction pattern can be represented by a composite component comprised of connectors only [25]. For example, Figure 11.1 shows two such composite components. In our case study, the component in Figure 11.1a could trigger the component corresponding to the current mode and

[1]Component-based approaches based on the exogenous coordination mechanism are discussed in Section 11.2.

route input data to Mode components; the component in Figure 11.1b could collect Mode components' outputs. The two components could be further combined to form one composite component by being exposed as independent services of the composite.



(a) Distributing Control and Data Inputs

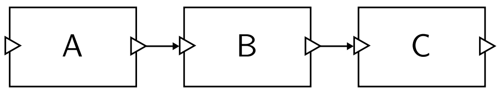


(b) Collecting Outputs

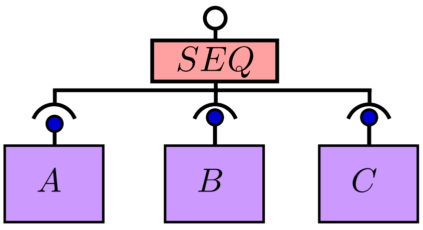
Figure 11.1: ProCom Components Encapsulating Connectors Only

However, in general it is not possible to aggregate any interaction pattern within a ProCom composite component. This is caused (i) by ProSave's inability to compose control sequencing behaviour, and (ii) by the strict execution semantics of ProSave components.

ProSave expresses control sequencing by means of its control connection. If two components are to be triggered sequentially, the output trigger port of the first component must be connected to the input trigger port of the second component. Sequencing of three components has to be expressed by two connections, as shown in Figure 11.2a. This makes it impossible to fully encapsulate sequencing within a composite component – the coordinated components would always have to have their output trigger port connected to the right port of the coordinating composite component. By contrast, control sequencing in our model can be fully encapsulated within a composite connector by means of a sequencer (Figure 11.2b).



(a) ProCom



(b) Our Model

Figure 11.2: Comparison of Control Sequencing

Services of ProSave components execute atomically, according to the read-execute-write semantics. Once a service's input trigger port receives a control signal, the service's input data ports are read, the service executes and, once the execution has finished, the results are written to output data ports and a control signal appears at the service's output trigger port. Such semantics is unsuitable for interaction patterns that involve waiting of some kind (e.g., an interaction pattern representing a dialogue, a two-way exchange of data between two components) – atomic execution of ProSave component rules out any waiting during a service invocation.

Consequently, some interaction patterns have to be represented in ProCom as composite components that mix coordination (connectors and connections) with computation (subcomponents), which negatively impacts their reuse.

11.1.2 Scade

Because Scade (see Section 4.6.3) has no connectors, interaction patterns have to be represented as components. Interaction patterns can be represented either using Scade's state machines or by combination of data connections and data routing components (see Appendix D.2 for examples of both).

State machines in Scade are syntactic sugar: their behaviour can be defined by several if-else statements or by manipulating clocks of their states' output flows [34]. Their observable behaviour therefore corresponds to that of an ordinary Scade component. They are often used in top-down hierarchical system design, in which states are identified first and refined later.

A state machine can represent an interaction pattern's instance: its states represent computations and state transitions represent the control flow aspect of interaction (the data flow aspect is represented by data connections). Such a representation of patterns precludes separate reuse of patterns' interaction parts, because they cannot be separated from the definition of computation in states and thus cannot exist as stand-alone entities.

However, Scade can separate the definition of interaction and computation since it is, like ProCom, based on the pipe-and-filter interaction style. The benefits of such separation for Scade designs have been noted by Labbani et al. [69]

In this approach, interaction patterns are represented by data connections and data routing components. Still, patterns cannot be reused, because their elements cannot be composed into entities that could exist and be reused in their own right.

Furthermore, because control flow cannot be modelled explicitly, Scade lacks the expressiveness required for rich interaction modelling.

11.1.3 Simulink

The Simulink (see Section 4.6.4) representation of interaction patterns is similar to that of Scade: an interaction pattern corresponds to a number of data connections and data-routing Simulink blocks. However, Simulink can express more complex interactions than Scade due to its richer palette of modelling constructs. For example, it can mimic control flow by means of so-called triggered and enabled subsystems (Appendix D.3 gives some examples of these).

Although Simulink's features that mimic control flow increase the expressiveness of interaction pattern modelling, it still cannot be considered as full-fledged control flow modelling support. Semantically, all inter-component connections represent data flow. The fact that some blocks interpret some inputs as 'control signals', which enable or trigger outputs' production, does not prevent other blocks from interpreting the same signal as data. Simulink has no special type for control flow. In fact, the data type specification of data flows in Simulink is optional. On the one hand, this gives certain flexibility to Simulink designs; on the other hand, run-time type errors may occur.

Additionally, Simulink has some features that simplify data flow modelling. It supports vector signals, which comprise the tuples of values of the same scalar type. Signals can be also composed using special blocks (called bus creators and muxes) into composite signals to simplify the graphical data flow models (so that one line represents multiple data flows) or to create hierarchical data structures. Composite signals have to be decomposed to the constituent primitive signals before they can be connected to a block.[2]

11.1.4 UML 2.0

Since the UML 2.0's connector set is fixed and not composable, interaction patterns can only be represented as components.

UML 2.0 lacks the ability to model interaction in terms of control flow and data flow directly in the architecture diagram. The assembly connectors only

[2]Several selected Simulink blocks, such as Memory, are exempt from this rule.

signify the fact that some of the services from the provided interface of one component (callee) are used to implement some services from the provided interface of the other component (caller) via the connected required port. However, the interaction is fully defined within the code of the caller component, never explicitly in the architecture diagram. The only information related to the inter-component communication, apart from implementation, is the interface signatures associated with component ports, which partially specify data flow in terms of data parameters and return values of all the interface services. Nevertheless, this does not define data flow fully as it cannot be discerned which services from the interface specification are actually called and in what order. Control flow definition is missing completely.

Because the exact behaviour of an interaction pattern is defined in the implementation of a component that coordinates other components, the component pattern representation has dependencies on particular interfaces of coordinated components. In general, this may negatively impact reuse, since it is not just the data types of incoming and outgoing data but also the names of operations, the order of their parameters and interface names, which must match in order for the pattern to be reusable.

11.1.5 Summary

None of the compared component models is able to represent interaction patterns as first-class component model entities different from components. Consequently, interaction patterns need to be represented as components, indistinguishable from any other computation in system architecture. In this section, we summarise our findings on component representations of interaction patterns among the compared component models. In particularly, we focus on their expressiveness and reuse potential.

Expressiveness of Component Representations of Interaction Patterns

The expressiveness of a particular representation of interaction patterns depends on (i) whether it supports explicit modelling of control flow and data flow, (ii) the execution semantics of the underlying component model, and (iii) whether the representation is compositional in nature. Table 11.1 summarises the relevant characteristics of the compared component models.

Component Model	**Control Flow**	**Data Flow**	**Execution Semantics**	**Compositional Representation**
ProCom	✓	✓	control-driven	×
Scade	×	✓	data-driven	✓
Simulink	partially	✓	data-driven	✓
UML 2.0	×	×	control-driven	×
Our approach	✓	✓	control-driven and data-driven	✓

Table 11.1: Expressiveness of Interaction Pattern Representations

The ability to model control flow and data flow explicitly increases the expressiveness of an interaction pattern representation. In UML 2.0, which lacks explicit control flow and data flow modelling, interaction patterns end up being absent from the system architecture, largely defined in the implementation of atomic components. Although both Simulink and Scade do not model control flow, Simulink has more expressive power compared to Scade, because some of its blocks (e.g., enabled and triggered subsystems) mimic control flow modelling. ProCom and our approach model both flows explicitly.

The execution semantics of a component model also affects the expressiveness of its interaction pattern representation. Data-driven models, such as Scade, cannot model control flow explicitly. A more subtle example of this is ProCom, which is control-driven but models both flows; however, since the only component execution mechanism is triggering, it cannot express more complex interaction patterns involving disabling/enabling components as faithfully as our model. Likewise, although some Simulink blocks have a 'control port', control flow does not have a first-class modelling status in Simulink: it cannot be distinguished in architecture from data flow, and blocks can freely interpret 'control values' as data and can read and change the data without any restrictions. That is, Simulink is not expressive enough because of its data flow only nature. In general, the richer execution semantics of a component model (such as the hybrid control- and data-driven semantics in our model), the more expressive pattern representation the model supports.

Another important characteristic of an interaction pattern representation is whether it is compositional. Compositional representations, in which complex patterns are defined as composition of simpler patterns, allow us to define patterns explicitly in software architecture. Furthermore, compositional pattern representations help to tackle the complexity of pattern definitions by making them hierarchical. Ultimately, compositional patterns are decomposed into basic building blocks that are simple, and thus easy to understand, pieces of behaviour defined implicitly at the component model level.

In our comparison, Scade and Simulink allow interaction patterns to be represented compositionally (albeit as components), as does our model. ProCom's representation is not fully compositional, because its atomic components, which can be part of a composite component representing an interaction pattern, are defined in code; on the other hand, its connectors can be composed to form more complex flows. Interaction patterns in UML 2.0 are entirely defined in the implementation of atomic components since composite components delegate their functionality to subcomponents; consequently, interaction patterns are hidden from architecture.

Reusability of Component Representations of Interaction Patterns

Reuse of most component representations of interaction patterns is limited, compared to our approach. ProCom has to represent some interaction patterns as composite components with subcomponents that define specific (and thus non-reusable) participant behaviour. Likewise, patterns represented as Scade state machines cannot be reused, because all components defining participant behaviour are inseparable from the state machine definition. UML 2.0 components representing patterns have unnecessary dependencies on the exact interface signature (names of operations, interfaces and the order of operation parameters), compared to port-based approaches (including ours).

Apart from dependencies, the variability of pattern representations is another factor that impacts their reuse potential. In general, component representations offer less variability than our composite connectors, thereby lowering reusability.

To assess the variability of different interaction pattern representations, we examine the following variability aspects: (i) independence of interface definitions on specific types, (ii) the ability to abstract multiple data flows into one in the pattern definition, and (iii) support for structural variability. Our approach addresses all of them (see Section 8.2): the sameTypeAs relation abstracts from specific types, multiports allow aggregating multiple data flows, and the mechanism of roles gives us some structural variability.

Component Model	**Generic Types**	**Flow Aggregation**	**Variable Structure**
ProCom	×	×	×
Scade	✓	✓	×
Simulink	✓	✓	✓
UML 2.0	✓	✓	×
Our approach	✓	✓	✓

Table 11.2: Variability of Connector Representations

Table 11.2 summarises the variability of connector representations in all compared component models. ProCom components lack any variability support. Most models allow component interfaces to be abstracted from specific types: UML 2.0 operations can contain type parameters, Scade supports so-called polymorphic nodes that use type parameters in their interface definitions, and Simulink subsystem ports can inherit their type from the subsystem's parameter or from the type of connected data flow.

In our approach, the type system only defines primitive data types, and we use multiports to abstract multiple data flows of primitive values into an aggregate data flow. Alternatively, a type system with composite data types and type parameters that can be set to these types achieves a similar result. UML 2.0, Scade and Simulink take this alternative approach. UML 2.0 can define composite data types (e.g., classes containing several attributes of primitive types), Scade and Simulink support arrays and structured types (called buses in Simulink).

Finally, the only model – except for our model – supporting any form of structural variability in our comparison is Simulink. It features a special subsystem, called Variant Subsystem, whose interface is associated with multiple implementations. It also contains a mechanism that maps combination of values of certain model parameters (called control variables) to these implementations. The behaviour of a variant subsystem is fixed before simulation: only the variant subsystem implementation that corresponds to the values of control variables in that system is used for simulation. Unlike our approach, each variant implementation has to be created manually. Therefore, our approach scales better if pattern variants can be defined using the role mechanism (and thus generated automatically).

11.2 Exogenous Coordination Languages

Composite connectors in our model effectively define a coordination language that controls the execution of computation encapsulated in components. It therefore seems meaningful to compare our approach with other coordination languages used in component-based software development.

Coordination languages are based on the distinction of computation and coordination as two orthogonal aspects of system behaviour [47] (see Section 3.1). They can be classified into endogenous and exogenous [4] (see Section 4.4.4). In this section, we only discuss exogenous coordination languages, which allow the existence of separate coordinating entities and are thus able to represent interaction patterns.

Coordination languages are further categorised as data-oriented or control-oriented [93]. Data-oriented approaches coordinate entities by means of shared data space or by controlling their mutual data exchanges (these are sometimes called dataflow-oriented [6]). Control-oriented approaches focus on coordinating the flow of control between entities.

In our comparison, we include at least one model used in component-based software development from both categories. As representatives of control-oriented coordination languages, we have chosen the X-MAN component model and WS-BPEL, a language for web service orchestration. As a representative of data-oriented coordination languages, we have chosen Reo. In our comparison, we focus on the abilities of compared coordination languages to represent interaction patterns. Because some of the languages in this category are not full-fledged component models (Reo, in particular), we do not compare them based on their implementation of our case study; instead, we analyse their properties through small illustrative examples.

In Sections 11.2.1, 11.2.2 and 11.2.3, we analyse X-MAN, WS-BPEL and Reo, respectively. Section 11.2.4 summarises our findings.

11.2.1 X-MAN

In X-MAN [74] (see Section 4.6.5), one can define solutions of some design patterns as composite connectors with constraints [70].

Example

For illustration, we show the definition of the Observer pattern in X-MAN (see Figure 11.3). Since X-MAN connectors coordinate components in terms of control flow, the composite connector realising the Observer pattern focuses on the control coordination aspect of the pattern's solution. The composite connector (the rounded rectangle in the figure) comprises two connectors, a pipe and a cobegin, and is able to compose three components, be means of its parameters: one represents the subject role in the pattern (denoted S in the figure) and the other two represent observers (O1 and O2). The connector first invokes S, collects its output and delivers it to O1 and O2 (the behaviour defined by the pipe); the observers are then concurrently invoked (by the cobegin connector)to process the data coming from S.

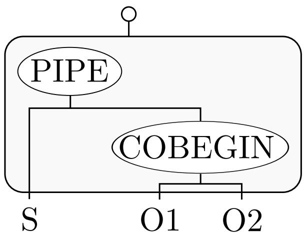


Figure 11.3: A Connector Representation of Observer in X-MAN

The pattern representation in X-MAN also contains additional constraints, specified in an OCL-like language. The constraint specification consists of a precondition, which has to be fulfilled by components before they can be composed by the composite connector, and a postcondition, which specifies some properties of the resulting composition. For the Observer pattern, the constraint is shown in Figure 11.4. The precondition (lines 1-5) requires that the subject (S) has some operations in its interface which return some outputs and that the two observers contain operations that can take these outputs as part of their input parameters. The postcondition (lines 7-10) specifies the flow of data from S to input parameters of operations of the two observers (only the case of O1 is shown).

Analysis

Compared to our approach, X-MAN composite connectors do not explicitly model data flow and are thus less expressive. Although the constraints, which also form part of the pattern definition, can define some elements of data flow (as shown in Figure 11.4), they cannot define the routing of input and output data

```
-- precondition :
let M, M1: Set(Method)
M1 = S.methods(all)->select(m:Method | m.Post implies (length(m.output) > 0))
M = M1->select (m1:Method | O1.methods->exists (m2:Method |
      m2.input->includes(m1.output)) and O2.methods->exists(m3:Method |
      m3.input->includes(m1.output))
M->size() >0

-- postcondition (the part for O2 is analogous and was omitted )
let pos: Integer
pos = O1.methods(invoke).input->indexOf(S.methods(invoke).output)
O1.methods(invoke).input(pos .. (pos + length(S.methods(invoke).output))).value =
      S.methods(invoke).output.value
```

Figure 11.4: The Observer Connector Constraint

fully. The reason is that, unlike our approach, composite connector interfaces are not typed and the interfaces of composed components are therefore unknown when the connector is defined. Furthermore, the constraint language has not been defined fully, and its semantics is thus unclear. Another advantage of our approach in pattern modelling is its ability to model control flow and data flow independently due to its control-driven and data-driven execution semantics. Finally, X-MAN connectors have fixed structure, which negatively impacts their reuse potential.

11.2.2 WS-BPEL

WS-BPEL [61] (see Section 4.6.6) can represent interaction patterns by means of WS-BPEL processes.

A WS-BPEL process coordinates several web services by means of a workflow and exports the resulting functionality as new composite web services. The specification of a process is an XML document comprising (i) the definition of so-called partner links to WSDL interfaces of the coordinated web services and of the new composite services, (ii) the declaration of variables, (iii) the specification of the main process workflow and, optionally, (iv) the specification of asynchronous event handlers.

Example

The following code shows an abridged specification of an example WS-BPEL process that coordinates two web services:

```
 1 <process name="BuyRecommendedProduct" ... >
 2   <partnerLinks>
 3     <partnerLink name="client" ... />
 4     <partnerLink name="recommenderService" ... />
 5     <partnerLink name="shoppingService" ... />
 6   </partnerLinks>
 7   <variables>
 8     <variable name="RecommenderResponse" messageType=.../>
 9     <variable name="ShoppingRequest" messageType=.../>
10     <variable name="ClientId" messageType=.../>
11   </variables>
12   <sequence>
13     <receive partnerLink="client" ... operation="buyRecommendedProduct"
           variable="ClientId" createInstance="yes"/>
14     <invoke partnerLink="recommenderService" ... operation="recommendProduct"
           outputVariable="RecommenderResponse" />
15     <assign><copy>
16       <from>bpel:doXslTransform('Tx.xsl', $RecommenderResponse, $ClientId)</from>
17       <to variable="ShoppingRequest"/>
18     </copy></assign>
19     <invoke partnerLink="shoppingService" ... operation="buy"
           inputVariable="ShoppingRequest"/>
20     <reply partnerLink="client" partnerLink="client" ...
           operation="buyRecommendedProduct"/>
21   </sequence>
22 </process>
```

The process, called BuyRecommendedProduct, sequences invocations to a product recommendation service and a product shopping service to yield a composite web service that buys a currently recommended product for the client. It defines three partner links (two for the coordinated web services and one for the interface of the composite service; lines 2-6), three variables for manipulating services' inputs and outputs (lines 7-11) and the main workflow (lines 12-21).

The workflow is defined by control flow constructs composing built-in WS-BPEL activities. Thus, WS-BPEL is a control-oriented coordination language. The example shows a sequence activity that executes nested activities in sequence. The workflow starts when a client invokes the buyRecommendedProduct operation from the composite web service's interface (line 13). It follows by invoking the product recommendation service's operation recommendProduct (line 14). The output of the service is then transformed using an XSL transformation to the input for the shopping service (lines 15-18), which is invoked subsequently (line 19). The workflow ends by the reply activity (line 20), which terminates the processing of the client's original request.[3]

Analysis

The repertoire of control flow constructs in WS-BPEL is richer than in our model. In addition to sequencing, branching and looping, concurrent execution of activities is supported.

Data flow is modelled differently than in our approach. WS-BPEL models data flow indirectly by assignments to mutable variables. Variables can store inputs and outputs of web service invocations; the receive and invoke activities receive them from or send them to web services, respectively. The variables declared directly within the process element are globally accessible within the process, which makes it challenging to avoid race conditions during simultaneous updates of their values by concurrently running activities. On the other hand, in our approach, data flow is modelled directly by data channels and is more akin to the functional programming style in nature since it avoids global, mutable variables; instead, data channels move immutable values locally between channel ends.

Another difference between WS-BPEL and our approach lies in the level of separation of computation and interaction. While composite connectors in our model define data flow and control flow only, WS-BPEL processes can contain some computation. The previous example contains the call to an XSL transformation (line 16) to compute a new value of the variable ShoppingRequest from the output of the recommendation service, stored in the RecommenderResponse variable.

Compared to our approach, WS-BPEL processes have smaller potential to be reused as stand-alone interaction patterns. Firstly, like other operation-based approaches, they depend on operation names and exact data types of all inputs and outputs. Secondly, unlike our connectors, they cannot be directly composed; there is no concept of composition of WS-BPEL processes. They can only be composed indirectly via the web services they define, but this, e.g., precludes using a repository of WS-BPEL processes to create a new process.

[3]The example contains no asynchronous event handlers – activities triggered by an occurrence of some event. Here, we only focus on coordination imposed by the main workflow.

Other Workflow Modelling Languages

As its name suggests, WS-BPEL orchestrates web services to model and execute business processes. It therefore belongs to a wider family of so-called workflow modelling languages. Here, we briefly mention this related area. However, full comparison with these approaches is out of the scope of this thesis.

A workflow is an automation of a business process [59]; it defines a set of activities that various participants carry out and information that participants exchange in order to contribute to some business goal. In general, workflows can describe more than software systems, because some workflow activities may be carried out by humans, and thus go beyond pure computation.

The two main application domains of workflow modelling are business process modelling, from which workflow originated, and scientific workflow modelling. WS-BPEL is the de facto standard in business process modelling[4] [11]; other languages, such as YAWL [1], have smaller industrial acceptance. Scientific workflow systems, such as Taverna [89] and Kepler [78], are used for the modelling and execution of computationally expensive experiments.

11.2.3 Reo

Reo [5, 6] (see Section 4.6.7) can represent interaction patterns by means of Reo connectors.

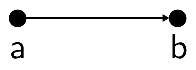
As mentioned in Section 4.6.7, Reo connectors consist of data channels composed by means of nodes. Nodes join several channel ends and actively move data among the connected channels while they can. A node connecting non-empty sets of sink ends I and source ends O non-deterministically chooses a datum from a data channel whose end lies in I and copies the datum to *all* of the channels whose ends are in O, but only if all the receiving channels can accept the datum. Nodes that have either I or O empty read data from or write data to outside of the connector in which they are defined; they therefore form connector interfaces.

Examples

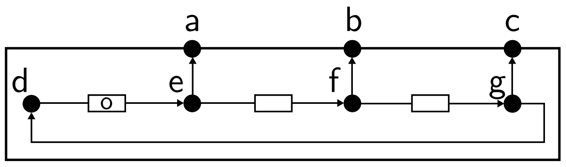
Figure 11.5 shows two example Reo connectors. The connector in Figure 11.5a comprises only one synchronous data channel and two nodes (`a` and `b`), which form the interface of the connector. A synchronous channel has a sink and source

[4]There exists a WS-BPEL extension for coordinating human activities to help model general workflows [66].

ends and no internal buffer; it forces a writer and a reader of the channel to synchronise (i.e., to wait until both are ready) in order to exchange a datum. The node a passes data from the writer (other connector or component) to the channel; the node b passes data from the channel to the reader. Nodes propagate synchronisation constraints of their adjacent channels.



(a) One-Channel Connector



(b) Ternary Sequencer

Figure 11.5: Examples of Reo Connectors

Figure 11.5b shows a less trivial Reo connector that enforces the sequencing behaviour among entities connected via its three interface nodes (a, b and c). The connector comprises seven nodes, four synchronous channels and three FIFO-1 channels. A FIFO-1 channel has the buffer of the size one; reads of a full channel and writes to an empty channel are executed immediately, whilst reads of an empty channel and writes to a full channel result in blocking (i.e., waiting for the channel's buffer to change). The left-most FIFO-1 channel is initialised with the datum o. The connector coordinates the three connected entities by having them read the datum o in sequence. First, only the inner node e can perform some activity, because its connected sink end belongs to a channel that contains data. The node e can take o and copy it to the two channels, whose source ends are connected to e, only if both channels can be written to. The FIFO-1 channel connecting e and f can accept a datum since it is empty. However, the synchronous channel connecting e and a can only be written to if the entity connected to a is ready to read the written value. The node e thus has to wait until that happens before it can act. After the entity connected to a becomes ready, e atomically removes the datum o from the FIFO-1 and copies it to both writeable adjacent channels. The situation then repeats for the node f, which ensures that the entity connected to b is sent o second. The node g then has to wait for two synchronous channels (going to c and d) to become writeable before it copies o to the entity connected to c and back to the left-most FIFO-1 channel. After that, the connector re-enters the state depicted in Figure 11.5b.

Analysis

The examples illustrate what assumptions Reo makes about component behaviour: (i) coordinated components are active (e.g., have their own thread of control) and (ii) they call operations on connectors' interface nodes, which is how their execution is coordinated. For example, after the component connected to the node a of the connector in the Figure 11.5a issues the writing operation, it will be blocked until the component connected to the node b issues the corresponding reading operation.

The Reo's assumption of active components contrasts with the assumption of passive components in our model. This has significant consequences on how connectors – and thus representations of interaction patterns – in both approaches differ. Reo connectors model data flow and enforce waiting between active components via synchronisation constraints. Connectors in our approach model data flow and control flow, which trigger execution of passive components. As a result, the two approaches take opposing strategies for component coordination: whereas Reo coordinates components by *blocking* their active execution in some order, our model coordinates components by *triggering* their execution in some order (from their default, passive state).

This explains the lack of synchronous data channels in our model (conversely, Reo defines more kinds of data channels than our model since Reo's data channels can be synchronous and have both channel ends of one type). Synchronisation constraints, enforced by synchronous data channels in Reo, are dealt with in our model (i) by control flow constructs (e.g., sequencing), (ii) by component execution semantics (e.g., waiting for all inputs before a data-driven component can be executed), and (iii) by global synchronisation rules (e.g., a control-driven component needs to wait for all inputs computable in the current execution cycle before it can skip triggering).

Finally, Reo's focus on composition of active components results in control flow being hidden inside components (which are not specified in Reo). On the other hand, our approach can model all control flow in a system explicitly, because it relies on passive components and because it is a full-fledged component model. This gives us more opportunities for interaction pattern modelling.

11.2.4 Summary

In this section, we summarise the results of comparison of our approach with other coordination languages used in component-based software development. We focus on expressiveness and reusability of various interaction pattern representations.

Expressiveness

Although all of the examined coordination languages are exogenous, and thus able to separate computation and interaction, their ability to define interaction patterns varies. The most distinguishing characteristic of our approach, compared to other coordination languages, is the equal emphasis on control flow and data flow modelling, enabled by the underlying control-driven and data-driven execution semantics. Compared to control-oriented coordination languages, data flow is modelled explicitly and directly using data channels, rather than implicitly in X-MAN or indirectly via re-writing shared variables in WS-BPEL. Compared to data-oriented languages, represented here by Reo, control flow is explicit, rather than hidden inside active components whose data exchanges are coordinated by Reo connectors.

X-MAN's composite connectors can define the control flow aspect of interaction patterns, but they lack expressiveness due to their inability to define data flow completely.

WS-BPEL has more control flow constructs than our approach as it supports concurrent control flows. It also allows data flow to be defined, albeit indirectly via shared variables; this gives rise to race conditions in the presence of concurrency.

Reo's connectors can model data flow aspects of interaction patterns, but they lack control flow modelling capabilities. In fact, Reo connectors also impose synchronous constraints on their coordinatees, which play the opposite role to control flow in our approach; however, unlike our approach, these two aspects – data flow definition and synchronous constraints – are not separated and are indistinguishable at the architectural level.

Reusability

X-MAN composite connectors can be reused via repositories; on the other hand, their reuse potential is decreased by low variability (no generic types, composite data flows or structural variability).

WS-BPEL processes cannot be reused directly via repositories since they are not compositional and thus cannot be reused for construction of other processes. They can only be reused indirectly via web services they define, which can be coordinated by another WS-BPEL process. Unlike our approach, web service reuse is limited by the dependency of their coordinators on operation names and order of their parameters. Web services also lack structural variability, which further decreases their reuse potential.

Reo connectors can be reused via repositories and are highly variable (including dynamic structural variability [67]). Unlike our approach, Reo connectors are suitable for coordinating active components, which makes them suitable for representing different kinds of interaction patterns than our approach.

11.3 Approaches to Formalising Pattern Solutions

Design patterns (see Section 3.2) are wide-spread form of capturing and reusing design knowledge. However, they have a number of shortcomings (see Section 3.2.4) that thwart the reuse of pattern solutions in current mainstream design (e.g., UML) and implementation (e.g., Java) notations. As a result, a significant body of software engineering research addresses some shortcomings in the definition and usage of design pattern solutions. Our approach also falls in this category as it formalises solutions of interaction patterns as composite connectors and thus enables their design and implementation reuse.

We have conducted a literature survey of these approaches. Section 11.3.1 describes the surveyed methods; Section 11.3.2 evaluates how they contribute to system development with patterns and compares them to our approach to show its novelty and advancement on the state of the art in this area.

11.3.1 Groups of Surveyed Approaches

In our literature survey, we have included various methods that address some of the mentioned inadequacies of design patterns and their usage by defining pattern solutions formally and making them first-class entities in design or implementation. We have excluded methods that formalise other pattern parts than solution, such as problem description, in order to keep focus on techniques that assist developers during design and implementation with patterns.

Altogether, we have surveyed more than thirty approaches. For brevity, we do not describe each method individually; instead, we have grouped similar approaches and describe each group. Additionally, Appendix E contains examples of methods from every category. We have identified the following groups according to how they represent pattern solutions: logic constraints, diagrammatic constraints, design transformations, reusable implementations and composition operators.

Logic Constraints

There are many methods proposing formal specification languages for pattern solutions based on first-order or temporal logic [14, 116, 115, 38, 85, 46, 41, 20, 107, 108]. In general, they define a language to model the structural or behavioural aspects of object-oriented designs and then specify design patterns as predicates over entities defined in that modelling language. The predicates take entities corresponding to pattern participants as their parameters, and they define constraints that the participants have to fulfil in order to comply with the pattern. Appendix E.1 gives an example of the Observer pattern specified in BPSL (Balanced Pattern Specification Language) [116].

Diagrammatic Constraints

Another group of approaches model pattern solutions using some custom diagrammatic notations [76, 50, 43, 80, 40, 62]. They also define constraints on participating entities as the previous group of methods, but they define participants and their relations diagrammatically. This representation better lends itself to deriving design models of pattern instances.

Diagrammatic constraints and logic constraints groups may overlap. In our survey, there is one approach (LePUS3 [46]) that specifies patterns using diagrammatic notation that directly translates to a set of logic formulae. However, unlike other approaches in the diagrammatic constraints group, pattern specification diagrams in LePUS3 are not used to derive design models of pattern instances.

For example, Role-based Modelling Language (RBML) [43] represents patterns using customised UML diagrams that define a family of designs rather than a single design as traditional UML diagrams. Appendix E.2 gives an example of the Observer pattern specified in RBML.

Design Transformations

These methods [31, 57, 39] focus on formalising the process of pattern application rather than on formalising a pattern solution as such. They define pattern application as a transformation that maps a design model exhibiting some of the problems that the pattern solves to a design model which conforms to the pattern. In other words, they formalise pattern application as a refactoring that improves some quality of the input design model by re-arranging (renaming, adding and removing) the elements of the input model according to the pattern. Transformations correspond to (anti-pattern, pattern) pairs. Appendix E.3 gives an example of the Composite pattern specified in SLAM-SL.

Reusable Implementations

These approaches attempt to achieve code reuse of pattern solutions. They therefore formalise pattern solutions by encoding them in some programming language. Typically, they rely on various techniques that extend the basic repertoire of object-oriented language features (classes, inheritance and polymorphism) to yield pattern implementations variable enough to be instantiated in different contexts

The techniques encountered in our survey include abstract aspects in AspectJ [52], Eiffel classes [9], Haskell higher-order datatype-generic operators [49], Scala abstract and generic types [90], nested classes [88] and traits [95], static meta-programming techniques [37], reflection in Java [92], and several programming languages specialised on implementing design patterns [21, 42, 2]. Appendix E.4 gives an example of the Observer pattern implemented in AspectJ.

Composition Operators

These methods represent pattern solutions as composite connectors in the context of component-based software development. Our approach belongs to this category. The only other approach we are aware of is that of Lau et al. [70], in which pattern solutions are defined as X-MAN composite connectors with constraints (see Section 11.2.1).

11.3.2 Analysis

In this section, we analyse the surveyed methods to evaluate how they contribute to system development with patterns. Again, we abstract from individual approaches and use the groups presented in the previous section in our analysis. In particular, we focus on how the composition operator group of approaches, which contains our method, compares with other groups.

To assess how particular groups of methods help in system development with patterns, we analyse several criteria. Firstly, we establish whether they specify pattern solutions as stand-alone entities with concrete identity and representation. Secondly, we assess their ability to contribute to the main development stages of design and implementation; i.e., whether they produce any design or implementation artefacts that can be directly used in system development. Finally, we consider whether they define any particular development process in whose context they should be used. Table 11.3 summarises our findings.

Group of Approaches	**PSS**	**DA**	**IA**	**IDP**
Logic constraints	✓	×	×	×
Diagrammatic constraints	✓	✓	×	×
Design transformations	×	×	×	×
Reusable implementations	✓	×	✓	×
Composition operators	✓	✓	✓	✓

PSS	Pattern solution specification
DA	Design artefacts (re)usable in system development
IA	Implementation artefacts (re)usable in system development
IDP	Integration with a development process

Table 11.3: Comparison of Approaches Formalising Pattern Solutions

Approaches defining pattern solutions as logic constraints specify pattern solutions mostly in the form of predicates. They can be used for mining patterns in reverse engineering existing legacy systems, checking compliance of some design with a design pattern's solution or for some reasoning about patterns (e.g., about composition of patterns). However, they cannot be directly used in the design or implementation stages of system development. Because they do not focus on system construction, the details of any particular system development process are irrelevant to them.

Methods specifying pattern solutions as diagrammatic models can derive design models (such as UML diagrams) corresponding to pattern instances. None of the methods provides any means for generating implementation artefacts from the design models. Generally, none of the approaches in this category considers the wider context of the underlying development process, including the issue of how to integrate the generated design models of pattern instances with design models for the rest of the system.

Approaches that express pattern application as transformations of design models do not formalise pattern solutions themselves; they only define how a certain design configuration can be changed to another one that conforms to a pattern. However, the definition of these initial design configurations is problematic as it can never be complete. None of the surveyed methods in this group demonstrated feasibility on any non-trivial design configuration. By their definitions, these methods do not produce any design or implementation artefacts in the first place; they can only transform existing ones.

Approaches implementing pattern solutions using various programming language constructs, such as abstract aspects [52] or higher-order operators [49], produce artefacts that can be directly used in system implementation. However, they fail to identify a suitable representation of pattern solutions in design models, and thus lack links to other phases of system development process. Furthermore, programming languages cannot impose all of the constraints specified by pattern solutions, which would be better represented in more high-level design models.

Finally, the last group of methods define pattern solutions as composition operators in the context of component-based software development. Since they are represented by composite connectors, entities defined within a component model, they have a first-class status in system architecture (as design artefacts), but they also possess a behaviour definition that enables them to be (re)used as implementation artefacts. Moreover, component-based software development has a well-defined development process (see Section 4.1) into which they are integrated. Unlike most methods in other groups, these methods are not object-oriented and thus cannot express every object-oriented design pattern.

To sum up, the approaches in our survey vary considerably in their focus and are therefore applicable only to certain parts of system development. However, this endangers the adoption of these approaches as eventually they have to be integrated into some development process to become useful in system development.

Composition operators have the advantage of the integration into the component-based development process, which inherently spans design and implementation stages by the means of component model abstractions.

11.4 Overall Evaluation of Conducted Research

This section evaluates the research reported in this thesis with respect to our research goals. In Section 11.4.1, we argue that all our main research goals have been achieved. The limitations of our approach are discussed in Section 11.4.2.

11.4.1 Achievement of Research Goals

The research presented in this thesis has been guided by the three goals stated in Chapter 1, which refine our research hypothesis. Here, we argue that all these goals have been achieved, and we support our argument by summarising the main results of the research conducted to achieve each of the goals.

For convenience, we include our research goals here again. They are as follows:

- to identify suitable component model characteristics (i.e., characteristics of components and composition mechanisms) for expressing interaction patterns separately from computation,
- to survey existing component models in order to establish their conformance with such characteristics, and
- to either adapt an existing component model or to develop a new component model with such characteristics so that interaction patterns can be represented as composite connectors – explicit in system architecture and reusable across many systems – in that component model.

Suitable Model Characteristics

In Chapter 5, we have identified the following six characteristics that a component model suitable for expressing interaction patterns as connectors should possess: (i) an explicit architectural representation of connectors, (ii) the ability to define new connectors, (iii) separate specification of interaction and computation, (iv) explicit control flow and data flow modelling, (v) composable connectors, and (vi) separate control flow and data flow modelling.

Some of these characteristics are essential and directly follow from the overall aim of our research to represent interaction patterns as connectors. Thus, they form a minimal set of properties that any approach pursuing the given aim must exhibit. Namely, these are the characteristics (i) and (ii), which postulate first-class connectors, and the characteristic (iii), which demands the separation of modelling computation and interaction concerns between components and connectors.

However, none of these essential characteristics addresses how the interaction is modelled; this is addressed by the remaining characteristics. They express our observations of what makes *good* interaction modelling. As such, their choice is specific to our approach and they are its distinguishing characteristics. The characteristic (v) demands the composable nature of interaction specifications in order to increase reuse potential of interaction connectors by not only reusing their instances but also their definitions in construction of more complex connectors. The characteristics (iv) and (vi) formulate the basics of interaction modelling in our approach – using the well-established modelling abstractions of control flow and data flow that are both explicit and without any dependencies on each other.

To evaluate these characteristics directly (particularly the non-essential ones) is impossible due to their informal nature. However, their usefulness can be seen in how they helped us in surveying existing component models and subsequently in creating our own component model with connectors representing interaction patterns.

Survey of Existing Component Models

The second goal of our research is to survey the existing component models in order to find out whether there exists any component model that would exhibit the aforementioned characteristics and would thus be suitable to represent interaction patterns as connectors. We have conducted such a survey, whose results are presented in Chapter 5.

The main outcome of the survey is that none of the surveyed models exhibits all the desired characteristics for expressing interaction patterns. Consequently, we have developed a new component model to base our approach on, complying with all of the characteristics (see the next subsection).

The survey is comprehensive in its scope as it combines component models identified by two recent and highly cited component model surveys by Lau and

Wang [75] and Crnkovic et al. [36]. To analyse the compliance of a particular component model with the identified characteristics, we have drawn from the contents of the two survey papers as well as the original publications about the model to double check our results. Therefore, we believe that the analysis precisely captures the state of the art in current component models.

Additionally, besides establishing non-existence of an approach fully suitable for our purposes, the survey brings some insights into current component models, with respect to the analysed properties. Firstly, a large proportion of current component models (about two thirds) lack the ability to express and define connectors in system architectures. Secondly, composition mechanisms separating computation and interaction, namely exogenous composition and the pipe-and-filter composition style, are in minority in current component models, with the majority being based on the procedure call composition mechanism (the request-response interaction style). Thirdly, even the models that separate computation and coordination do not represent interaction in terms of control flow and data flow explicitly and without any constraints between the two flows.

Definition of the New Component Model and Composite Connectors

Following the result of the survey, which found no existing component model fully compliant with the required characteristics, we have developed a new component model designed to comply with the characteristics, according to our third research goal. Table 11.4 illustrates the compliance of our component model with the required characteristics.

We have defined (i) the structure of components, connectors and systems by means of a meta-model, and (ii) the run-time behaviour of systems constructed in the model (i.e., the model's execution semantics) in Chapter 6.

To be able to represent interaction patterns in our model, we have defined composite connectors, both their structure and behaviour (given by a transformation of a composite connector into a composition of basic connectors), in Chapter 7; we further refined them to increase pattern reuse in Chapter 8 by splitting connectors into two entities – reusable templates and context-specific instances – and by defining an instantiation transformation for creating the latter from the former.

To evaluate the feasibility of the above definitions, we have implemented a prototype tool, which allows design and simulation of systems developed in our

✓/ ×	**Characteristic**	**Justification**
✓	an explicit architectural representation of connectors	connectors present in architecture
✓	the ability to define new connectors	by connector composition
✓	separate specification of interaction and computation	components specify computation, connectors specify interaction
✓	explicit control flow and data flow modelling	control and data connectors
✓	composable connectors	connector composition defined
✓	separate control flow and data flow modelling	possible due to the control-driven and data-driven execution semantics

Table 11.4: Compliance of Our Component Model with the Desired Characteristics

component model (see Chapter 9). The tool has been used to model the mode-switching pattern, a real-world example of an interaction pattern from the domain of reactive control systems, and the climate control system, in which the pattern's instance coordinates mode switching (Chapter 10). The execution semantics has been formalised using Coloured Petri Nets; this gives us an unambiguous and precise description, which can be also used to reason about some of the properties of system execution (see Section 6.6).

Further evaluation by qualitative comparison with related work in this chapter confirms that the unique capabilities of our approach that correspond to the desired characteristics – explicit control flow and data flow modelling without any of the flows dominating, loose coupling of connector interfaces and the variability of connector templates – result in an interaction pattern representation that is more expressive and more reusable than the state of the art in component-based software development.

Moreover, the connector representation of patterns has been shown superior to other approaches to formalising pattern solutions, because pattern solutions in our approach have first-class status throughout design and implementation stages of the system development life cycle: they are explicit in system architecture as connectors, which are also the abstractions that define their run-time behaviour. Other approaches, not based on component-based development, lack such an abstraction that would bind pattern representations in multiple development phases.

Known limitations of our approach are discussed in Section 11.4.2; possible further research directions extending the work presented in this thesis can be found in Section 12.3.

11.4.2 Limitations of Our Approach

Since the research presented in this thesis has been conducted within a limited period of time and with a limited amount of resources, it is necessarily incomplete and lacking in a number of aspects. In this section, we discuss known limitations of our approach. We have grouped them into four categories: (i) structural definition of our component model, (ii) its execution semantics, (iii) representation accuracy of interaction patterns and (iv) prototype implementation.

Structural Definition of Our Component Model

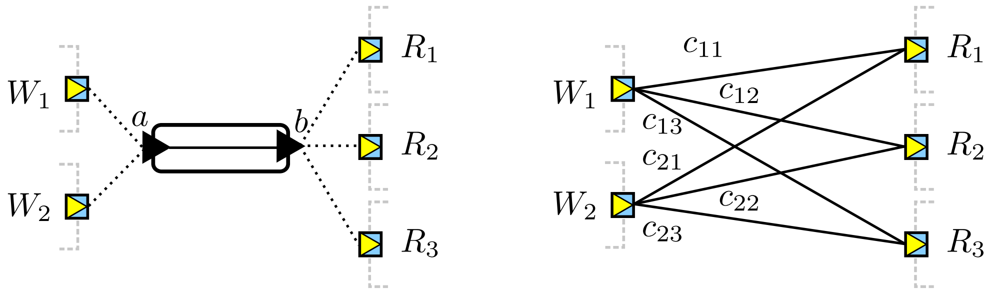
The structural definition of our model (that is, its meta-model) could be further extended in a number of ways. In particular, we are aware of the absence of hierarchical components, of data stores and of state-based routing within composite connectors.

Hierarchical components (also called composite components) are defined as compositions of several other components. They are an established means for tackling scalability and complexity in component-based software development. However, since our research has primarily focused on composition mechanisms represented by composite connectors (which can be hierarchically composed), the definition of composite components is missing in the current version of our component model. Moreover, defining composite components in our component model is non-trivial due to having multiple component types. Although composition of only control-driven components or data-driven components is quite straightforward (and could be done analogously to X-MAN or Scade), composing control-switched data-driven components is less obvious (How should their execution state be composed?) as is the composition of components of different types (What would be the resulting component type of a composition of a data-driven and a control-driven component?). Analysis and resolution of such questions requires further research.

Our component model in its current version lacks any representation of data stores – entities whose only role is to hold some data and provide access (reads or writes) to them to other entities. There exist other component model entities that encapsulate state, but storing state is not their main responsibility.

Firstly, components can be stateful, but they primarily represent computation and thus enforce some constraints that are unnecessary for a generic data store. For instance, a component does not execute (consume incoming data) until all of its input providers (writers) have data ready.

Secondly, data channels are stateful, because they have internal buffers (of various sizes and access policies). However, they transfer data between one writer and one reader, which does not suffice to fully model a data store with m readers and n writers.



(a) A Composite Connector (b) Its Semantics

Figure 11.6: Connector Representation of Data Store (3 Readers and 2 Writers)

Thirdly, composite data connectors are stateful as they comprise data channels. They alleviate the limitation of data channels, and they can model m readers and n writers scenarios. For example, Figure 11.6a shows a composite connector, comprising a single data channel, that realises the case of two writers and three readers, by connecting its required ports a and b to two output ports and three input ports, respectively. Semantically, the configuration is equivalent to the one depicted in Figure 11.6b, in which the channel is instantiated for each writer-reader pair. This also removes the above-mentioned component constraint since writers can produce data at different times. However, the behaviour of the configuration of data channels in Figure 11.6a is non-deterministic as it relies on the non-deterministic selection of several incoming values at the reader ports R_1, R_2, R_3. Introducing explicit data stores with other access policies would significantly affect the execution semantics and requires further investigation.

Another state-related limitation affects composite connectors. Modelling of a stateful interaction pattern by delegating the computation of its state changes to a component that feeds the current state to the composite connector representing the pattern creates the tight coupling between the component and the connector (see Section 7.4.1). This is caused by representing state using data values: state changes in this representation correspond to data value transformations (i.e., computation), which can only be carried out by components. However, some alternative ways of representing state within connectors might eliminate the need for state-changing components in representing stateful interaction patterns.

For example, control connectors in a composite connector maintain some implicit state, which corresponds to the location of the control signal in the connector's control hierarchy. A binary sequencer, for instance, has three implicit states: one for when control is not present, one for control triggering its first parameter and one for control triggering its second parameter; the state changes as control flows through the sequencer. Using this implicit state for data flow and control flow routing would enable some stateful interaction patterns (e.g., the ABA dialogue pattern in Section 7.4.1) to be defined fully within a composite connector. The extension of composite connectors along these lines has been left as a future work.

Execution Semantics

We have seen how the structural definition of a component model is affected by various aspects of execution semantics. Here, we discuss the limitations of the execution semantics of our model (beyond the technical limitations discussed in Section 6.5.7), as described in Section 6.5. In particular, we discuss (i) its domain-specificity, (ii) its performance problems and (iii) the single-threaded nature of control flow.

Firstly, the execution semantics of our component model is specific to the domain of reactive control systems. The concept of synchronous execution cycles and synchronisation rules enforcing deterministic system behaviour[5] match the characteristics of system execution in this domain. Nevertheless, system execution characteristics in other domains necessarily differ, which may result in sub-optimal properties that systems from other domains exhibit when executed according to our execution semantics.

For example, data-intensive processing systems prioritise performance (data throughput). Thus, an execution model supporting asynchronous parallel data processing would be more suitable than splitting execution into synchronous cycles.

[5]This holds provided that a system is composed of a deterministic sub-set of the component model.

Secondly, the current execution semantics exhibits some performance limitations, as the above example illustrates. Execution split into execution cycles adds a global synchronisation constraint: at the end of each execution cycle, all components stop executing as they have either no more input data to process (data-driven components) or the control loop iteration has finished (control-driven components). On the one hand, this allows us to have the hybrid execution semantics, which maximises the expressiveness of interaction modelling in terms of control flow and data flow; on the other hand, it results in poorer performance. Particularly, data-driven components have high performance potential due to the easily parallelisable and asynchronous nature of their execution.

Thirdly, control flow coordination is single-threaded and limited to one iteration of the main loop per execution cycle. This may seem to be a limiting factor, compared to other component models allowing multi-threaded execution (most of them rely on component implementation to deal with threading). However, this design decision leads to a single control coordination hierarchy, which simplifies system architecture. Furthermore, data-driven components may be executed concurrently in our current execution semantics, playing the role of multiple threads of control in control-driven component models.

Representation Accuracy of Interaction Patterns

Representing interaction patterns as composite connectors in our component model necessarily fails to capture interaction patterns in their entirety. The main limitations associated with the pattern representation in our approach are as follows:

Composite connectors in our approach only represent pattern solutions. We intentionally avoid modelling other pattern parts, because they are hard to formalise and – unlike pattern solutions – do not define any reusable design artefacts, which could be embodied in a component model.

Some aspects of pattern solutions cannot be represented in our component model. This is inevitable since we use our component model (a single formalism) to model pattern solutions coming from various other contexts. For instance, some object-oriented patterns involve dynamic creation or destruction of objects, which cannot be represented in the context of our model that is based on static architecture. Likewise, aspects of pattern solutions relying on inheritance cannot be expressed in the component-based setting.

We cannot specify constraints between participants. Constraints are only defined in terms of types associated with interface elements of composite connectors, but this fails to capture the constraints unrelated to the coordination behaviour realised by the connectors.

An example of such a constraint might be the requirement that some pattern participants cannot exchange data with each other. These participants end up being represented as components. But because only the data ports that interact with the pattern's coordination behaviour (i.e., the ports connected with the composite connector representing the pattern) are checked for type conformance, other data ports of the participant components can be connected in an arbitrary way, which may violate the requirement.

Prototype Implementation

The main purpose of the prototype implementation is to practically demonstrate our approach. Since the focus of our research is on component-based design, the prototype tool does not support deployment of designed systems to any hardware or software platform. It only allows designing systems in our component model, designing composite connectors and simulating system behaviour. Furthermore, many aspects of the prototype would need to be improved if the tool was to go beyond its research prototype status (e.g., the efficiency of the simulator and the component definition language).

Chapter 12

Conclusion

This thesis reports on our research in the area of component-based software engineering. In particular, our research focuses on high-level component composition mechanisms. Currently prevalent approaches to component-based development rely on primitive composition mechanisms, such as procedure calls, which are unable to model complex interactions separately from computation defined within components. This leads to software architectures in which computation and interaction are both mostly defined in components; interaction and computation are thus indistinguishable in software architecture and cannot be reused independently. To tackle these shortcomings, our research aims to define composition mechanisms – represented in software architecture as connectors, distinct from computation-defining components – that are able to encapsulate complex interactions. In this thesis, we investigate the possibility to base the definition of such composition mechanisms on interactions among participants within some design patterns.

To this end, we have identified a number of component model characteristics suitable for modelling interaction separately from computation. We have defined a new component model that conforms to these characteristics. We have defined composite connectors in this model that can encapsulate interaction and can be reused via repositories in different contexts, independently of components. To evaluate the feasibility of our approach, we have developed a prototype tool and carried out a case study from the reactive control system domain. We have also compared our approach with related component models, coordination languages and pattern solution formalisations for further evaluation.

The main contribution of our research lies in the novel way of representing interaction by means of composite connectors in our component model, which makes interaction patterns architecturally explicit and reusable. Comparison with the related work shows that our approach improves the state of the art in component-based interaction modelling. Other contributions of our work to existing knowledge in component-based software development include the analysis of interaction representations in current component models, the novel control-driven and data-driven execution semantics of our model and the novel way of increasing reuse of composite connectors through their structural variability. Our research contributions are further detailed in Section 12.1. Section 12.2 discusses the limitations of our work. We conclude in Section 12.3 by outlining interesting future research directions building on the research reported in this thesis.

12.1 Contributions

In this section, we highlight the contributions of our research to the current body of knowledge. They mainly fall into the area of component-based software engineering, but there are also contributions in the wider context of software engineering.

In Chapter 5, we have identified suitable component model characteristics for separating the specification of interaction and computation in system architecture and for having stand-alone composite connectors representing interaction patterns. The survey of existing component models confirmed that they represent a novel set of component model design criteria with unique focus on separate interaction modelling. The survey itself contributed to the knowledge of existing component models by analysing them with respect to these criteria.

These conceptual contributions have been materialised into the main technical contributions of this thesis in Chapters 6, 7 and 8. We have defined (even formally) the structure and execution semantics of the new component model, complying with the aforementioned characteristics, in Chapter 6. To be able to compose control flow and data flow elements comprising interaction patterns, we have defined composite connectors in that component model in Chapter 7. To further increase their reusability, we have defined several variability mechanisms for their instantiation from a repository in Chapter 8. Comparison with related component-based approaches showed that the composite connectors in the new component model enhance the state of the art in interaction modelling in component-based development by their expressiveness (both control flow and data flow are explicit) and by supporting separate reuse of interaction through their variability (particularly, the structural variability is novel). This is the main result of our research as it directly confirms our research hypothesis. Further, the control-driven and data-driven execution semantics of our model is novel; it improves the expressiveness of interaction modelling by removing any unnecessary dependencies between control flow and data flow, imposed by one being the dominant execution trigger over the other.

Additionally, the survey of pattern solution formalisation approaches in Chapter 11 contributed to existing knowledge by analysing how these methods support various phases of software development life cycle. The results suggest that – unlike most approaches which are confined to a single development phase – component-based approaches to pattern formalisation, such as ours, can represent patterns in both design and implementation phases, because they rely on component model abstractions that bridge these two phases.

12.2 Limitations

We also need to consider the main limitations of this work.

Firstly, some parts of the new component model definition are domain-specific. Particularly, the execution semantics is geared towards the domain of reactive control systems. The case study proving the feasibility has also been carried out in this domain only. The obtained results may thus not hold in other domains with significantly different requirements.

Secondly, the definition of the new component model is primarily focused on modelling the behavioural aspect of interaction patterns, which leaves some other aspects of interaction patterns unexplored. For example, dynamic architecture reconfiguration is not supported, which means that related interaction patterns cannot be represented. Likewise, extra-functional aspects of interaction patterns have not been investigated.

Thirdly, due to the informal nature of design patterns (and thus interaction patterns), it is impossible to design any formal representation that can be considered complete, in the sense of capturing a particular pattern's solution fully or in the sense of being able to capture all patterns. Therefore, the interaction pattern representation in our approach cannot be considered complete.

12.3 Future Work

The research reported in this thesis raises a number of new research ideas. In this section, we outline some future research directions that would extend the presented research to overcome some of its limitations and to widen the scope of its application. We first discuss possible extensions of our component model as a whole; we then narrow our focus on its control-driven and data-driven execution semantics and composite connectors.

Component Model Extensions

As pointed out in Section 12.2, our component model could be extended in a number of ways beyond the pure behaviour modelling, which was the focus of this thesis. This could improve various properties of software architectures constructed in our model. For example, hierarchical component composition improves the scalability of software architectures; extra-functional modelling capabilities increase the predictability of software architectures with respect to the modelled properties, such as performance or reliability; dynamic reconfiguration, i.e., the ability to change software architecture at run-time, increases architecture's adaptability. All these component model extensions would affect how interaction is represented, and it would be interesting to examine their implications. However, the choice of needed component model extensions depends on a particular domain; the research here should be therefore driven by the need to model systems in specific domains.

Control-driven and Data-driven Execution Semantics

The control-driven and data-driven execution semantics of our component model also raises interesting research questions. Again, requirements on system execution, such as the need for high performance or deterministic system behaviour, vary in different domains and could drive experimenting with the current execution semantics.

Furthermore, it would be interesting to investigate the applicability of the control-driven and data-driven coordination in the wider family of coordination-based approaches, for example, to perform orchestration in service-oriented architectures or to act as a workflow modelling language. Barker and van Hemert [11] observed the divide of workflow languages into control flow-oriented, used for business process modelling, and data-flow-oriented, used for scientific workflow modelling; it seems interesting to examine whether a control-driven and data-driven coordination language based on our component model could unify these approaches.

Another intriguing possible research direction would be to investigate the application of our model's control-driven and data-driven component modelling capabilities to architecture modelling of today's heterogeneous parallel systems. These systems are deployed to run on control flow dominated CPUs (central processing units) and data flow oriented GPGPUs (general-purpose graphics processing units). Modules running on CPUs are specified in a different language and executed differently from those running on GPGPUs. Could a component-based control-driven and data-driven approach, such as ours, be used to holistically design architectures of such systems?

Composite Connectors

Finally, composite connectors themselves can be explored further beyond their interaction pattern representation emphasised in this thesis. The existing structural variability mechanism based on roles and participants does not have the full expressiveness of a generic transformation language and thus may be insufficient for representing some variations of some pattern solutions. This motivates the research into more expressive variability mechanisms. The challenge here is to balance their expressiveness and ease of use for composite connector designers.

One might employ the variable nature of our composite connectors for other purposes than their instantiation from a repository. For example, if they were able to vary at run-time, the evolution of a system during dynamic reconfiguration might be guided by the permissible variants of composite connectors. As a result, system evolution would be restricted by the combined variants of composite connectors in a system and thus easier to verify compared to unrestricted approaches to dynamic reconfigurations.

Another possibility would be to use variability in composite connectors to define reference architectures in software product line engineering. Software product line engineering aims to develop a family of related software systems, rather than single systems. The product family is defined by a number of models. For example, a feature model defines the features of products in the family and their dependencies. The most important model is the reference architecture, which defines architectures of all products in a given family. Composite connectors with variable structure would play the role of variation points in a reference architecture; variants of each composite connector would be mapped to features in the product family. However, their variability parameters would not be fixed during instantiation from a repository into a reference architecture but later during the derivation of a product architecture from selected product features. That is, a reference architecture would become a product architecture after all of its composite connectors' variants were fixed according to the product's selected features.

Bibliography

[1] Wil M.P. van der Aalst and Arthur HM Ter Hofstede. Yawl: yet another workflow language. *Information systems*, 30(4):245–275, 2005. (Cited on pages 128 and 228.)

[2] Ellen Agerbo and Aino Cornils. How to preserve the benefits of design patterns. *ACM SIGPLAN Notices*, 33(10):134–143, October 1998. (Cited on page 234.)

[3] Christopher Alexander, Sara Ishakawa, and Murray Silverstein. *A Pattern Language: towns, buildings, construction.* Oxford University Press, New York, 1977. (Cited on pages 20 and 43.)

[4] Farhad Arbab. What do you mean, coordination? In *Bulletin of the Dutch Association for Theoretical Computer Science*, pages 11–22, 1998. (Cited on pages 20, 64, and 223.)

[5] Farhad Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):1–38, 2004. (Cited on pages 75, 82, and 228.)

[6] Farhad Arbab. Composition of interacting computations. In *Interactive Computation*, pages 277–321. Springer, 2006. (Cited on pages 75, 223, and 228.)

[7] Farhad Arbab. Coordination for component composition. *Electronic Notes in Theoretical Computer Science*, 160:15–40, 2006. (Cited on page 75.)

[8] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place EMF model transformations. In *MODELS*, volume 6394 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2010. (Cited on page 193.)

[9] Karine Arnout and Bertrand Meyer. Pattern componentization: The factory example. *Innovations in Systems and Software Engineering*, 2(2):65–79, 2006. (Cited on page 234.)

[10] Colin Atkinson, Philipp Bostan, Daniel Brenner, Giovanni Falcone, Matthias Gutheil, Oliver Hummel, Monika Juhasz, and Dietmar Stoll. Modeling components and component-based systems in KobrA. In *CoCoME*, volume 5153 of *Lecture Notes in Computer Science*, pages 54–84. Springer, 2007. (Cited on page 85.)

[11] Adam Barker and Jano I. van Hemert. Scientific workflow: A survey and research directions. In *Parallel Processing and Applied Mathematics*, volume 4967 of *Lecture Notes in Computer Science*, pages 746–753. Springer-Verlag, 2008. (Cited on pages 228 and 249.)

[12] Rémi Bastide and Eric Barboni. Software components: A formal semantics based on coloured petri nets. In *Workshop on Formal Aspects of Component Software*, Electronic Notes in Theoretical Computer Science, 2006. (Cited on page 85.)

[13] Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous component-based system design using the BIP framework. *IEEE Software*, 28(3):41–48, 2011. (Cited on pages 86 and 152.)

[14] Ian Bayley and Hong Zhu. Formal specification of the variants and behavioural features of design patterns. *Journal of Systems and Software*, 83(2):209–221, 2010. (Cited on page 233.)

[15] Kent Beck and Ward Cunningham. Using pattern languages for object-oriented programs. In *Workshop on Specification and Design, held in conjunction with OOPSLA'87*, December 1987. (Cited on pages 20 and 43.)

[16] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, January 2009. (Cited on pages 86 and 128.)

[17] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991. (Cited on pages 71 and 119.)

[18] Gérard Berry. Scade: Synchronous design and validation of embedded control software. In *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 19–33. Springer, 2007. (Cited on pages 66, 71, 82, and 86.)

[19] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992. (Cited on page 121.)

[20] Alex Blewitt, Alan Bundy, and Ian Stark. Automatic verification of design patterns in Java. In *International Conference on Automated Software Engineering*, page 224, New York, New York, USA, 2005. ACM Press. (Cited on page 233.)

[21] Jan Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, May 1998. (Cited on page 234.)

[22] Don Box. *Essential COM*. Addison-Wesley Professional, 1998. (Cited on pages 19, 58, and 85.)

[23] Alan W. Brown and Kurt C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, September / October 1998. (Cited on page 19.)

[24] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in Java. *Software – Practice and Experience*, 36(11–12):1257–1284, September/October 2006. (Cited on pages 59, 86, and 89.)

[25] Tomáš Bureš, Jan Carlson, Ivica Crnkovic, Séverine Sentilles, and Aneta Vulgarakis. *ProCom—the Progress Component Model Reference Manual*. Mälardalen University, June 2010. (Cited on page 215.)

[26] Tomáš Bureš, Petr Hnětynka, and František Plášil. SOFA 2.0: Balancing advanced features in a hierarchical component model. In *Software Engineering Research and Applications*, pages 40–48. IEEE Computer Society, 2006. (Cited on pages 59, 64, and 86.)

[27] Tomáš Bureš and František Plášil. Communication style driven connector configurations. In *Software Engineering Research and Applications*, volume

3026 of *Lecture Notes in Computer Science*, pages 102–116. Springer, 2004. (Cited on page 86.)

[28] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-oriented Software Architecture: On Patterns and Pattern Languages*, volume 5. John Wiley and Sons, 2007. (Cited on pages 20, 43, and 45.)

[29] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented Software Architecture: A System of Patterns*, volume 1. John Wiley and Sons, 1996. (Cited on pages 43 and 44.)

[30] Michel Chaudron. Reflections on the anatomy of software composition languages and mechanisms. In *Workshop on Composition Languages*, pages 15–24, Vienna, Austria, September 2001. (Cited on pages 61 and 62.)

[31] Mel Ó Cinnéide. Automated refactoring to introduce design patterns. In *International Conference on Software Engineering*, pages 722–724, 2000. (Cited on page 234.)

[32] Michael Clarke, Gordon S. Blair, Geoff Coulson, and Nikos Parlavantzas. An efficient component model for the construction of adaptive middleware. In *Middleware 2001*, volume 2218 of *Lecture Notes in Computer Science*, pages 160–178. Springer-Verlag, 2001. (Cited on page 85.)

[33] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting Software Architectures: Views and Beyond.* Pearson Education, 2002. (Cited on page 28.)

[34] Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. Mixing signals and modes in synchronous data-flow systems. In *International Conference on Embedded Software*, pages 73–82. ACM, 2006. (Cited on page 217.)

[35] Ivica Crnkovic, Michel Chaudron, and Stig Larsson. Component-based development process and component lifecycle. In *International Conference on Software Engineering Advances*. IEEE, October 2006. (Cited on page 54.)

[36] Ivica Crnkovic, Séverine Sentilles, Aneta Vulgarakis, and Michel R. V. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5):593–615, 2011. (Cited on pages 55, 62, 81, 82, and 239.)

[37] Daniel von Dincklage. Making patterns explicit with metaprogramming. In *Generative Programming and Component Engineering*, volume 2830 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 2003. (Cited on page 234.)

[38] Jing Dong, Paulo S.C. Alencar, Donald D. Cowan, and Sheng Yang. Composing pattern-based components and verifying correctness. *Journal of Systems and Software*, 80(11):1755 – 1769, 2007. (Cited on page 233.)

[39] Ghizlane El-Boussaidi and Hafedh Mili. A model-driven framework for representing and applying design patterns. In *Computer Software and Applications Conference*, pages 97–100. IEEE Computer Society, 2007. (Cited on page 234.)

[40] Maged Elaasar, Lionel C. Briand, and Yvan Labiche. A metamodeling approach to pattern specification. In *MoDELS 2006*, volume 4199 of *Lecture Notes in Computer Science*, pages 484–498. Springer, 2006. (Cited on page 233.)

[41] Andres Flores, Richard Moore, and Luis Reynoso. A formal model of object-oriented design and GoF design patterns. In *Lecture notes in computer science*, pages 223–241. Springer, 2001. (Cited on page 233.)

[42] Peter Forbrig and Ralf Lämmel. Programming with patterns. In *TOOLS*, pages 159–170. IEEE Computer Society, 2000. (Cited on page 234.)

[43] Robert B. France, Dae-Kyoo Kim, Sudipto Ghosh, and Eunjee Song. A UML-based pattern specification technique. *IEEE Transactions on Software Engineering*, 30(3):193–206, 2004. (Cited on pages 233 and 294.)

[44] Erich Gamma, Richard Helm, John Vlissides, and Ralph E. Johnson. *Design Patterns*. Addison-Wesley, 1995. (Cited on pages 20, 21, 26, 42, 43, 44, 47, and 291.)

[45] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems*, chapter 3, pages 47–67. Cambridge University Press, New York, NY, 2000. (Cited on page 82.)

[46] Epameinondas Gasparis, Jonathan Nicholson, and Amnon H. Eden. Lepus3: An object-oriented design description language. In *International Conference on Diagrammatic Representation and Inference*, pages 364–367, Berlin, Heidelberg, 2008. Springer-Verlag. (Cited on page 233.)

[47] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992. (Cited on pages 20, 22, 24, 38, 39, and 223.)

[48] Thomas Genßler, Oscar Nierstrasz, and Bastiaan Schönhage. Components for embedded software — the PECOS approach. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2002. (Cited on page 86.)

[49] Jeremy Gibbons. Design patterns as higher-order datatype-generic programs. In *ACM SIGPLAN Workshop on Generic Programming*, pages 1–12. ACM, 2006. (Cited on pages 234 and 236.)

[50] Alain Le Guennec, Gerson Sunyé, and Jean-Marc Jézéquel. Precise modeling of design patterns. In *«UML» 2000*, volume 1939 of *Lecture Notes in Computer Science*, pages 482–496. Springer, 2000. (Cited on page 233.)

[51] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991. (Cited on pages 71 and 121.)

[52] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. *ACM SIGPLAN Notices*, 37(11):161–173, November 2002. (Cited on pages 234, 236, and 296.)

[53] Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, Mats Lindberg, John Lundbäck, and Kurt-Lennart Lundbäck. The Rubus component model for resource constrained real-time systems. In *Symposium on Industrial Embedded Systems*, pages 177–183. IEEE, 2008. (Cited on page 86.)

[54] David Harel and Amir Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems*, pages 477–498, New York, NY, USA, 1985. Springer-Verlag New York, Inc. (Cited on page 39.)

[55] Patrik Haslum. Patterns in reactive programs. In *Cognitive Robotics Workshop*, 2004. (Cited on pages 24, 39, and 40.)

[56] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, June 2001. (Cited on pages 19, 53, and 55.)

[57] Angel Herranz and Juan José Moreno-Navarro. Modeling and reasoning about design patterns in SLAM-SL. In Toufik Taibi, editor, *Design Pattern Formalization Techniques*. Idea Group Inc., 2007. (Cited on pages 234 and 295.)

[58] Scott Hissam, James Ivers, Daniel Plakosh, and Kurt Wallnau. Pin component technology (v1.0) and its C interface. Technical report, Software Engineering Institute, Carnegie Mellon University, 2005. CMU/SEI-2005-TN-001. (Cited on page 86.)

[59] David Hollingsworth. The workflow reference model, 1995. (Cited on page 228.)

[60] Kurt Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use*, volume 1. Springer, 1996. (Cited on page 128.)

[61] Diane Jordan and John Evdemon. Web Services Business Process Execution Language Version 2.0. OASIS Standard, April 2007. `http://docs.oasis-open.org/wsbpel/2.0/`. (Cited on pages 74, 86, and 225.)

[62] Ahmad Waqas Kamal and Paris Avgeriou. Modeling the variability of architectural patterns. In *Symposium on Applied Computing*, pages 2344–2351. ACM, 2010. (Cited on page 233.)

[63] Xu Ke, Krzysztof Sierszecki, and Christo Angelov. COMDES-II: A component-based framework for generative development of distributed real-time control systems. In *Embedded and Real-Time Computing Systems and Applications*, pages 199–208. IEEE Computer Society, 2007. (Cited on page 86.)

[64] Rudolf K. Keller, Jean Tessier, and Gregor von Bochmann. A pattern system for network management interfaces. *Communications of the ACM*, 41(9):86–93, 1998. (Cited on page 46.)

[65] Ji Eun Kim, Rahul Kapoor, Martin Herrmann, Jochen Härdtlein, Franz Grzeschniok, and Peter Lutz. Software behavior description of real-time embedded systems in component based software development. In *IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 307–311. IEEE Computer Society, 2008. (Cited on page 86.)

[66] Matthias Kloppmann, Dieter Koenig, Frank Leymann, Gerhard Pfau, Alan Rickayzen, Claus von Riegen, Patrick Schmidt, and Ivana Trickovic. WS-BPEL extension for people – BPEL4People. Technical report, IBM and SAP, 2005. (Cited on page 228.)

[67] Christian Krause, Ziyan Maraikar, Alexander Lazovik, and Farhad Arbab. Modeling dynamic reconfigurations in Reo using high-level replacement systems. *Science of Computer Programming*, 76(1):23–36, January 2011. (Cited on page 232.)

[68] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995. (Cited on pages 28 and 63.)

[69] Ouassila Labbani, Jean-Luc Dekeyser, and Pierre Boulet. Mode-automata based methodology for Scade. In *Hybrid Systems: Computation and Control*, pages 386–401. Springer, 2005. (Cited on pages 24, 39, 40, 48, 203, and 217.)

[70] Kung-Kiu Lau, Ioannis Ntalamagkas, Cuong M. Tran, and Tauseef Rana. (behavioural) Design patterns as composition operators. In *Component-based Software Engineering*, volume 6092 of *Lecture Notes in Computer Science*, pages 232–251. Springer-Verlag, 2010. (Cited on pages 21, 128, 223, and 234.)

[71] Kung-Kiu Lau and Tauseef Rana. A taxonomy of software composition mechanisms. In *EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 102–110. IEEE, 2010. (Cited on page 61.)

[72] Kung-Kiu Lau, Lily Safie, Petr Štěpán, and Cuong Tran. A component model that is both control-driven and data-driven. In *Component-based Software Engineering*, pages 41–50. ACM, 2011. (Cited on page 91.)

[73] Kung-Kiu Lau, Faris M. Taweel, and Cuong M. Tran. The W model for component-based software development. In *EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 47–50. IEEE, 2011. (Cited on page 54.)

[74] Kung-Kiu Lau, Perla Velasco Elizondo, and Zheng Wang. Exogenous connectors for software components. In *Component-based Software Engineering*, volume 3489 of *Lecture Notes in Computer Science*, pages 90–106, 2005. (Cited on pages 64, 65, 73, 82, 86, and 223.)

[75] Kung-Kiu Lau and Zheng Wang. Software component models. *IEEE Transactions on Software Engineering*, 33(10):709–724, October 2007. (Cited on pages 19, 54, 55, 56, 58, 74, 81, 82, 155, and 239.)

[76] Anthony Lauder and Stuart Kent. Precise visual specification of design patterns. *Lecture Notes in Computer Science*, 1445:114–134, 1998. (Cited on page 233.)

[77] Doug Lea. Design patterns for avionics control systems, January 17 1995. (Cited on pages 24, 39, and 40.)

[78] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew B. Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006. (Cited on page 228.)

[79] Hugh Maaskant. A robust component model for consumer electronic products. In *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, volume 3 of *Philips Research*, pages 167–192. Springer Netherlands, 2005. (Cited on page 85.)

[80] David Maplesden, John G. Hosking, and John C. Grundy. A visual language for design pattern modelling and instantiation. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 338–339. IEEE Computer Society, 2001. (Cited on page 233.)

[81] Josip Maras, Luka Lednicki, and Ivica Crnkovic. 15 years of CBSE symposium: impact on the research community. In *Component-based Software Engineering*, pages 61–70. ACM, 2012. (Cited on page 82.)

[82] Malcolm Douglas McIlroy. Mass produced software components. In *Software Engineering*, pages 138–150. NATO Science Committee, January 1969. (Cited on pages 19 and 54.)

[83] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000. (Cited on page 63.)

[84] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *International Conference on Software Engineering*, pages 178–187. ACM, 2000. (Cited on page 62.)

[85] Tommi Mikkonen. Formalizing design patterns. In *International Conference on Software Engineering*, pages 115–124, Washington, DC, USA, 1998. IEEE Computer Society. (Cited on page 233.)

[86] Object Management Group. Corba component model specification. `http://www.omg.org/spec/CCM/4.0/`, 2006. [Online; accessed 18-05-2013]. (Cited on page 85.)

[87] Object Management Group. Unified modeling language specification v2.4.1. `http://www.omg.org/spec/UML/2.4.1/`, 2011. [Online; accessed 18-05-2013]. (Cited on page 86.)

[88] Martin Odersky and Matthias Zenger. Scalable component abstractions. *ACM SIGPLAN Notices*, 40(10):41–57, October 2005. (Cited on page 234.)

[89] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004. (Cited on page 228.)

[90] Bruno C. d. S. Oliveira, Meng Wang, and Jeremy Gibbons. The visitor pattern as a reusable, generic, type-safe component. In *OOPSLA*, pages 439–456. ACM, 2008. (Cited on page 234.)

[91] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala component model for consumer electronics software. *IEEE Computer*, pages 78–85, March 2000. (Cited on page 86.)

[92] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Computer Software and Applications Conference*, pages 9–15, Los Alamitos, California, August 1998. IEEE. (Cited on page 234.)

[93] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. In *The Engineering of Large Systems*, volume 46 of *Advances in Computers*. Academic Press, August 1998. (Cited on pages 20, 24, and 223.)

[94] Portland pattern repository. `www.c2.com`. [Online; accessed 26-April-2013]. (Cited on page 44.)

[95] Michael Pradel and Martin Odersky. Scala roles - A lightweight approach towards reusable collaborations. In *International Conference on Software and Data Technologies*, pages 13–20. INSTICC Press, 2008. (Cited on page 234.)

[96] Anne V. Ratzer, Lisa Wells, Henry Michael Lassen, Mads Laursen, Jacob Frank Qvortrup, Martin Stig Stissing, Michael Westergaard, Søren Christensen, and Kurt Jensen. CPN tools for editing, simulating, and analysing coloured petri nets. In *Applications and Theory of Petri Nets*, volume 2679 of *Lecture Notes in Computer Science*, pages 450–462. Springer, 2003. (Cited on pages 128 and 138.)

[97] Linda Rising. *The Pattern Almanac 2000.* Addison Wesley, 2000. (Cited on page 44.)

[98] Linda Rising. Understanding the power of abstraction in patterns. *IEEE Software*, 24(4):46–51, 2007. (Cited on page 45.)

[99] Linda Rising. The benefit of patterns. *IEEE Software*, 27(5):15–17, 2010. (Cited on page 45.)

[100] Douglas C. Schmidt and Chris Cleeland. Applying a pattern language to develop extensible ORB middleware, July 29 2000. (Cited on pages 42 and 46.)

[101] Bran Selic. An architectural pattern for real-time control software. In *Collected papers from the PLoP '96 and EuroPLoP '96 Conferences*. Washington University, October 17 1996. Technical Report, #wucs-97-07. (Cited on pages 10, 24, 25, 39, and 40.)

[102] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003. (Cited on page 187.)

[103] Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš, Jan Carlson, and Ivica Crnkovic. A component model for control-intensive distributed embedded systems. In *Component-based Software Engineering*, volume 5282 of *Lecture Notes in Computer Science*, pages 310–317. Springer, 2008. (Cited on pages 66, 69, and 86.)

[104] Mary Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In *ICSE Workshop on Studies of Software Design*, volume 1078 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 1996. (Cited on page 62.)

[105] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Englwood Cliffs, NJ, 1996. (Cited on pages 19, 39, 40, 58, 62, 63, and 78.)

[106] MathWorks Simulink. `www.mathworks.co.uk/products/simulink/`. [Online; accessed 4-June-2013]. (Cited on pages 71, 82, and 86.)

[107] Jason McC. Smith and David Stotts. SPQR: flexible automated design pattern extraction from source code. volume 0, page 215, Los Alamitos, CA, USA, 2003. IEEE Computer Society. (Cited on page 233.)

[108] Neelam Soundarajan and Jason O. Hallstrom. Responsibilities and rewards: Specifying design patterns. In *International Conference on Software Engineering*, pages 666–675. IEEE Computer Society, 2004. (Cited on page 233.)

[109] Petr Štěpán. Design pattern solutions as explicit entities in component-based software development. In *Workshop on Component-oriented Programming*, pages 9–16. ACM, 2011. (Cited on page 77.)

[110] Petr Štěpán and Kung-Kiu Lau. Controller patterns for component-based reactive control software systems. In *Component-based Software Engineering*, pages 71–76. ACM, 2012. (Cited on page 139.)

[111] Christoph Suender, Alois Zoitl, James H. Christensen, Heinrich Steininger, and Josef Fritsche. Considering IEC 61131-3 and IEC 61499 in the context

of component frameworks. In *Industrial Informatics*, pages 277–282. IEEE, 2008. (Cited on page 83.)

[112] Sun Microsystems, Inc. JavaBeans Specification. `http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html`, 1997. [Online; accessed 13-08-2013]. (Cited on pages 58 and 85.)

[113] Sun Microsystems, Inc. JSR 220: Enterprise JavaBeans 3.0. `http://jcp.org/en/jsr/detail?id=220`, 2007. [Online; accessed 18-05-2013]. (Cited on pages 19, 58, and 85.)

[114] Clemens A. Szyperski. *Component Software — Beyond Object-Oriented Programming*. Addison Wesley, second edition, 2002. (Cited on pages 19, 53, and 54.)

[115] Toufik Taibi. Stepwise Refinement Validation of Design Patterns Formalized in TLA+ using the TLC Model Checker. *Journal of Object Technology*, 8(2):137–161, 2009. (Cited on page 233.)

[116] Toufik Taibi and Fathi Taibi. Formal Specification of Design Patterns and Their Instances. pages 33–36. Ieee, 2006. (Cited on pages 233 and 293.)

[117] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture - Foundations, Theory, and Practice*. Wiley, 2010. (Cited on page 63.)

[118] Thuan L. Thai and Hoang Lam. *.NET Framework Essentials (2nd Edition)*. O' Reilly & Associates, Inc., 2002. (Cited on page 85.)

[119] The Eclipse Foundation. The Eclipse Modeling Project. `http://www.eclipse.org/modeling/`, 2013. [Online; accessed 22-08-2013]. (Cited on page 184.)

[120] The OSGi Alliance. OSGi service platform — core specification, August 2005. Release 4. (Cited on pages 58 and 85.)

[121] Walter F. Tichy. A catalogue of general-purpose software design patterns. In *TOOLS (23)*, pages 330–339. IEEE Computer Society, 1997. (Cited on pages 39 and 40.)

[122] Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. Translating discrete-time Simulink to Lustre. *ACM Transactions on Embedded Computing Systems*, 4(4):779–818, November 2005. (Cited on page 72.)

[123] John von Neumann. First draft of a report on the EDVAC (1945). In Brian Randell, editor, *The Origins of Digital Computers: Selected Papers*, pages 383–392. Springer Verlag, 1982. (Cited on page 41.)

[124] Yahoo! design pattern library. `developer.yahoo.com/ypatterns/`. [Online; accessed 26-April-2013]. (Cited on page 44.)

[125] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, first edition, 1979. (Cited on page 42.)

Index of Definitions

Appendix A

CPN Formalisation of Execution Semantics

In this appendix, we present some additional material on the formalisation of our component model's execution semantics.

A.1 Component Model Elements

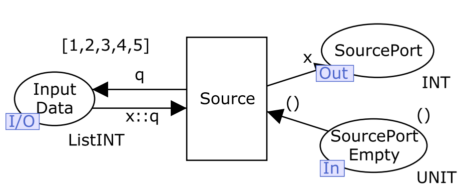
In this section, we show CPN models of most component model elements. For simplicity, they do not represent synchronisation rules.

A.1.1 Components

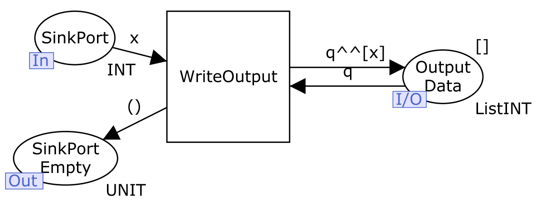
Source and Sink Components

Source and sinks have CPN models comprising their data ports, a place storing the data for each data port and a transition that moves data (token colours) between the two. Figure A.1[1] shows an example of a source and sink components with one data port of the integer type. Note that we use the CPN's ability to define and manipulate token colours for complex data types – each token colour of the type ListInt represents a list of integers; the transition's occurrence then either removes the first element of the list (Source) or adds an element at the end of the list (Sink).

[1]The labels In, Out, I/O in the net denote port places used during composition of CPN models (Section 6.6.5).



(a) Source

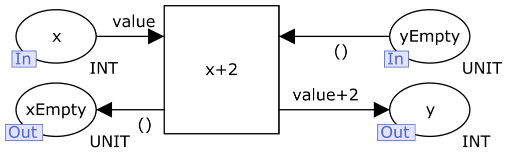


(b) Sink

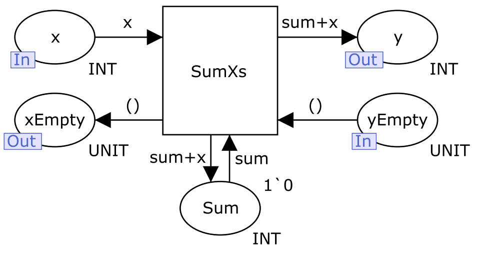
Figure A.1: CPN Models of Source and Sink Components

Data-driven Components

The CPN model of a data-driven component consists of a single transition (realising the component's computation), a pair of places for each data port and, optionally, a place for each state variable. The example in Figure A.2a depicts a component (with one input and one output data port) that increases the values of its inputs by two. Notice how the inscription on one of the outgoing arcs defines the data transformation of the input. Figure A.2b shows a stateful component summing all its inputs; the place Sum holding the state (the sum of inputs already processed) is initialised to zero, and it is read and written during every transition occurrence.



(a) Stateless Component



(b) Stateful Component

Figure A.2: CPN Models of Data-driven Components

Control-driven Components

See Figure 6.27 in Section 6.6.3.

Control-switched Data-driven Components

They have the most complex execution triggering mechanism, and their CPN models are thus most complex. Apart from the computation transition and places for data ports, control port and state, a CPN model of a control-switched data-driven component contains its execution state – enabled or disabled – represented by a pair of places, only one of which contains a token colour at any one time. The arrival of control always moves the token colour from the currently filled place to the other place, thereby switching the component's state. Computation can happen whenever data inputs are available and the Enabled place contains a token colour (Figure A.3).

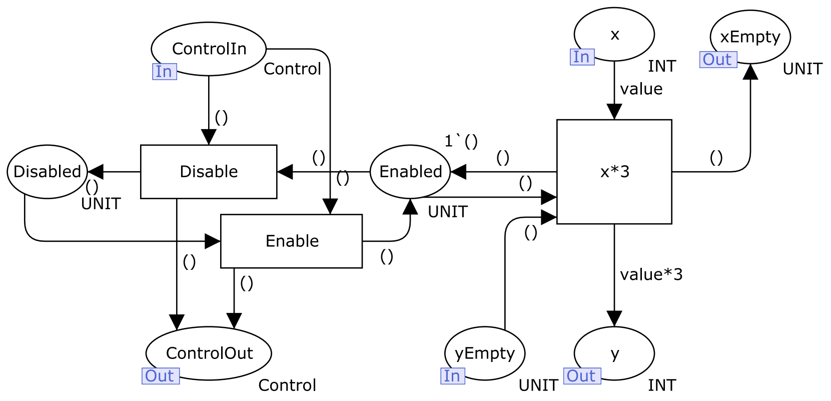
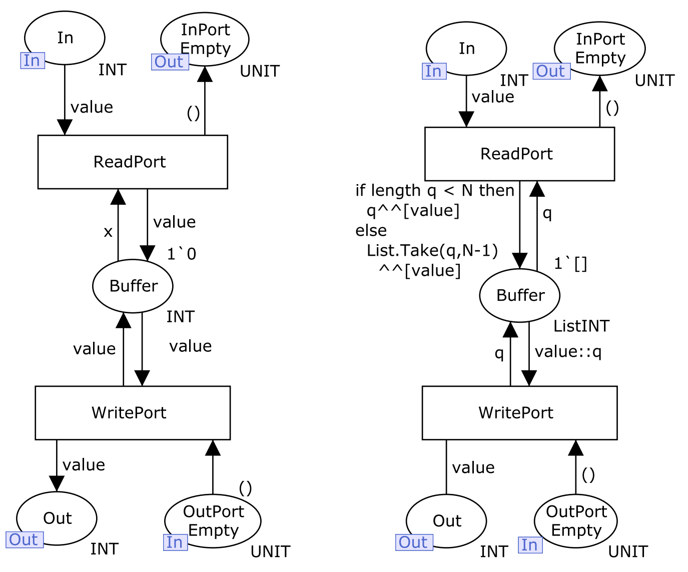


Figure A.3: CPN Model of a Control-switched Data-driven Component

A.1.2 Connectors

Data Channels

They are modelled using two pairs of places representing source and target data ports, a place representing the channel's buffer and two transitions – one for reading the source port and another for writing to the target port. Figure A.4 shows the CPN models of basic channel types (apart from FIFO, which is shown in Figure 6.28a) transferring data between two integer data ports. They differ in the definition of the buffer place and its adjacent arcs. The NDR channel's buffer has the INT (integers) colour set and always contains one value: the initial one is determined by the initial marking, every occurrence of the ReadPort transition rewrites the value and any occurrence of WritePort copies the value to the target data port keeping the buffer's contents unchanged. The FIFO-N channel has the buffer place of the list of integers type, always containing one token colour; ReadPort adds the new value at the end of the list after removing the last value from the list if buffer is about to overflow, whereas WritePort removes the first value in the list.



(a) NDR Channel (b) FIFO with Capacity N

Figure A.4: CPN Models of Data Channels

Data Coordinators

See Figure 6.28b in Section 6.6.3.

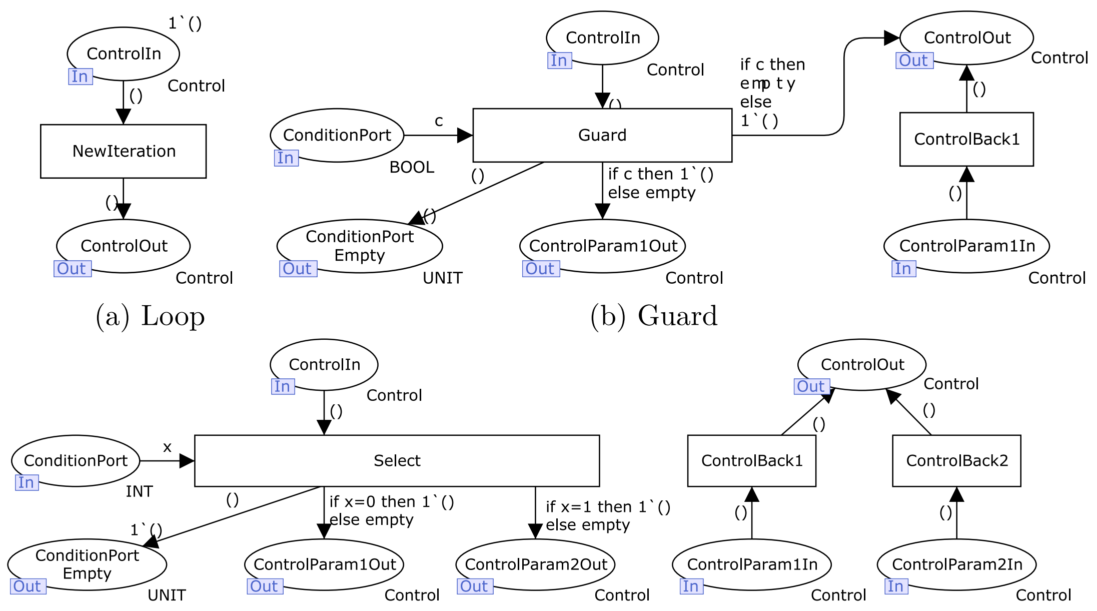
Control Connectors

Control Connectors' CPN models comprise representations of their interface elements (pairs of places corresponding to control ports, control parameters and, possibly, an input data port) and their control routing functionality (several transitions). Figure A.5 lists CPN models of control connector types.

A loop's model (Figure A.5a) is the simplest – it contains two Control places representing incoming and outgoing parts of its only control parameter and a transition whose occurrence accepts and emits a single token colour, thereby modelling start of an iteration of the loop.

A binary sequencer is shown in Figure 6.28c in Section 6.6.3.

The models of a binary guard and selector (Figures A.5b and A.5c) contain pairs of places for a control port, control parameters and their input data port. In both cases, a single transition models the selection behaviour; but while the guard can return control straight back (see the arc from Guard to ControlOut), the selector returns control only after it has returned from the selected control parameter, via ControlBack1 or ControlBack2.



(a) Loop (b) Guard

(c) Selector

Figure A.5: CPN Models of Control Connectors

A.2 Synchronisation Rules

Figure A.6 shows the synchronisation CPN sub-net for a system with two source components and three data flow entities. The execution cycle begins with one token colour in the place CycleStartState, taken by the StartCycle transition, which starts an iteration of the control loop by placing a Control token to ControlFlow-StartState and invokes source components via their synchronisation places. The execution then continues by invoking the rest of data flow entities (StartDF) until data flow becomes idle (DataflowIdleState has marking $\{true\}$) and control loop iteration finishes (ControlFlowFinalState contains the control token), at which point the EndCycle transition moves the token back to CycleStartState to start a new cycle.

The middle part of the sub-net realises the mechanism of detecting idle data flow (data computation and data transfer); it relies on synchronisation places (tagged with I/O in the figure) and global places Restart and DataflowIdleState, whose markings are changed by data flow entities (data channels, coordinators and data-driven components). The idea is that every data flow entity has its own place with the colour set Sync that contains a single token colour, indicating whether the entity can be scheduled for execution (busy) or not (idle). Initially, all synchronisation places contain the busy token colour (added by StartCycle for

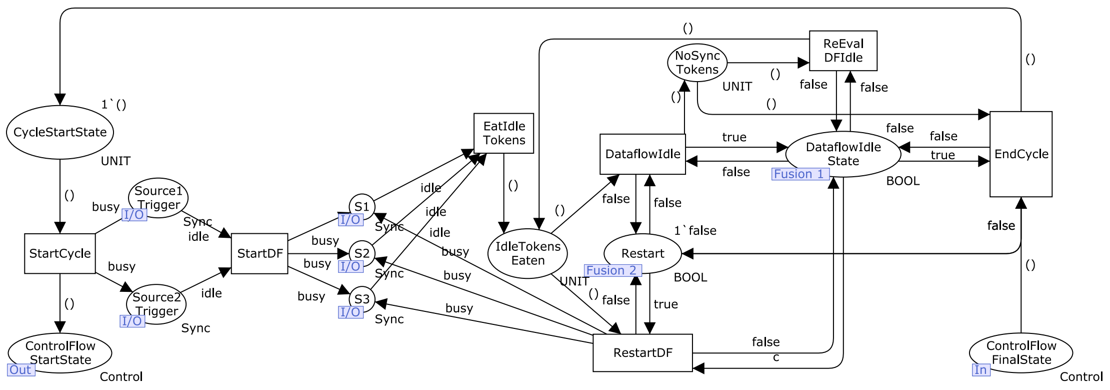
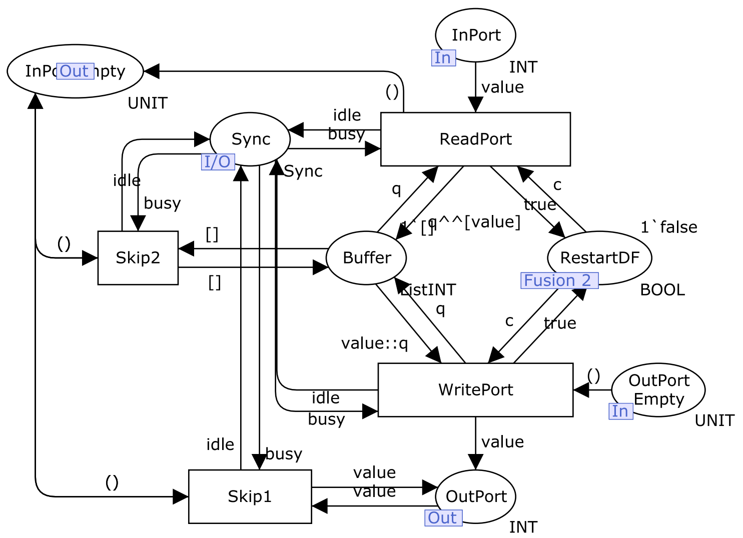


Figure A.6: CPN Sub-net for Synchronisation

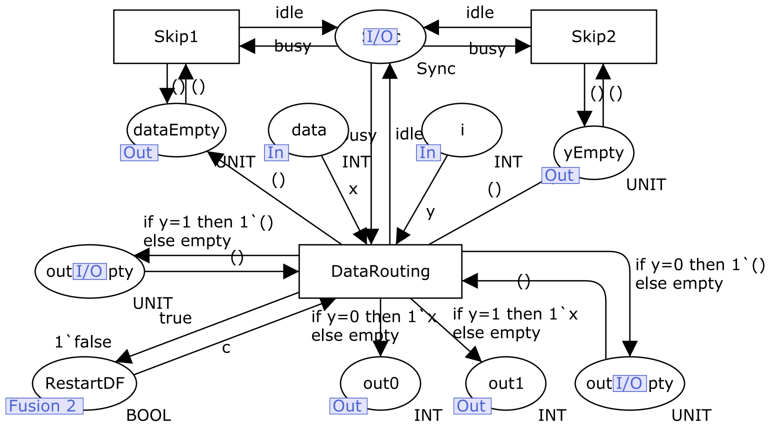
sources and by StartDF for other entities); as the execution progresses, each data flow entity takes the busy token from its synchronisation place, performs any action if it can (e.g., if its inputs are available), and puts the idle token back to its synchronisation place; additionally, if the current action of an entity (e.g., component execution) can result in scheduling another data flow entity for execution, the entity sets the marking in Restart to $\{true\}$. When all synchronisation tokens become idle, their tokens are removed (EatIdleTokens) and if there is any chance of further data flow scheduling (the marking of Restart is $\{true\}$), the synchronisation places are again filled with busy tokens by RestartDF, and the process repeats; if, however, the marking of Restart is $\{false\}$, the system has reached DataflowIdleState.

Figure A.7 shows CPN models of two data flow entities to illustrate the impact of the idle data flow detection mechanism. Compared to their counterparts from Section A.1, the models additionally contain synchronisation places (Sync), places referring to the global place Restart from the synchronisation sub-net (RestartDF) and the transitions named Skip1, Skip2 that occur when an entity cannot perform any action. A FIFO (Figure A.7a) cannot read its source port or write its target port when the source port is empty and the internal buffer is empty, or the target port is full. Notice that either of the FIFO's actions can trigger further data flow, hence RestartDF being set to *true*. A data switch (Figure A.7b) cannot perform its routing action when any of its input ports is empty (Skip1, Skip2); the routing can trigger further data flow execution.

To realise *(SR4)* and *(SR5)*, we need to change the CPN models of control-driven and control-switched data-driven components so that they wait for any inputs computable in the current cycle, before they skip computation during



(a) FIFO



(b) Data Switch

Figure A.7: CPN Models of Two Entities with Synchronisation

triggering or switch the execution state to disabled.

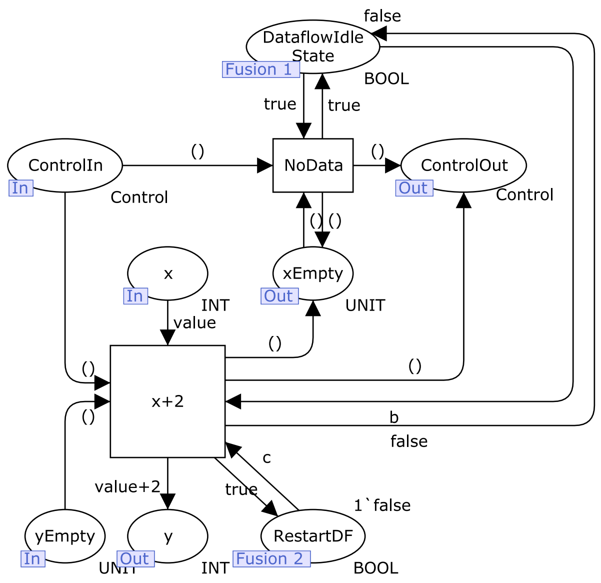


Figure A.8: CPN Model of a Control-driven Component with Synchronisation

A CPN model of a control-driven component is depicted in Figure A.8. The waiting behaviour is expressed by the NoData transition waiting for DataflowIdleState to become true. Another change is setting the global Restart state (from Figure A.6) to *true* during a component's execution, as it may trigger further data flow computation. The execution also causes DataflowIdleState to become $false$; this allows the ReEvalDFIdle transition in the synchronisation sub-net from Figure A.6 to trigger data flow execution after the data flow idle state has been reached.

Figure A.9 shows an updated CPN model of a control-switched data-driven component. Since it is data-driven, its model has undergone the same changes as models in Figure A.7: addition of the synchronisation place, of the transitions that may occur when a component cannot perform any activity (Skip1 and Skip2) and of RestartDF being set to *true* during its computation. Additionally, the transition for disabling the component waits for DataflowIdleState to become true.

A.3 CPN Model of An Example System

To illustrate the composition of CPN models, we use a simple system, whose architecture is depicted in Figure 6.31. The CPN model of the system is shown

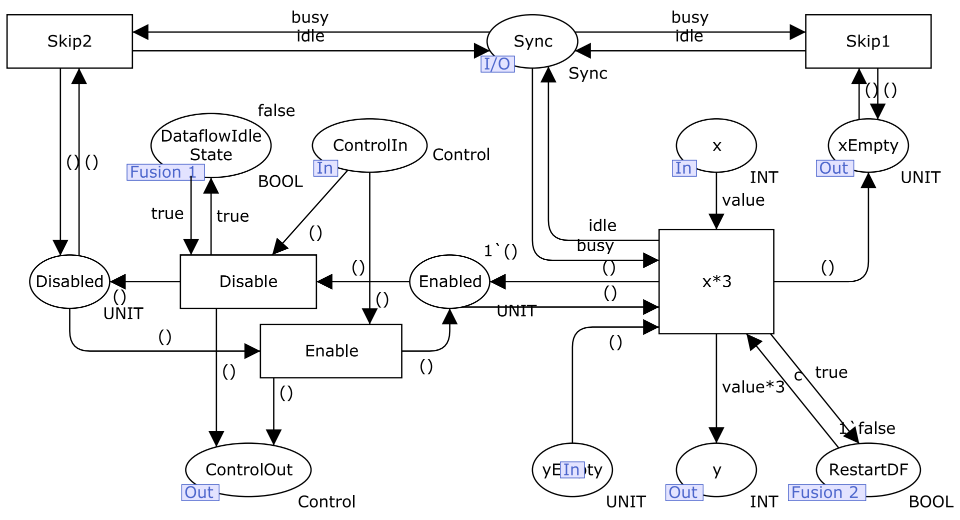


Figure A.9: CPN Model of a Control-switched Data-driven Component with Synchronisation

in Figure A.10. Rectangles with double border are substitution transitions corresponding to pages of component model elements. Notice the similarity between CPN system models and system architectures due to the chosen composition mechanism. Also, note that this CPN model is not deterministic since it is not composed with the synchronisation sub-net. The deterministic version would not be legible in print, due to many arcs joining synchronisation places and the synchronisation sub-net.

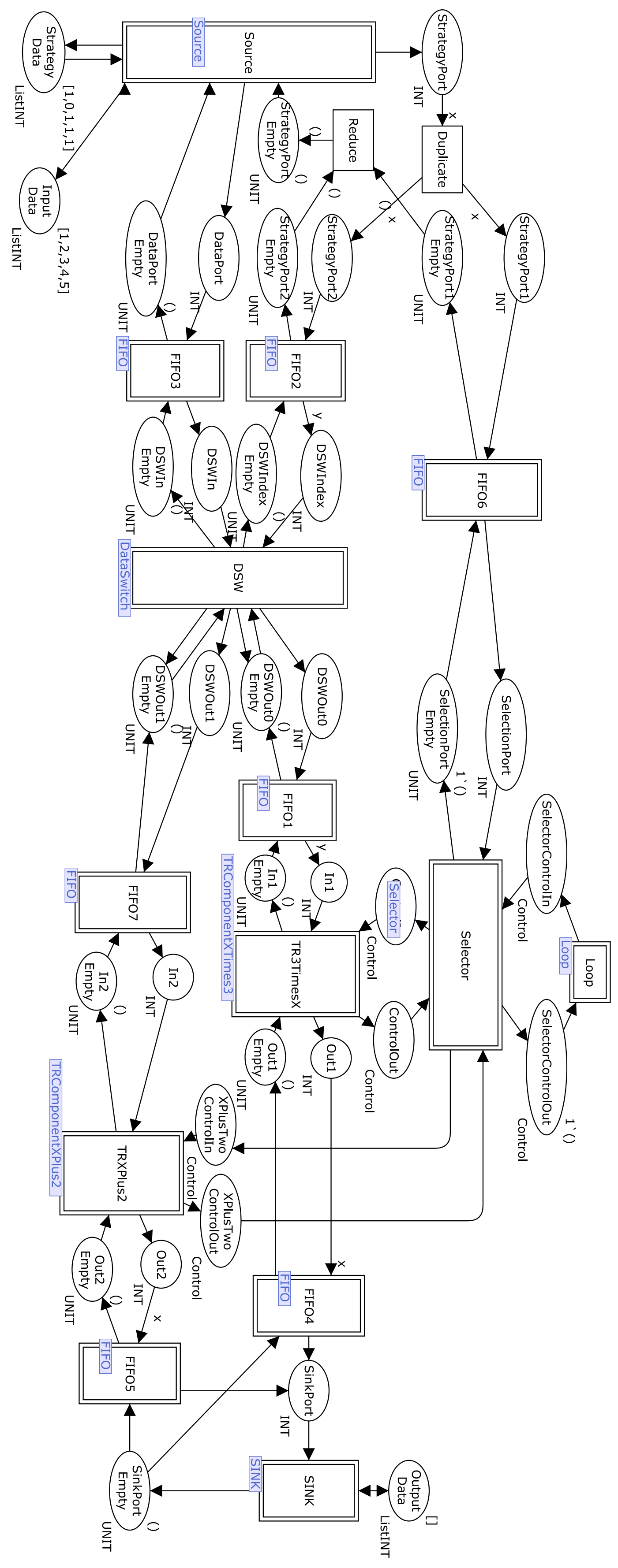


Figure A.10: CPN Model of an Example System

Appendix B

Meta-Model

In this appendix, we present a simplified[1] version of our prototype tool's meta-model, specified in the ECore language[2]:

```
package system : system = 'http://system.hybridmodel.ac.uk/1.0'
{
    class System extends component::WithComments
    {
        property components : component::ComponentInstance[*] { ordered composes };
        property sinks : component::Sink[*] { derived transient volatile };
        property sources : component::Source[*] { derived transient volatile };
        property coordinators : connector::deployment::DeployedDataCoordinator[*] { ordered composes };
        property controlConnectors : connector::deployment::DeployedControlConnector[*] { ordered
            composes };
        property dataConnectors : connector::DataConnector[*] { ordered composes };
    }
}
```

```
package component : component = 'http://component.hybridmodel.ac.uk/1.0'
{
    class Component extends Nameable,WithComments
    {
        property imports : Import[*] { ordered composes };
        property body : xtext::XExpression { ordered composes };
        property ports : DataPort[*] { ordered composes };
        property states : ComponentState[*] { ordered composes };
    }
    class Import
    {
        attribute importedNamespace : String[?] { ordered };
    }
```

[1]For brevity, we omitted OCL invariants and derivation expressions.

[2]Note that the multiplicity of a property or an attribute is denoted by a single character in square brackets: [*] means at least zero, [?] means zero or one, and [+] means at least one; otherwise, ECore's syntax is quite self-explanatory.

```
enum DataType { serializable }
{
   literal int;
   literal string = 1;
   literal boolean = 2;
   literal double = 3;
   literal undefined = −1;
}
abstract class Nameable
{
   attribute name : String[?] { ordered };
}
class DataPort extends Nameable
{
   attribute type : DataType[?] { ordered };
   attribute direction : DataPortDirection[?] { ordered };
}
abstract class ComponentInstance extends Nameable
{
   property component : Component[?] { ordered };
   property ports : PortInstance[∗] { ordered composes };
}
class EDComponentInstance extends ComponentInstance,ControlComposable
{
   attribute initiallyEnabled : Boolean[?] { ordered };
}
class DataComponentInstance extends ComponentInstance;
class Sink extends ComponentInstance
{
   attribute filename : String [?] { ordered };
}
class Source extends ComponentInstance
{
   attribute filename : String [?] { ordered };
}
class TRComponentInstance extends ComponentInstance,ControlComposable;
abstract class ControlComposable { interface };
class SingletonPort extends PortInstance,DataPort;
enum DataPortDirection { serializable }
{
   literal IN;
   literal OUT = 1;
}
class PortInstance
{
   property port : DataPort[?] { ordered };
   attribute portName : String[?] { ordered derived readonly transient volatile }
}
abstract class WithComments
{
   attribute comment : String[?] { ordered };
}
class ComponentState extends Nameable
{
        attribute type : DataType[?] { ordered };
        property defaultValue : xtext::XExpression { ordered composes };
    }
}



package connector : connector = 'http://connector.hybridmodel.ac.uk/1.0'
{
    package design : design = 'http://design.connector.hybridmodel.ac.uk/1.0'
    {
        abstract class DesignControlConnector
        {
            property controlParameter : cctemplate::RequiredControlComposable { ordered };
        }
        class DesignSequencer extends DesignControlConnector,Sequencer;
        class DesignSelector extends DesignControlConnector,Selector;
        abstract class DesignDataCoordinator
        {
            property controlData : component::SingletonPort { ordered composes };
            property inData : component::SingletonPort { ordered composes };
            property outData : component::SingletonPort { ordered composes };
        }
        class DesignDataJoin extends DataJoin,DesignDataCoordinator;
        class DesignDataSwitch extends DesignDataCoordinator,DataSwitch;
        class DesignDataGuard extends DataGuard,DesignDataCoordinator;
        class DesignCompositeConnector extends CompositeConnector
        {
            property portBindings : DesignPortBinding[*] { ordered composes };
            property controlPortBindings : DesignControlPortBinding[*] { ordered composes };
        }
        class DesignPortBinding
        {
            property inner : cctemplate::RequiredPortInstance[?] { ordered };
            property outer : cctemplate::RequiredPortInstance[?] { ordered };
            attribute name : String[?] { ordered derived readonly transient volatile }
        }
        class DesignControlPortBinding
        {
            property inner : cctemplate::RequiredControlComposable[?] { ordered };
            property outer : cctemplate::RequiredControlComposable[?] { ordered };
            attribute name : String[?] { ordered derived readonly transient volatile }
        }
    }

    package deployment : deployment = 'http://deployment.connector.hybridmodel.ac.uk/1.0'
    {
        abstract class DeployedControlConnector
        {
            property parameters : ControlParameter[+] { ordered composes };
        }
        class ControlParameter
        {
            attribute order : ecore::EInt[?] { ordered };
    property controlTarget : component::ControlComposable[?] { ordered };
}
class DeployedLoop extends DeployedControlConnector,Loop;
class DeployedGuard extends Guard,DeployedControlConnector;
class DeployedSelector extends DeployedControlConnector,Selector;
class DeployedSequencer extends DeployedControlConnector,Sequencer;
class DeployedCompositeConnector extends DeployedControlConnector,CompositeConnector
{
    property portBindings : PortBinding[*] { ordered composes };
    property controlPortBindings : ControlPortBinding[*] { ordered composes };
    property participants : Participant [*]  { ordered composes };
}
abstract class DeployedDataCoordinator
{
    property controlData : component::SingletonPort[?] { ordered composes };
}
class DeployedDataGuard extends DeployedDataCoordinator,DataGuard
{
    property inData : component::SingletonPort { ordered composes };
    property outData : component::SingletonPort { ordered composes };
}
class DeployedDataSwitch extends DataSwitch,DeployedDataCoordinator
{
    property inData : component::SingletonPort { ordered composes };
    property outData : component::SingletonPort[+] { ordered composes };
}
class DeployedDataJoin extends DeployedDataCoordinator,DataJoin
{
    property inData : component::SingletonPort[+] { ordered composes };
    property outData : component::SingletonPort { ordered composes };
}
abstract class PortBinding
{
    property inner : cctemplate::RequiredPortInstance[?] { ordered };
    attribute name : String[?] { ordered derived readonly transient volatile }
    property participant : Participant [?]  { ordered };
}
class ControlPortBinding
{
    property inner : cctemplate::RequiredControlComposable[?] { ordered };
    property outer : component::ControlComposable[?] { ordered };
    attribute name : String[?] { ordered derived readonly transient volatile }
    property participant : Participant [?]  { ordered };
}
class ParticipantData
{
    attribute value : String [?]  { ordered };
    property type : cctemplate::RoleData[?] { ordered };
}
class Participant
{
    property data : ParticipantData[*] { ordered composes };
    property role : cctemplate::Role [?]  { ordered };
    attribute order : ecore :: EInt { ordered };
    }
    class SinglePortBinding extends PortBinding
    {
        property outer : component::PortInstance[?] { ordered };
    }
    class InMultiPortBinding extends PortBinding
    {
        property outer : PortInstanceMapping[+] { ordered composes };
    }
    class OutMultiPortBinding extends PortBinding
    {
        property outer : component::PortInstance[+] { ordered };
    }
    class PortInstanceMapping
    {
        property from : component::PortInstance { ordered };
        property to : component::PortInstance { ordered };
        attribute name : String[?] { ordered derived readonly transient volatile }
    }
}

package cctemplate : cctemplate = 'http://cctemplate.connector.hybridmodel.ac.uk/1.0'
{
    class CompositeConnectorTemplate extends component::Nameable,component::WithComments
    {
        property requiredPorts : RequiredPortInstance[*] { ordered composes };
        property subConnectors : DataConnector[*] { ordered composes };
        property controlSubconnectors : ComposableControlConnector[*] { ordered composes };
        property coordinators : DataCoordinator[*] { ordered composes };
        property requiredControlPorts : RequiredControlComposable[*] { ordered composes };
        property roles : Role[*] { derived readonly transient volatile }
        property ownRoles : Role[*] { composes };
        attribute constraint : String [?] { ordered };
    }
    class RequiredPortInstance extends component::Nameable,component::PortInstance
    {
        attribute reqType : component::DataType[?] { ordered };
        property sameTypeAs : RequiredPortInstance[?] { ordered };
        attribute reqDirection : component::DataPortDirection { ordered };
        property role : Role[?] { ordered };
        attribute multiPort : Boolean[?] { ordered };
    }
    class RequiredControlComposable extends component::Nameable,component::ControlComposable
    {
        property role : Role[?] { ordered };
    }
    class Role extends component::Nameable
    {
        property config : RoleData[*] { ordered composes };
        attribute multiplicityLowerBound : ecore::EInt[?] = '1' { ordered };
        attribute multiplicityUpperBound : ecore::EInt[?] = '1' { ordered };
    }
    class RoleData extends component::Nameable
    {
            property dataSpec : RequiredPortInstance[?] { ordered };
        }
    }
    abstract class ControlConnector;
    abstract class DataConnector;
    abstract class Loop extends ControlConnector;
    abstract class ComposableControlConnector extends ControlConnector,component::ControlComposable
    {
        property associatedRole : cctemplate::Role[?] { ordered };
    }
    abstract class Guard extends ComposableControlConnector
    {
        property condition : component::SingletonPort { ordered composes };
    }
    abstract class Selector extends ComposableControlConnector
    {
        property selection : component::SingletonPort { ordered composes };
    }
    abstract class Sequencer extends ComposableControlConnector;
    class AtomicDataConnector extends DataConnector
    {
        attribute type : AtomicDataConnectorType[?] { ordered };
        attribute initialValue : String [?] { ordered };
        property source : component::PortInstance { ordered };
        property target : component::PortInstance { ordered };
        attribute initialValueType : component::DataType[?] { ordered };
    }
    enum AtomicDataConnectorType { serializable }
    {
        literal FIFO;
        literal FIFO1 = 1;
        literal NDR1 = 2;
    }
    abstract class CompositeConnector extends DataConnector,ComposableControlConnector
    {
        property template : cctemplate::CompositeConnectorTemplate[?] { ordered };
        attribute name : String[?] { ordered derived readonly transient volatile }
    }
    abstract class DataCoordinator
    {
        property associatedRole : cctemplate::Role[?] { ordered };
    }
    abstract class DataJoin extends DataCoordinator;
    abstract class DataSwitch extends DataCoordinator;
    abstract class DataGuard extends DataCoordinator;
}
```

Appendix C

Model Transformations in Henshin

In this appendix, we show sample Henshin transformation rules that show the feasibility of defining (and formalising) model transformations in our prototype tool using graph rewriting systems. In particular, we present several rules defining some steps of the transformation that creates the composition structure of deployed connector instances (see Section 8.4.4).

Figure C.1 shows rules realising the deployment of a data switch design instance.[1] The deployDataSwitch rule takes a role as its input parameter, finds an instance of a design data switch associated with that role and creates its corresponding deployed data switch instance. The ports of the design instance are reconnected to belong to the deployed instance, and the design instance is deleted. The newly created deployed data switch and its output data port are output as parameters ds and outPort of the rule, respectively. Additionally, a new instance of the Henshin built-in class Trace is created, its name attribute is set to the name of the role, and it is returned via the counter parameter; its purpose is to help distinguish ports associated with different participants (the role of the partMap mapping from the transformation's pseudocode).

The duplicateOutports rule adds another output port to ds. More interestingly, it also creates an instance of Trace that maps the original output port outPort to its newly created duplicate (realisation of the map mapping from the transformation's pseudocode) and tags the new port by means of yet another instance

[1]For simplicity, both presented transformations are in-place replacements, and we also do not show all attribute values.

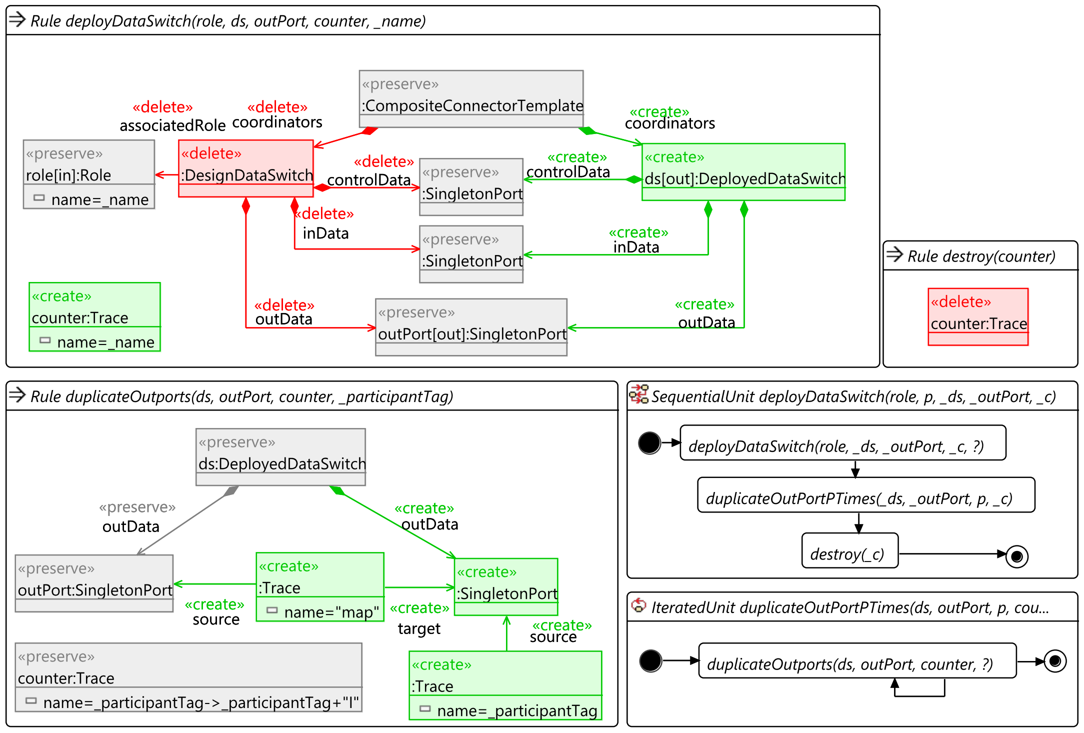

Rule deployDataSwitch(role, ds, outPort, counter, _name)
«preserve» :CompositeConnectorTemplate
«delete» associatedRole
«delete» coordinators
«create» coordinators
«preserve» role[in]:Role
name=_name
«delete» :DesignDataSwitch
«delete» controlData
«preserve» :SingletonPort
«create» controlData
«create» ds[out]:DeployedDataSwitch
«delete» inData
«preserve» :SingletonPort
«create» inData
«create» counter:Trace
name=_name
«delete» outData
«preserve» outPort[out]:SingletonPort
«create» outData
Rule destroy(counter)
«delete» counter:Trace
Rule duplicateOutports(ds, outPort, counter, _participantTag)
«preserve» ds:DeployedDataSwitch
«preserve» outData
«create» outData
«preserve» outPort:SingletonPort
«create» :Trace
name="map"
«create» :SingletonPort
«create» source
«create» target
«create» source
«preserve» counter:Trace
name=_participantTag->_participantTag+"I"
«create» :Trace
name=_participantTag
SequentialUnit deployDataSwitch(role, p, _ds, _outPort, _c)
deployDataSwitch(role, _ds, _outPort, _c, ?)
duplicateOutPortPTimes(_ds, _outPort, p, _c)
destroy(_c)
IteratedUnit duplicateOutPortPTimes(ds, outPort, p, cou...
duplicateOutports(ds, outPort, counter, ?)

Figure C.1: Transformation Rules for Deploying a Design Data Switch

of Trace to associate it with a participant corresponding to the tag. We see that counter serves as a counter variable in this process.

To deploy a design data switch instance, the deployDataSwitch rule has to be applied first, followed by p applications of duplicatePorts and by the destroy rule removing the counter instance. This coordination is defined by the sequential and iterated transformation units at the bottom-right of Figure C.1.

The rule addDataChannel in Figure C.2 uses the mapping created by duplicate-Outports (and other rules) to construct data channels between ports of duplicated entities. In particular, it creates a new instance of a data channel between two port instances that are mapped from a pair of port instances connected with a data channel. The AtomicDataConnector class with the «forbid» stereotype represents a negative matching condition; i.e., a sub-graph does not match the left hand side of the rule if there exists a match of the elements annotated with «forbid». In this case, the constraint ensures that no data channel will be created between a pair of ports already connected with a data channel. Additionally, the rule only allows the ports that are tagged by the same participant tag to be connected by a data channel (see the two Trace instances at the bottom). If we

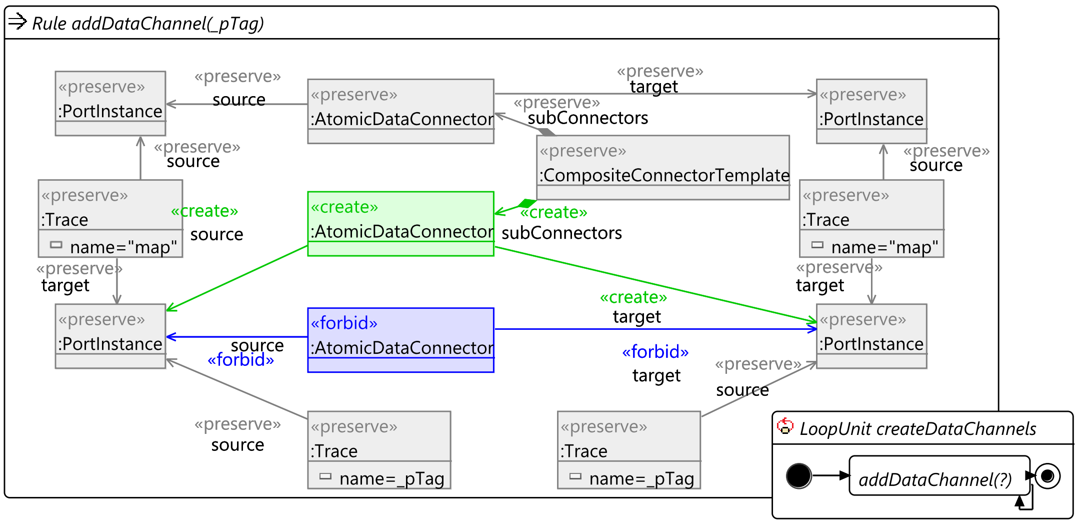


Figure C.2: Transformation Rule for Duplicating Data Channels

compare the constraints expressed by the rule with the condition on line 32 in the transformation's pseudocode in Section 8.4.4, we see that the rule does not deal with all possible cases. The cases in which at least one of the ports to be connected is not associated with any role (and therefore has no participant tag) have to be solved by other rules.

One application of the rule creates one new data channel. To duplicate all data channels, the rule (together with other rules solving complementary cases) has to be applied repeatedly until no more applications of the rule are possible (its left-hand side cannot be matched against the input graph). Henshin provides the Loop transformation unit for this purpose – see createDataChannels at the bottom-right of Figure C.2.

Appendix D

Climate Control System Design in Other Component Models

In this appendix, we show the architectures of the climate control system (the system we used in our case study in Chapter 10) constructed in the four component models with which we compared our approach in Section 11.1: ProCom, Scade, Simulink and UML 2.0.

D.1 ProCom

Figure D.1 shows a composite component realising the climate control system in ProCom. It models the behaviour of the system in one reactive cycle (one iteration of the loop in Figure 10.10). The component consists of four atomic subcomponents, whose names and roles correspond to the components in the climate control system's architecture created in our model (Figure 10.10). However,

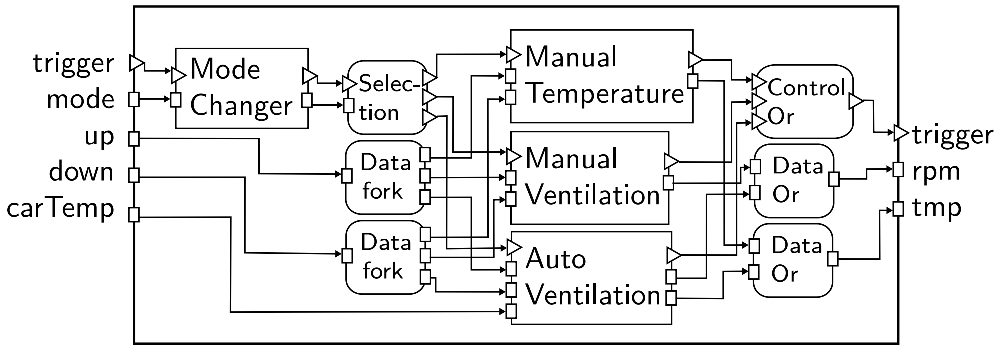


Figure D.1: Climate Control Component in ProCom

there are some differences in component semantics. Unlike our model, ProSave has only one type of components that are control-triggered (as our control-driven components). As a result, the components modelling the system's functionality in one of its three modes (ManualTemperature, ManualVentilation and AutoVentilation) have to be triggered each time the system is in the respective mode and their parent composite component receives control. This simplifies the required control flow and data flow routing: it leads to the behaviour equivalent to that of the Trigger-Strategy interaction pattern (see Figure 8.5).

The interaction pattern in Figure D.1 is represented by six connectors: Selection chooses a subcomponent to be triggered; Data forks distribute inputs to all Mode components[1]; Control Or aggregates control outputs of the Mode components (i.e., waits for the active Mode component to finish); and Data Ors aggregate Mode components' outputs.

D.2 Scade

We present two Scade implementations of the climate control system that illustrate two different component representations of interaction patterns in Scade.

Figure D.2 shows the first implementation. The system is represented by a composite component comprised of three subcomponents that correspond to the three modes of the climate control systems. The subcomponents' behaviour is similar to their counterparts in ProCom and our model. Their parent composite component is defined as a Scade state machine with three states corresponding to the three system modes. Each of the subcomponents is used to process their parent component's inputs when their respective state is active.

An alternative Scade implementation of the climate control system, shown in Figure D.3, separates the definition of interaction and computation. The separation is possible, because Scade, like ProCom, is based on the pipe-and-filter interaction style. Even the resulting architectures are similar (compare Figure D.3 with Figure D.1).

The composite component that constitutes the climate control system contains

[1]Connections in ProCom have no buffers and we assume that old data on component ports are overwritten to ensure that Mode components process up-to-date inputs.

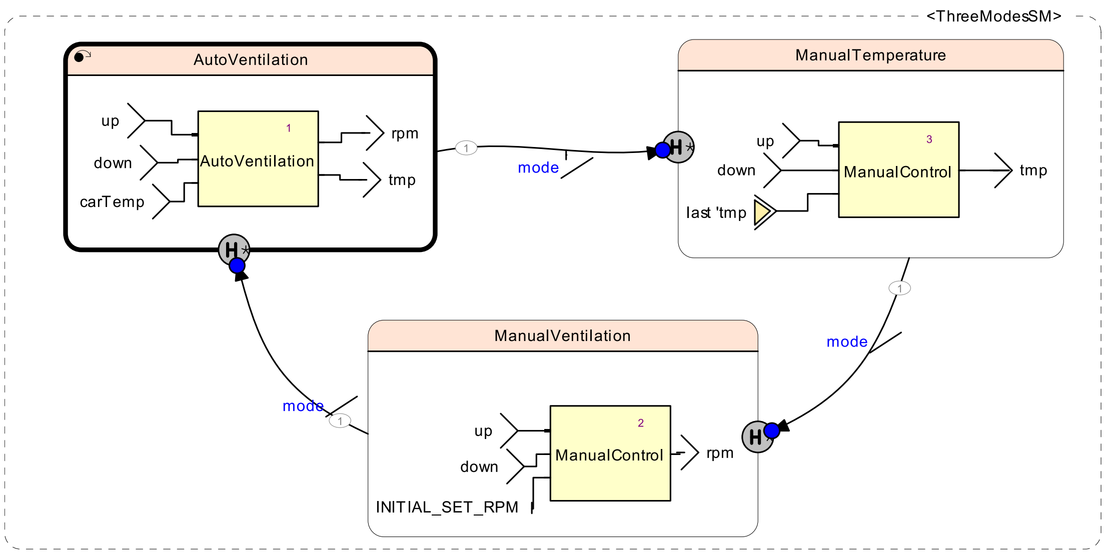


Figure D.2: Climate Control Component as a State Machine in Scade

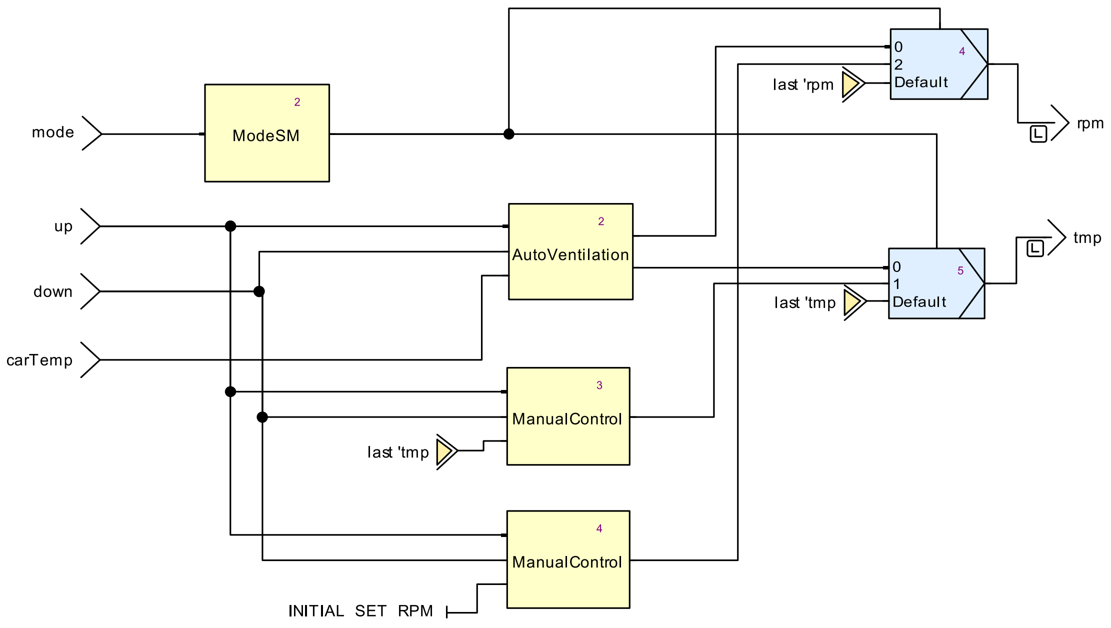


Figure D.3: Climate Control Component with Separate Mode Switching in Scade

four subcomponents: ModeSM for computing the new mode and three other components implementing the system's behaviour in each of its modes. The subcomponents are composed by means of data connections. Additionally, their outputs are merged by two 'case' operators (the two darker rectangles in Figure D.3). Unlike our model, subcomponents cannot be disabled and thus produce results all the time; the system outputs the results selected by the case operators.

D.3 Simulink

A Simulink subsystem implementing the behaviour of one reaction of the climate control system is depicted in Figure D.4. The implementation is very similar to the Scade one (compare with Figure D.3): the three subsystems realising the system modes' functionality as well as the subsystem computing the current mode have the same data ports; a similar mechanism is used to merge the outputs of the mode components to obtain the system's outputs; and the mode switching interaction pattern is represented by means of a number of data connections and Simulink blocks.

However, the interaction pattern's implementation in Simulink embodies more complex behaviour than in the Scade's implementation, where it just distributes inputs to all mode components and selects the output of the one that corresponds to the current mode. Since we have represented mode components as enabled subsystems (note the square wave icons within the component boxes in Figure D.4), we can also model the active subsystem selection. This is realised by splitting the output data flow of ModeChanger, which corresponds to the current mode, and testing whether its value equals the respective mode value; the results of these tests (outputs of the blocks R1, R2, and R3) are passed to Enabled ports of the mode components and determine whether a given subsystem computes its outputs. This also makes the two switches (Switch and Switch1) logically redundant, because just one subsystem is enabled at any one time; however, they cannot be removed due to the Simulink's constraint of only one incoming signal to an output port (see ports tmp and rpm in the figure).

Figure D.5 shows a Simulink implementation of the ManualTemperature component, which is an example of an enabled subsystem (note the Enable port in the upper-left part of the figure). The component also contains a special 'Memory' block, named temp, which delays outputting of its current input until the next

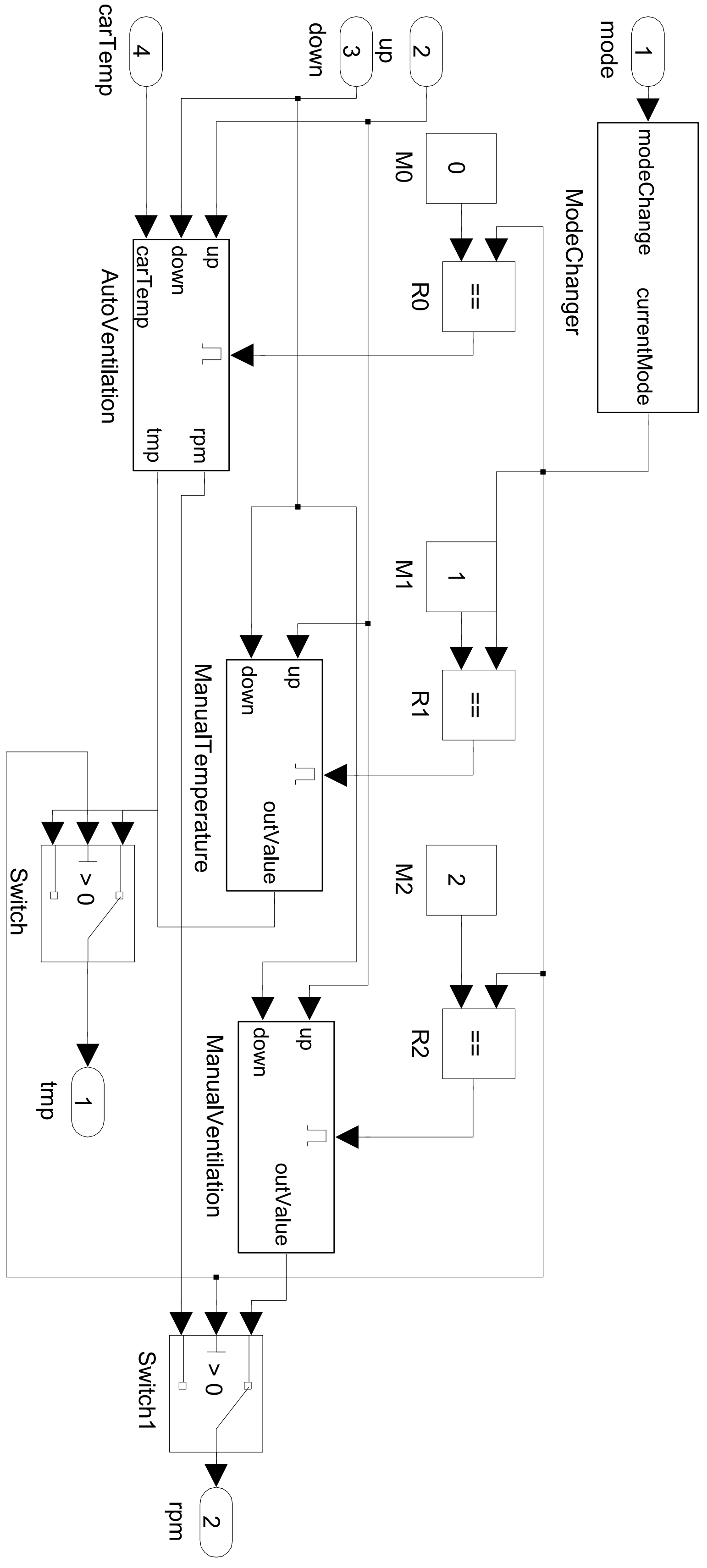


Figure D.4: Climate Control Component in Simulink

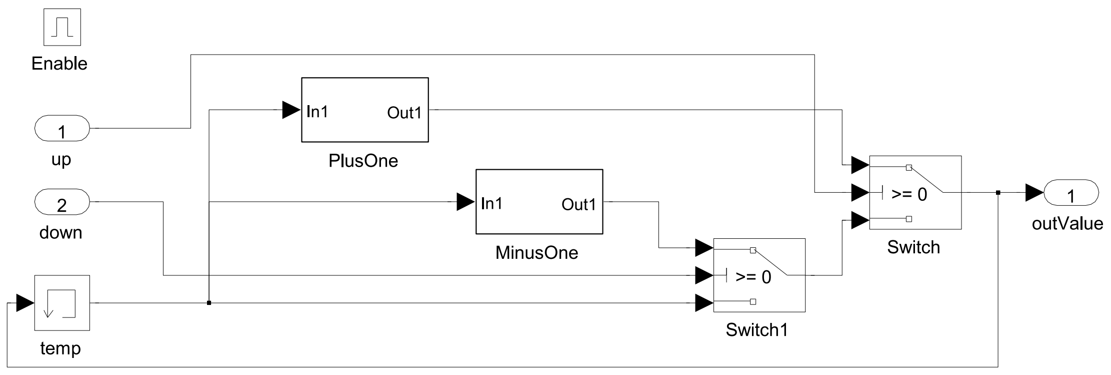


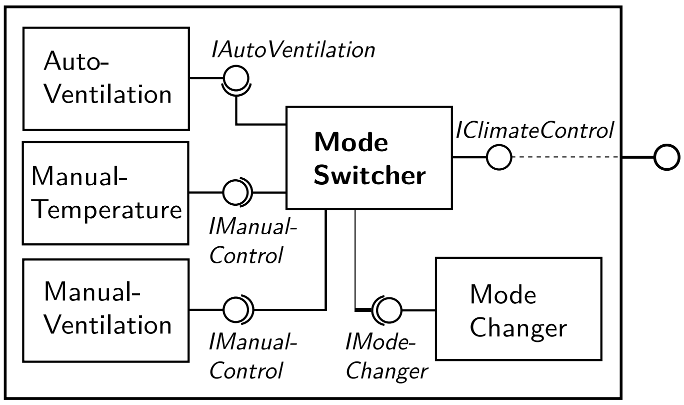
Figure D.5: ManualTemperature in Simulink

simulation step – effectively modelling the internal state of the subsystem (the currently set temperature).

D.4 UML 2.0

Figure D.6 shows a representation of the mode switching pattern within a UML 2.0 composite component, implementing the functionality of the climate control system in one reactive cycle. Because we designed the component to maximally separate computation and interaction, it consists of three subcomponents responsible for the functionalities of the three modes of operation of the system, a subcomponent computing the next active mode and a subcomponent, called ModeSwitcher, that realises the mode switching interaction pattern. In fact, the design conforms to the Mediator design pattern [44], with ModeSwitcher playing the role of the mediator and other subcomponents playing the role of Colleagues.

The ModeSwitcher component, which represents the mode switching interaction pattern, has one provided port and one required port for each component it coordinates. Since it is responsible for distributing and collecting data from all the coordinated components, the signature of its regulate service aggregates all the manipulated data. Its provided port is delegated to become the provided port of the whole climate control composite component. The exact behaviour of the pattern is defined in the implementation of the component, shown in Figure D.7 (as pseudocode).



```
interface IAutoVentilation {
    regulate(up: bool, down: bool, carTemp: int): int_pair
}
interface IManualControl {
    regulate(up: bool, down: bool): int
}
interface IModeChanger {
    nextMode(mode: bool): int
}
interface IClimateControl {
    regulate(mode: bool, up: bool, down: bool, carTemp: int): int_pair
}
```

Figure D.6: Climate Control Component in UML 2.0

```
component ModeSwitcher {
    requires auto : IAutoVentilation
    requires temp : IManualControl
    requires vent : IManualControl
    requires changer : IModeChanger
    provides IClimateControl {
        regulate(mode: bool, up: bool, down: bool, carTemp: int): int_pair {
            nextMode = changer.nextMode(mode)
            switch (nextMode) {
                case 0: return auto.regulate(up, down, carTemp)
                case 1: return int_pair(temp.regulate(up, down), null)
                case 2: return int_pair(null, vent.regulate(up, down))
            }
        }
    }
}
```

Figure D.7: Implementation of ModeSwitcher

Appendix E

Examples of Design Pattern Formalisation Techniques

In this appendix, we give examples of methods formalising design pattern solutions, which we analysed in Section 11.3. We present one example for each group of approaches identified in Section 11.3.1.

E.1 Logic Constraints

BPSL (Balanced Pattern Specification Language) [116] uses first-order logic to describe structural aspects of patterns and temporal logic of actions to define behaviour of pattern participants. BPSL pre-defines sets of entities comprising object-oriented designs (classes, attributes and methods) and relations describing their structural relationships (e.g., Defined-in(attribute, class) or Reference-to-one (class1, class2)). The behavioural part of patterns si defined in terms of temporal relations and a set of actions that specify changes to the temporal relations in time.

For instance, in the Observer pattern, the $Attached(subject, listener)$ temporal relation denotes that the listener is registered in the list of entities receiving updates when the subject's state changes; the actions $Attach(subject, listener)$ and $Detach(subject, listener)$ add or remove the pair $(subject, listener)$ to/from the relation.

A pattern is defined as a formula of the following form:

$$\exists(x_1, \ldots, x_n)(\wedge_i R_i \bigwedge \vee_j A_j),$$

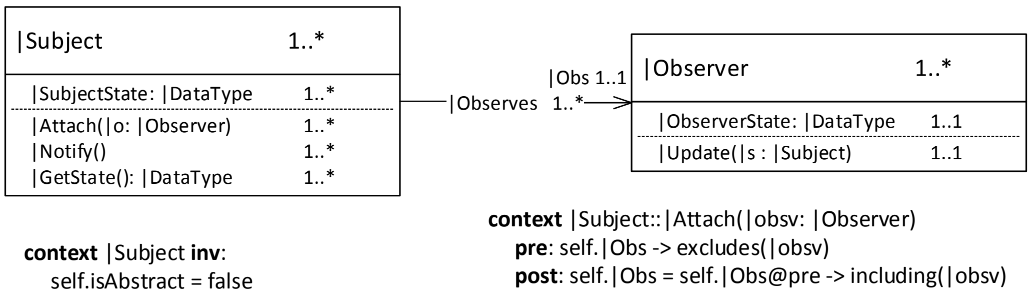
where $x_1, \ldots, x_n$ denote pattern parameters – classes, methods and other entities comprising the pattern specification; R_i denote structural and temporal relations among pattern parameters; and A_j are actions that modify the temporal relations in time. For example, the formula in Figure E.1 defines the Observer pattern.

$$
\begin{array}{r}
\exists\, subject, observer \in Classes \\
\exists\, subject\text{-}state, observer\text{-}state \in Attributes \\
\exists\, attach, detach, notify, get\text{-}state,\ set\text{-}state, \\
update \in Methods \\
\exists\, o, s \in Objects \\
\exists\, d \in Values \\
\hline
Defined\text{-}in(subject\text{-}state, subject) \wedge \\
Defined\text{-}in(observer\text{-}state, observer) \wedge \\
Defined\text{-}in(attach, subject) \wedge \\
Defined\text{-}in(detach, subject) \wedge \\
Defined\text{-}in(notify, subject) \wedge \\
Defined\text{-}in(set\text{-}state, subject) \wedge \\
Defined\text{-}in(get\text{-}state, subject) \wedge \\
Defined\text{-}in(update, observer) \wedge \\
Reference\text{-}to\text{-}one(observer, subject) \wedge \\
Reference\text{-}to\text{-}many(subject, observer) \wedge \\
Argument(observer, attach) \wedge \\
Argument(observer, detach) \wedge \\
\hline
Invocation(set\text{-}state, notify) \wedge \\
Invocation(notify, update) \wedge \\
Invocation(update, get\text{-}state) \wedge
\end{array}
\qquad
\begin{array}{r}
Instance(s, subject) \wedge \\
Instance(o, observer) \wedge \\
Attached(subject[0 \ldots 1], observer[*]) \wedge \\
Updated(observer[*], subject[0 \ldots 1]) \\
\hline
\vee \\
\mathbf{Attach(s, o)}: \\
\neg Attached(s, o) \rightarrow Attached'(s, o) \\
\vee \\
\mathbf{Detach(s, o)}: \\
Attached(s, o) \rightarrow \neg Attached'(s, o) \\
\vee \\
\mathbf{Notify(s, d)}: \\
true \rightarrow \\
(\neg Updated'(s, observer) \wedge \\
s.subject\text{-}state' = d) \\
\vee \\
\mathbf{Update(s, o)}: \\
Attached(s, o) \wedge \neg Updated(s, o) \rightarrow \\
(Updated'(s, o) \wedge \\
o.observer\text{-}state' = s.subject\text{-}state)
\end{array}
$$

Figure E.1: The Observer Pattern Specified in BPSL

E.2 Diagrammatic Constraints

Role-based Modelling Language (RBML) [43] represents patterns using customised UML diagrams. Traditional UML diagrams are meant to represent a single design (e.g., a class diagram models a set of particular classes). However, pattern solutions define a family of complying designs (of pattern instances). RBML addresses this shortcoming of UML diagrams in modelling pattern solutions by introducing the concepts of roles. Roles semantically correspond to sets of design elements of the same type (e.g., sets of classes or associations), whose elements are unknown at the time when roles are created. By including roles in a UML diagram, RBML yields a ‘meta-diagram’ representing a number of designs that correspond to different sets of design elements assigned to the roles present in the meta-diagram. RBML specifies pattern solutions using these meta-diagrams and Object Constraint Language.



|Subject 1..*
|SubjectState: |DataType 1..*
|Attach(|o: |Observer) 1..*
|Notify() 1..*
|GetState(): |DataType 1..*
|Obs 1..1
|Observes 1..*
|Observer 1..*
|ObserverState: |DataType 1..1
|Update(|s : |Subject) 1..1
context |Subject inv:
self.isAbstract = false
context |Subject::|Attach(|obsv: |Observer)
pre: self.|Obs -> excludes(|obsv)
post: self.|Obs = self.|Obs@pre -> including(|obsv)

Figure E.2: An RBML Meta-class Diagram

Figure E.2 depicts the structural description of the Observer pattern using a meta-class diagram. All entities in the diagram with ‘|’ prepended to their name are roles. Each role also has an associated multiplicity, which denotes the admissible size of the set of design elements associated with the role. For instance, |Subject is a role that can be associated with at least one class, |Notify is a role that can be associated with at least one method, etc. It is easy to see how the RBML meta-class diagram in the figure (representing the structural part of the pattern solution) can be instantiated into a number of UML class diagrams (particular instances of the pattern solution).

E.3 Design Transformations

Herranz et al. [57] define patterns as class operators in SLAM-SL, an object-oriented specification language. Figure E.3 shows an excerpt from the definition of a transformation introducing the Composite design pattern. The operator is defined as the method apply in the class Composite, which takes a set of classes (denoted as leaves in the code) as its parameter. The input classes have to fulfil the precondition (line 4) – they have to be a non-empty set and must have

```
class Composite inherits DPattern <Unit >
  public observer apply ([Class]): [Class]
    let common_meths = {m with cl in leaves | m in cl methods } with m in cl. methods
    pre (not leaves isEmpty ) and (not common_meths isEmpty )
     call apply (leaves)
    post result = [component, composite] + [c\inheritance<−(component) with c in leaves]
```

Figure E.3: Transformation Introducing the Composite Pattern in SLAM-SL

some method declarations in common – before the operator can be applied. The approach relies on reflection to inspect and modify the input design models. In our example, the operator creates two new classes (variables component and composite, whose definitions are not shown but conform to the Composite pattern) and modifies the input classes to inherit from component.

E.4 Reusable Implementations

We give an example of the Observer implementation in AspectJ (an aspect-oriented extension of Java) by Hannemann and Kisczales [52]. The basic idea of the approach is that a pattern solution's implementation usually crosscuts code of multiple classes (participants). Pattern solutions are thus defined as abstract aspects; instantiated patterns are concrete aspects that extend the abstract aspects. Figure E.4 shows the abstract aspect defining the generic part of Observer's solution. The participant roles are represented as interfaces (lines 2-3). The aspect also implements the subject's management of registered observers via a hash map (lines 5-8) and declares abstract pointcuts[1] (line 10) and methods (line 11) to be defined by the concrete aspects corresponding to particular pattern instances. Finally, it defines an after advice which injects calls to the updateObserver method in all locations specified by the subjectChange pointcut.

Figure E.5 gives an example of a concrete aspect, called ConcreteObserver, representing a particular instance of the Observer pattern. The aspect extends the abstract aspect defined in Figure E.4. It associates concrete classes (Point and Screen) with pattern roles by making them implement the interfaces defined by the parent aspect (lines 2-3) and defines the inherited abstract pointcut subjectChange and the method updateObserver. The pointcut is set to all code locations from

[1]In AspectJ terminology, a pointcut denotes a set of code locations (joinpoints) in which an aspect can change behaviour using so-called advices.

```
public abstract aspect ObserverProtocol {
    protected interface Subject { }
    protected interface Observer { }

    private WeakHashMap perSubjectObservers ;
    protected List getObservers (Subject s) { ... }
    public void addObserver (Subject s, Observer o) { ... }
    public void removeObserver (Subject s, Observer o) { ... }

    abstract protected pointcut subjectChange (Subject s);
    abstract protected void updateObserver (Subject s, Observer o);
    after (Subject s): subjectChange (s) { /* for each observer, call updateObserver */ }
}
```

Figure E.4: Abstract Aspect Realising the Observer Pattern in AspectJ

which the setColour method is called, and the method is defined to refresh the registered screen observers if a point's colour changes (not shown in the code).

```
public aspect ColorObserver extends ObserverProtocol {
    declare parents : Point implements Subject ;
    declare parents : Screen implements Observer ;
    protected pointcut subjectChange (Subject s):
        call (void Point.setColour(Colour)) && target(s);
    protected void updateObserver (Subject s, Observer o) { ... }
}
```

Figure E.5: Concrete Aspect Realising an Observer Instance

Appendix F

Rationale for Our Component Model's Design

In this appendix, we motivate various design choices made in the process of defining our component model by explicitly tracing high-level design objectives that led to these choices. The information presented in this appendix helps the reader to quickly understand the rationale for the features of our component model, without having to read the detailed motivation presented in the main body of this thesis.

We analyse the component model's design in terms of (i) its *objectives*, (ii) its *principles* (high-level strategies for realising the design objectives), and (iii) the *decisions* that correspond to particular features of the resulting component model. We link these entities to allow for an easy tracing between them.

The main design objective (see objective #1 in Table F.1) corresponds to the main goal of our research – to come up with first-class abstractions for interaction patterns that would be reusable in the context of component models. Other design objectives either support the main objective (#2 and #3) or result from our decisions to limit the scope of our research to a particular domain (#4) and to define our component model fully, in a way that would make it possible to implement the prototype tool capable of designing and simulating systems in our model (#5).

Design principles (see Table F.2) elaborate on the design objectives; they are strategies applied when defining particular areas of the component model. They mostly correspond to the characteristics identified in Section 5.2 (principles A-G). Additionally, there are principles that refer to the domain-specificity of some parts

ID	Design Objective
1	First-class abstractions suited for interaction patterns
2	Modelling interaction across a wide range of systems (from data-driven to control-driven systems)
3	Scalable architecture modelling
4	Focus on reactive control software systems
5	Practical implementation of the component model designer and simulator

Table F.1: Design Objectives

of the component model (H) and that refine the objective of defining practical implementation of the component model (I and J).

ID	Design Principle	Objectives				
		1	2	3	4	5
A	Explicit architectural representation of connectors	×				
B	The ability to define new connectors	×	×			
C	Separate specification of interaction and computation	×				
D	Explicit control flow and data flow modelling	×	×		×	
E	Hierarchical composition of connectors	×	×	×		
F	Separate control flow and data flow modelling		×		×	
G	Design generic connectors to increase their reuse potential	×	×			
H	Relevant parts of the component model can be specific to the domain of reactive control systems				×	
I	Easily testable systems					×
J	Define simplified, but working, solutions for the parts of the model the deeper development of which is out of the scope of this thesis					×

Table F.2: Design Principles

Finally, Table F.3 traces which design principles led to particular design decisions in defining the features of our component model.

Area	**Design Decision**	**Design Principles**									
		A	B	C	D	E	F	G	H	I	J
Composition mechanism	Exogenous control and data flow coordination			×	×						
Interfaces	Port-based interfaces			×					×		
	Control and data ports				×		×				
Components	No required services			×							
	Flat composition structure										×
	3 types of components						×				
Connectors	Separate control and data connectors	×			×		×				
	Hierarchical composition		×			×					
Connector templates	Generic ports and multiports				×			×			
	Structural variability by means of roles							×			
	Constraints of connector template composition					×					×
Execution semantics	Control-driven and data-driven execution						×				
	Reactive execution cycle						×		×		
	Synchronisation rules						×		×	×	×
	Single flow of control					×				×	
	Concurrent flow of data						×				

Table F.3: Design Decisions