A SOFTWARE COMPONENT MODEL THAT IS BOTH CONTROL-DRIVEN AND DATA-DRIVEN

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

2012

Lily Suryani Safie
School of Computer Science
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Using Our Component Model for the Embedded Systems Domain</td>
<td>133</td>
</tr>
<tr>
<td>7.1</td>
<td>The Embedded Systems Domain</td>
<td>134</td>
</tr>
<tr>
<td>7.2</td>
<td>Climate Control System</td>
<td>136</td>
</tr>
<tr>
<td>7.3</td>
<td>System Modelling with Our Model</td>
<td>139</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Comparison with ProCom</td>
<td>143</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Comparison with SCADE</td>
<td>145</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Comparison with PECOS</td>
<td>147</td>
</tr>
<tr>
<td>7.4</td>
<td>Analysis</td>
<td>149</td>
</tr>
<tr>
<td>7.5</td>
<td>Chapter Summary</td>
<td>151</td>
</tr>
<tr>
<td>8</td>
<td>Using our Component Model for a Generic Domain</td>
<td>153</td>
</tr>
<tr>
<td>8.1</td>
<td>Domains and Domain Modelling</td>
<td>154</td>
</tr>
<tr>
<td>8.2</td>
<td>Feature Oriented Domain Analysis (FODA)</td>
<td>155</td>
</tr>
<tr>
<td>8.2.1</td>
<td>A Domain Model in FODA</td>
<td>156</td>
</tr>
<tr>
<td>8.2.1.1</td>
<td>Feature Models</td>
<td>156</td>
</tr>
<tr>
<td>8.2.1.2</td>
<td>Functional Models</td>
<td>157</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Hierarchy in Functional Models</td>
<td>162</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Vending Machine Example</td>
<td>164</td>
</tr>
<tr>
<td>8.3</td>
<td>Component-Based Domain Models</td>
<td>168</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Relating the FODA Functional Model to the Component Model</td>
<td>170</td>
</tr>
<tr>
<td>8.3.1.1</td>
<td>Terminators</td>
<td>170</td>
</tr>
<tr>
<td>8.3.1.2</td>
<td>Data Transformations</td>
<td>170</td>
</tr>
<tr>
<td>8.3.1.3</td>
<td>Data Flows</td>
<td>173</td>
</tr>
<tr>
<td>8.3.1.4</td>
<td>Control Transformations and Control Signals</td>
<td>175</td>
</tr>
<tr>
<td>8.3.2</td>
<td>A Component-Based Domain Model for the Vending Machine Example</td>
<td>180</td>
</tr>
<tr>
<td>8.4</td>
<td>Discussion</td>
<td>187</td>
</tr>
<tr>
<td>8.5</td>
<td>Chapter Summary</td>
<td>191</td>
</tr>
<tr>
<td>9</td>
<td>Evaluation</td>
<td>193</td>
</tr>
<tr>
<td>9.1</td>
<td>Control, Data, and Computation Modelling</td>
<td>194</td>
</tr>
<tr>
<td>9.2</td>
<td>Separation of Control and Data</td>
<td>198</td>
</tr>
<tr>
<td>9.3</td>
<td>A Component Model for Domains</td>
<td>201</td>
</tr>
<tr>
<td>9.3.1</td>
<td>The Embedded Systems Domain</td>
<td>202</td>
</tr>
<tr>
<td>9.3.2</td>
<td>Generic Domains</td>
<td>205</td>
</tr>
<tr>
<td>9.4</td>
<td>Software Reuse</td>
<td>207</td>
</tr>
<tr>
<td>9.5</td>
<td>Architecture Expressiveness</td>
<td>209</td>
</tr>
<tr>
<td>9.6</td>
<td>Chapter Summary</td>
<td>211</td>
</tr>
<tr>
<td>10</td>
<td>Conclusions and Future Work</td>
<td>213</td>
</tr>
<tr>
<td>10.1</td>
<td>Conclusions</td>
<td>213</td>
</tr>
<tr>
<td>10.2</td>
<td>Future Work</td>
<td>215</td>
</tr>
</tbody>
</table>
# Bibliography

## A. Constructing a Component-Based Domain Model for a Window Control System

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>Problem Description</td>
<td>227</td>
</tr>
<tr>
<td>A.2</td>
<td>A FODA Model for the Window Control System</td>
<td>228</td>
</tr>
<tr>
<td>A.2.1</td>
<td>The Feature Model</td>
<td>228</td>
</tr>
<tr>
<td>A.2.2</td>
<td>The Context Model</td>
<td>228</td>
</tr>
<tr>
<td>A.2.3</td>
<td>The Functional Model</td>
<td>230</td>
</tr>
<tr>
<td>A.3</td>
<td>A Component-Based Domain Model for the Window Control System</td>
<td>232</td>
</tr>
</tbody>
</table>
List of Figures

2.1 A generic component. ........................................ 30
2.2 Component definition in current models. .................. 31
2.3 Component composition in current models. ............... 32

3.1 Control, data and computation. ............................ 36
3.2 A control flow graph. ....................................... 36
3.3 An example of control-driven computation. ............... 37
3.4 A data flow graph. ......................................... 38
3.5 An example of data-driven computation. .................. 39
3.6 Data-driven and control-driven computation. ............. 40
3.7 The decomposition of functions in SADT. .................. 42
3.8 The spectrum of software systems. ........................ 43
3.9 Separation of control and data in SCADE. Figure taken from [62]. 45
3.10 An abstract architecture that consists of two parts: control and data. ................................. 46

4.1 Control flow and data flow with method calls. .......... 50
4.2 Entity bean for a library example. ........................ 52
4.3 Session bean for the library example. ..................... 52
4.4 Control-driven computations in EJB. ...................... 53
4.5 Architectural units as components. ....................... 54
4.6 ProSys subsystems. ........................................ 55
4.7 A simple ProSave component. .............................. 56
4.8 ProCom components and connectors. ....................... 57
4.9 Control-driven computation in ProCom: a distance and speed measurement subsystem example. ............... 57
4.10 A Lustre block. ............................................ 59
4.11 SSM in SCADE. ............................................ 59
4.12 Separation of control and data in SCADE. ............... 60
4.13 Encapsulated components. ................................ 62
4.14 X-MAN components and connectors. ...................... 63
4.15 Components as computation units connected by a coordinator as the connector. ......................... 68
4.16 The spectrum of software systems and component models. .............................................. 70
4.17 Comparing SCADE with our aim for the new component model. ........................................ 71
8.32 The STD for the hierarchy of control connectors shown in Figure 8.31
8.33 The component-based domain model for the vending machine.

9.1 A fuel system example that operates in three main modes.
9.2 A fuel system with a control connector hierarchy.
9.3 A fuel system without a control connector hierarchy.
9.4 A cruise control system architecture.
9.5 A separation of control and data with our model.
9.6 No separation of control and data in the system architecture with ProCom.
9.7 Separation of control and data in SCADE.
9.8 FODA and component-based domain models.
9.9 The code for a Calculate Ventilation component.

A.1 Feature model for the window control system domain.
A.2 Context model for the window control system.
A.3 Window control system: first level decomposition.
A.4 Window control system: second level decomposition.
A.5 All primitive data transformations for the window control system.
A.6 Move Passenger Window in Component-based Domain Model.
A.7 The rest of the functions in the component-based domain model.
A.8 The component-based domain model for the window control system.
## List of Tables

<table>
<thead>
<tr>
<th>Table Number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Categories of component semantics and component models.</td>
<td>34</td>
</tr>
<tr>
<td>4.1</td>
<td>An analysis of current component models.</td>
<td>65</td>
</tr>
<tr>
<td>4.2</td>
<td>Motivations and objectives of the new component model.</td>
<td>67</td>
</tr>
<tr>
<td>6.1</td>
<td>Expected behaviour of the cruise control system for our test scenario.</td>
<td>127</td>
</tr>
</tbody>
</table>
Abstract

A software component model is the cornerstone of any Component-based Software Development (CBSD) methodology. Such a model defines the modelling elements for constructing software systems. In software system modelling, it is necessary to capture the three elements of a system’s behaviour: (i) control (ii) computation and (iii) data. Within a system, computations are performed according to the flow of control or the flow of data, depending on whether computations are control-driven or data-driven. Computations are function evaluations, assignments, etc., which transform data when invoked by control or data flow. Therefore, a component model should be able to model control flow, data flow as well as computations. Current component models all model computations, but beside computations tend to model either control flow only or data flow only, but not both.

In this thesis, we present a new component model which can model both control flow and data flow explicitly. Furthermore, the modelling of control flow is separate from that of data flow; this enables the modelling of both control-driven and data-driven computations. The feasibility of the model is shown by means of an implementation of the model, in the form of a prototype tool. The usefulness of the model is then demonstrated for a specific domain, the embedded systems domain, as well as a generic domain. For the embedded systems domain, unlike current models, our model can be used to construct systems that are both control-driven and data-driven. In a generic domain, our model can be used to construct domain models, by constructing control flows and data flows which together define a domain model.
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

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which The University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in The University IP Policy (see http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University’s policy on presentation of Theses.
Acknowledgements

This thesis would not have been possible without the support of many people. I would like to express my deepest gratitude to my late dad, for giving me all the love and the strength that I have today. I needed those to support me throughout my PhD journey. I would also like to thank the Majlis Amanah Rakyat (MARA) for providing me with funding to support my PhD education.

Thanks to my supervisor, Dr. Kung-Kiu Lau, for all the intense discussions to ensure that I gained the skills required for a PhD.

I am truly indebted and thankful to Hilverd for the endless support and encouragement.

During this work, I have collaborated with two of my colleagues for whom I have great regard, and I wish to extend my warmest thanks to Petr and Cuong.

I would like to thank my comrades during my PhD years in Manchester: Fardad, Chenjuan and Yannis who have been such good friends, during both good and bad times. They have made Manchester a bearable place to live over all those years.

Friendship knows no distance and I want to thank my best friends Rahma and Witut back home for their endless love and support. To all my close friends whom I knew when I was with Telekom Malaysia, thank you for the friendship during all these years.

I would like to also thank my examiners, for taking the time to thoroughly read my thesis and suggest improvements.

I also wish to thank Azlin, Keng-Yap, Faris and Tauseef for all the valuable feedback that I received during the regular group workshops.

Finally, I would like to thank my mother and sister for their love.
Chapter 1

Introduction

Software development is getting increasingly complex, with a huge expansion in the use of software in our everyday lives. This has stimulated application of new techniques that help to reduce the cost and time-to-market of software systems. One of these techniques is Component-Based Software Development (CBSD), which is based on the concept of building the whole system from its parts. By constructing software systems from parts, the size and complexity of software systems is reduced, therefore making software development more manageable. This feature of CBSD makes it more attractive for software development, as compared to the traditional method of building software systems that focus on writing monolithic code. The methodology of CBSD aims to move from huge monolithic systems to more modular structures for software system development.

The concept of building the whole from its parts with the CBSD approach can be viewed as a process of building a house with bricks. The complete software system and the software components are analogous to the house and the bricks. Software components are units that are produced independently, to provide certain functionality that is meant to be composed into a functioning system [100]. This functioning system is the whole, which contains sub-parts with the functionality provided by the software components.

For CBSD methods, the development focuses not only on building software systems from parts, but also on developing these parts as reusable entities, and performing maintenance of the entities in order to maintain the systems [32]. One of the key characteristic of CBSD is software reuse. The focus is on building software systems by integrating pre-built software components. The principle of CBSD is to reuse, rather than reinvent, and to assemble pre-built components
rather than code the systems entirely from scratch. This leads to improved software development productivity, and reduction in production cost, since systems are built from pre-existing entities.

The context of the research presented in this thesis is the Component-Based Software Development methodology. In particular, the work in this thesis focuses on a software component model that models the fundamental aspects of any software system, which are control, data and computation. We introduce such a component model in the following.

1.1 A Component Model That Is Both Control-Driven and Data-Driven

A software component model is the cornerstone for any CBSD methodology. It provides the foundational framework of how software systems can be built using a CBSD approach. Over the past years, various component models have been defined in the literature \cite{35, 77}. These component models are defined with various usage contexts, and targeting different domains.

In general, a software component model defines what components are, what composition operators for composing components are, and how components can be connected or assembled together using these composition operators \cite{76, 77}. From this, we can be certain that software components are for composition, and the composition is done using software connectors. Both software components and connectors form a software architecture. A software architecture represents the software structure of a system. This means that a software component constitutes the independent sub-part of a software system, and a software connector forms the mechanism to bridge or compose the software components to form a system.

Like other software models, a component model is an abstraction of a running software system. In this regard, it is interesting to model the constituents of a software system. The three basic elements of a software system are (i) control, (ii) data, and (iii) computation. Computation means function evaluation, assignment, etc; control is used to trigger these executions; and data is the set of values used in computations and values that result from computations. These three elements should form the underlying fundamental elements for any kind of software systems modelling.
The order of computation executions can be defined in different ways, depending on whether the computations are control-driven or data-driven. In the classic von Neumann architecture, computations are control-driven. Executions occur one at a time, and control defines the order of computation executions. Control flow can be sequencing, branching or looping. By contrast, in data flow architectures, computations are data-driven (as in data flow languages [58]). Computation executions are triggered by data availability, i.e. a computation executes when all its required data is available. Any number of computation executions may occur simultaneously at any one time, therefore, the order of computation executions is non-deterministic.

In industry, tools and languages for developing systems in different domains are designed according to the domain's needs. They can be predominantly control-driven (e.g. Esterel [21] and Argos [84]) or data-driven (e.g. Lustre [28] and Signal [53]), or more usually they are a mixture of both.

For the domain of embedded systems, it is desirable to separate the modelling of control and data. For example, the standard tool for the avionics domain\(^1\), SCADE (Safety Critical Application Development Environment) [43], separates control from data flow, with control defined by Esterel and data flow defined by Lustre.

The modelling of control flow and data flow is also the main focus of the Feature Oriented Domain Analysis (FODA) approach [60]. It is a well-established domain analysis method, developed at the Software Engineering Institute (SEI). The method is designed to support the development of software systems for any domain. It uses the feature model to capture the common and variable features of a domain.

Because of the above reasons, for CBSD, the challenge is to define component models that can provide component-based counterparts to these tools and methods. In what follows, we contribute a reasonable amount of knowledge towards fulfilling this challenge.

In this thesis, we define a software component model that is both control-driven and data-driven. As we mentioned earlier, control, data and computation are the important elements for software system modelling. In particular, these elements determine the computation model that forms the basis of how a software system works. Considering all of these aspects has led us to develop a

\(^1\)This domain requires compliance with the DO-178B standard.
component model that models control, data and computation. The component model contains the explicit modelling of control flow and data flow, integrated but separated in a model. This separation of control flow and data flow leads to a model that is both control-driven and data-driven \[72\]. As we shall show through our survey, the features of control, data and computation modelling listed as the desiderata for a component model are not fully supported by other component models. Consequently, we fill this gap by presenting a new component model that is both control-driven and data-driven.

The design principles that we embrace for the new component model are deemed important and significant in software development for domains. In particular, the separation of control and data is viewed as the desirable characteristic for a model in the domain of embedded systems, and the explicit modelling of control flow and data flow is important in specifying the domain model for a domain analysis activity. Therefore, we place our work in the context of software development for domains. The component model is evaluated in the context of building software systems for domains, particularly the domain of embedded systems and a generic domain.

1.2 Contributions of the Thesis

The main contribution of the thesis is a better understanding of control, data and computation modelling in general and the development of a new component model in particular. Our overall contribution can be split into three groups: contributions to the understanding of control, data and computation modelling, embedded systems domain, and domain analysis approach, as well as contributions to a wider range of communities. The following contributions will be discussed in greater detail in the Conclusion and Future Work chapter (Chapter 10).

Contributions to the Understanding of Control, Data and Computation Modelling in a Component Model

- A literature review on control flow and data flow, control-driven and data-driven computation model (Chapter 3)
- A set of desiderata for control, data and computation modelling in a component model (Chapter 3)
1.3. **STRUCTURE OF THE THESIS**

- An in-depth review and analysis on control, data and computation modelling in current component models (Chapter 4)

- Definition of a new component model that is both control-driven and data-driven (Chapter 5)

- A proposal for execution semantics of handling control flow and data flow in the component model (Chapter 6)

**Contributions to the Understanding of Embedded Systems Domain**

- An observation of separation of control and data characteristic in embedded systems domain (Chapter 7)

- An in-depth evaluation of the proposed component model and the selected component models against the separation of control and data feature (Chapter 7)

**Contributions to the Understanding of Domain Analysis Approach**

- A literature review on the domain modelling approach (Chapter 8)

- A component-based domain model approach to perform domain modelling for CBSD (Chapter 8)

**Contributions in Wider Scope**

- A new approach to build software systems using a component model for Component-based Software Engineering (Chapter 5)

- Support from a component model for software development for domains (Chapter 7 and Chapter 8)

### 1.3 Structure of the Thesis

We develop the above mentioned contributions along the following outline.

In Chapter 2, we introduce software component models that constitute the cornerstone for any CBSD methodology. We describe the definition of software components and software component models as presented in the literature. Also,
a brief overview of the existing component models with respect to the software components definition is given as the background for the thesis.

In Chapter 3, we introduce the concept of control, data and computation modelling in a component model. We begin the chapter by introducing the basic concepts used throughout the thesis such as control flow, data flow, control-driven computation and data-driven computation. These concepts are fundamental to the theoretical foundation of the new component model proposed in this thesis. Having established the basic concepts, we then establish the desiderata of control, data and computation modelling in a component model.

In Chapter 4, we examine the related work, that is, control, data and computation modelling in existing component models. The survey is done based on the desiderata presented in Chapter 3. Based on the survey, we perform our analysis and present the motivations for proposing a new component model.

In Chapter 5, we present a novel component model that is both control-driven and data-driven. We begin the chapter by introducing the features of the new component model. This is followed by the definitions of the modelling elements for the new component model. They are explained in details with examples.

In Chapter 6, we describe a possible realisation for the new component model. This involves explaining the execution semantics for a constructed system in details. Following this, we briefly present the tool that is developed for the model and discuss the relevant issues.

In Chapter 7, we apply the new component model to the domain of embedded systems. We describe the characteristics of embedded systems and we perform the evaluation of our model with respect to the relevant characteristics. A case study is used for the evaluation, and we perform a close comparison with two component models in the embedded systems domain.

In Chapter 8, we apply the new component model to a generic domain. That is, we propose the idea of using the new component model for a component-based domain model. First we thoroughly explain the FODA domain modelling approach as the background. Second, we show how we construct the component-based domain model using an example.

In Chapter 9, we evaluate the research work presented in this thesis. This chapter addresses how the proposed research in this thesis exceeds the existing models in modelling the control, data and computation in software systems. This includes a discussion of the strengths and weaknesses of the new component
In Chapter 10, we conclude by reviewing the work that has been done for the thesis, and relate back to the stated research objectives. Additionally, we propose directions for future work.
Chapter 2

Software Component Models

A software component model forms the fundamental underlying framework in any Component-Based Software Development (CBSD) methodology. Therefore, in this chapter, we introduce software component models by providing an overview of what a software component model is, and the current work that involves software component models as described in the literature. This chapter aims to serve as the background of this thesis.

We begin with the definitions of a software component model in general in Section 2.1. This includes the definition of the basic elements involved in a component model. Following this, we present a detail view of software components and the composition mechanism that involved in connecting software components in Section 2.2. Finally, we review briefly the current models defined for CBSD approach in Section 2.3.

2.1 Definitions of Software Component Models

Component-Based Software Development (CBSD) [100, 56, 33, 32, 78, 42] represents a development paradigm in Software Engineering that is based on building software systems from software components. The importance of software components in the context of CBSD is evident from the various existing definitions in the literature for software components, including [100, 87, 56, 11, 23, 30]. We quote the two commonly adopted definitions from Szyperski [100] and Meyer [87]:

A software component is a unit of composition with a contractually specified interface and explicit context dependencies only. An interface is a set of named operations that can be invoked by the clients.
Context dependencies are specifications of what the deployment environment needs to provide, such that the components can function. A software component can be deployed independently and is subject to composition by third parties.

A component is a software element (modular unit) satisfying the following conditions: 1) It can be used by other software elements, its “clients”. 2) It possesses an official usage description, which is sufficient for a client author to use it. 3) It is not tied to any fixed set of clients.

These definitions emphasise the nature of component as being a self-contained unit that is designed to be reusable and composable through a well-defined interface. This means that a set of software components can be combined to build larger software applications. However, these definitions do not mention how components can be composed together to build software systems. This fact is recognised by Heineman and Councill [56] and Szyperski [100], where the notion of a component model is mentioned as the entity that determines a specification for a component:

(A 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. ([56])

... to enable composition, a software component adheres to a particular component model. ([100])

The above definitions show that it is important for a component to be defined in the context of a component model. A software component model plays an important role in any CBSD approach, and this is illustrated in the definition by Lau and Wang [77], and Chaudron and Crnkovic [29]:

A software component model should define: (a) the syntax of components, i.e. how they are constructed and represented; (b) the semantics of components, i.e. what components are meant to be; (c) the composition of components, i.e. how they are composed or assembled. ([77])
A Software Component is a software building block that conforms to a component model. A Component Model defines standards for (i) properties that individual components must satisfy and (ii) methods, and possibly mechanisms, for composing components. \[29\]

All of these definitions illustrate the importance of a component model in any CBSD approach. A component model represents the basis in which the components can exist and interact. It provides the underlying semantics framework that affects the way the software is built in CBSD.

Besides component models, there exist architecture description languages (ADLs) that define the same entities as the component model. An ADL \[85\] is a language that is used to describe the architecture for a software system. Basically, the underlying semantics of an ADL is a component model, and therefore an ADL can be regarded as a component model.

The essence of a component model lies on the definitions of the software components and how they can be composed to build a software system. These two definitions form the modelling structure of a software system that represent the basic constituents – control, data and computation. Software components are units of computation that provide certain functionalities as sub-parts of a software system. These components interact with each other, through a medium that involved passing the control and data.

To summarise, a component model defines the basic elements of constructing software systems in a CBSD approach. The definition of components and the component composition forms the two fundamental elements in a component model.

### 2.2 Software Components and Compositions

A software component is the fundamental unit of a system constructed using a CBSD approach, and must be defined in the context of a component model. In this definition, the component is specified to represent the basic building blocks, and the semantics of the component essentially supports the composition of components to build software systems.

In general, a software component has an interface that acts as the communication point between the component and the outside world. These interfaces are normally the provides interface and the requires interface, that specify the
services delivered by the component and the services expected by the component, respectively. Figure 2.1 illustrates the structure of a generic component as a unit with provided and required services.

![Figure 2.1: A generic component.](image)

The composition of components is highly influenced by the way that components are defined. In the current component models, components and composition of components are defined in various ways, which can be categorised in the following three categories [77, 71]: (i) objects (as in object-oriented programming), (ii) architectural units (as in ADLs), and (iii) encapsulated components. These categories focus on how components are defined in current component models. By looking at the component definitions, we can better understand the composition mechanisms used by these models to compose software components.

Components that are defined as objects only have provided services which specify the provided methods by the objects. Figure 2.2(a) depicts a structure of a component that is defined as an object. It is a piece of code that is specified with respect to the conventions defined in the model specification. The interface contains the provided methods implemented in the classes or objects. These methods can be called by other components.

For architectural units, components are defined with their input and output ports, that represent required and provided services respectively. Figure 2.2(b) depicts a structure of an architectural unit that is defined with two input ports and two output ports. Architectural units are composed by connecting the provided port to the required port, and those ports must be compatible.

For encapsulated components, the only interface specified is the provided services. Computations inside the components are encapsulated and self-contained. This means that a component performs its own function without depending on other components to realise the function that it is supposed to provide. This makes components truly independent of each other when they are composed. Figure 2.2(c) shows a structure of an encapsulated component.
Component composition is a central focus in the definition of a component model. The composition mechanism adopted by a component model is closely related to the semantics of software components. We have explained the categories of current component models based on the semantics of components. Accordingly, the composition mechanism follows the same categories, which are: (i) composition by direct method calls for the category where components are defined as objects, (ii) composition by indirect message passing for the category where components are defined as architectural units, and (iii) composition by coordination for the category of encapsulated components.

In the first category, where components are defined as objects, composition of components happens by direct method calls. Figure 2.3(a) shows how components A and B that are defined as objects communicate with each other using direct method calls. This connection is not specified explicitly and it is buried inside the implemented code. Components communicate directly and know about each other, which in turn, induces a tight coupling between components.

The composition of components is different in the second category, where components are defined as architectural units. The connections between components through the ports are realised using connectors. Figure 2.3(b) depicts the structure of the composition where components C and D are two architectural units that are linked together using a connector X. The communication between components is realised indirectly via the connector. This implies that components do not know about each other, however, the connection requires the knowledge of the connector that is involved in the connection. The connectors are channels that link up the interfaces of components in the specification of the models. Although the connected components communicate with each other indirectly via the connectors, it is the component that initiates the method calls to the other components.
In the third category, a component is defined as an encapsulated entity that performs computation within itself. The computation does not involve invoking other entities in the system, and therefore, it needs a composition operator that lives as a separate entity outside the components. This entity is responsible for performing the coordination in composing software components. The coordinators are called *exogenous* connectors \[66\]. For the composition, exogenous connectors initiate the method calls to the component, and handle the control coordination between components. Figure 2.3(c) depicts this composition mechanism, where components E and F are defined as two encapsulated components that are connected using an exogenous connector EX. These two components do not call each other. It is the responsibility of connector EX to initiate the method calls to these components, and coordinate the functions for the composition. The coordination performed by the exogenous connector removes the coupling between components.

![Component composition in current models.](image)

(a) Connecting components by direct method call  
(b) Connecting components by indirect method call  
(c) Connecting components by a coordinator

In the following subsection, we will review briefly the current component models with respect to the definition of the software components and composition presented.
2.3 Overview of Current Models

Following the notion of having a component model in the CBSD approach, a number of component models have been developed in recent years\(^1\). Some examples are EJB [41], COM [22], ProCom\(^2\) [97, 25], PECOS [90], SOFA [21], Koala [103], Palladio [16] and X-MAN [77, 70]. Examples of ADLs include Acme [48], MetaH [81] and EAST-ADL2 [10]. Additionally, there exist tools that support CBSD which are widely used in the industry, such as SCADE [43] and Simulink [1]. These tools carry the underlying semantic of component models that make the application development using a component-based approach possible. The aforementioned component models, ADLs, and tools are referred to hereafter as component models.

The existence of various component models with different characteristics reflects the diversity of the targeted system domain that component models can be used for. On one hand, a component model is designed to fulfil the requirements of building software for a specific domain. For instance, there exist a number of component models that are specifically designed for the domain of embedded systems. ProCom, PECOS, SOFA and SCADE are examples of component models that are targeted for this domain. Another example is the Koala component model, which is targeted for the consumer electronics domain. On the other hand, there are generic component models that are designed for general use in building software systems. Such component models include EJB, COM and X-MAN.

With regards to Figure 2.2 that shows how components are defined in these component models, we can divide the component models into three categories [77, 71]: (i) models that define objects as components; (ii) models that define architectural units as components, and (iii) models that define encapsulated components. Based on these component semantics, current component models can be grouped into three categories. Table 2.1 shows the categories and the examples of component models. These categories, together with the representative models, will be examined in detail in Chapter 4.

Similar to other software modelling methodology in software engineering, component models are used to specify the design of software systems. In essence, software models represent abstractions of some aspect of software systems. Control, data and computation that form the basic elements of any software system\(^1\) Recent survey of component models can be found in [35, 77].\(^2\) ProCom is a successor of SaveCCM [54].
must be defined clearly in any software models. These elements highlight the two essential aspects of software system modelling which are control flow and data flow. One is concerned with the functional computations and data dependencies between computations, and the other one is concerned with the ordering of triggering the computations and defining the overall behaviour of the system.

In this thesis, we present the design of the new component model whose main concern is the modelling of control, data and computation. This in turn, results in having a component model that is both control-driven and data-driven. The details of this topic will be given in the next chapter.

<table>
<thead>
<tr>
<th>Component semantics</th>
<th>Component models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objects as components</td>
<td>EJB, COM</td>
</tr>
<tr>
<td>Architectural units as components</td>
<td>ProCom, Koala, PECOS, SOFA, Palladio, Acme, MetaH, EAST-ADL2, Simulink, SCADE</td>
</tr>
<tr>
<td>Encapsulated components</td>
<td>X-MAN</td>
</tr>
</tbody>
</table>

Table 2.1: Categories of component semantics and component models.
Chapter 3

Modelling Control, Data and Computation

In this thesis, we set out to propose a new component model that is both control-driven and data-driven. This chapter explains the main research ideas which the thesis is based on. We focus on the importance of a software component model in modelling control, data and computation. We begin by introducing control-driven and data-driven computations in the view of control flow and data flow, respectively, in Section 3.1. This is because these two computation models form the fundamental concepts underlying the new component model. Following this, in Section 3.2 we elaborate on the characteristics of the desired component model with respect to the modelling of control, data and computation: (i) explicit control flow and data flow modelling (Section 3.2.1); (ii) control-driven and data-driven system modelling (Section 3.2.2); and (iii) explicit separation of control flow and data flow (Section 3.2.3).

3.1 Control, Data and Computation

Control, data and computation form the basic elements for every software system. This is explained using Figure 3.1. Generally, computation represents function evaluation, that is, given an input it will transform it to an output. Computation needs control and data to realise this transformation. Control is required to trigger the computation, and data is the set of values required by the computation and provided as the results from computation. This is the fundamental idea behind any computation model. In this thesis, we focus on the two main
computation models on which a spectrum of software architecture models can be based. They are the von Neumann serial control flow model and the data flow architecture model.

![Figure 3.1: Control, data and computation.](image)

In the classic von Neumann architecture, the control flow can be described by a directed graph such as in Figure 3.2, where the nodes represent the computation and the dashed directed arcs define the flow of control between computations. This flow represents the sequential execution where the path is based on the program counter. The program counter determines the ordering for program execution and the control flow triggers the execution of computations. Figure 3.2 shows that the ordering causes A to execute first, followed by B, and then C. The execution of computations is control-driven since control flow triggers the executions which occur one at a time. Control flow can be sequencing, branching or looping.

![Figure 3.2: A control flow graph.](image)

In Figure 3.3, we sketch a specific example to explain control flow and control-driven computation. In this example, the value for x is computed by three computations illustrated as boxes connected by the dash lines that denote the control
flow. These computations involve the variables \(x, y, z, t_1\) and \(t_2\). For control-driven computations, data is communicated indirectly by reference. Therefore, in this example, all the variables are kept in a shared memory container that stores the values for the variables. Consider the initial values of \(y\) and \(z\): these are set to 3 and 7, respectively. The first box of computation will produce the value 4 for \(t_1\). This value will be kept in the memory container for reference. The sequential order of triggering the computation by the control flow will cause the second box of computation to compute the value of \(t_2\), which in this case yields the value 2 for \(t_2\). This value will be kept in the memory location for \(t_2\). Finally, the control flow will trigger the third box of computation, which calculates the value for \(x\) using the values for both \(t_1\) and \(t_2\).

\[
x := (y + 1) \times (z - 5)
\]

Figure 3.3: An example of control-driven computation.

Similarly, we describe data flow architectures in terms of directed graphs where the nodes of the graph represent the computations shown in Figure 3.4. Here \(D, E\) and \(F\) are the nodes that represent computations. The solid arrows denote the data flow toward a node (as the data input to the node) and data flow away from the node (as the data output from the node). The executions of nodes can happen only when the data is available as inputs to the nodes. There is no ordering of execution, and the execution of the nodes is driven by the availability of data arriving at them. Figure 3.4 shows that there are data dependencies between \(D\) and \(F\), and between \(E\) and \(F\). The node \(F\) needs data produced by \(D\) and \(E\) to execute. Similarly, \(D\) and \(E\) need data input in order to execute. For
this type of architecture, computations are *data-driven*, where control exists in the form of data availability. Computations can be coordinated by coordinating the data arrival at computations. This means that a number of computation executions can occur simultaneously if data is made available at multiple places in the architecture.

Figure 3.4: A data flow graph.

Figure 3.5 shows a specific example for data-driven computations. The value for $x$ is computed by three data transformations that are connected via data flow channels. The entities denoted as $x, y, z, t1$ and $t2$ represent data tokens that transport data values in the data flow channels. Note that in data-driven computation model, data is communicated directly by data values between data transformation operations. Data tokens are used to pass data values from one computation to another and also drive the execution of the computation. Consider $y$ and $z$, which carry data values of 3 and 7 respectively. The two top data transformations are performed in arbitrary order, since the data availability drives the operations of data transformations in a data-driven computation model. Subsequently, the values of $t1$ and $t2$ are produced by these two computations. The data values will travel along the channels moving towards the third computation box. Similarly, the availability of both data values $t1$ and $t2$ causes the operation of the third data transformation, and produces the value for $x$.

Historically, different requirements of these two computational models on design and implementation methods led to the emergence of control-oriented imperative programming languages and dataflow programming languages. The imperative or control flow languages like Esterel [17] or Argos [84] are used to model systems where the control is driving the execution. These languages have been designed to describe modes, handle events, and support interrupt-driven
behaviour. The declarative or data flow programming languages like Lustre [28] or Signal [53] are used to model a system with regularity in its behaviour and which is data-driven. These languages are designed to handle a system that reacts continuously to the environment.

There exist software systems that combine non-trivial control and data processing parts. The need to model both a data-driven computation and a control-driven computation part of the system in the software design has inspired the development of a hybrid model, that combines both a data flow architectural model and a von Neumann serial control flow model [57, 47]. Figure 3.6 illustrates the scenario of a hybrid model that consists of both data-driven and control-driven computations. Initially, D1 to D6 are computations in the system that perform data transformations when data is available, having been transported to the computation units via the data flow. The model is purely data-driven, since all the computation units perform their data transformation as long as data is available. The situation can change if we change the system requirements, say, we want either D3 or D5 to operate at one particular time. This requires the introduction of control to ensure that only one of them is active at a particular time. The control mechanism employed in the system makes D3 and D5 not only data-driven, but also control-driven. The control flow region sketched in Figure 3.6 represents this scenario.

We consider control flow and data flow to be two essential elements for software modelling. Many aspects of software can be modelled from a control flow and data flow perspective. It is our intention to design a component model that
emphasises the modelling of control, data and computation. Having presented control flow and data flow in detail, in the following sections we explain the main features of the component model that we develop in this thesis.

### 3.2 Desiderata of Control, Data and Computation Modelling

As described in Chapter 2, the cornerstone of any CBSD method is the underlying component model that defines what components are and how can they be composed together. The quoted definitions of components and component models [100, 87, 56, 77, 29] clearly show the importance of having a component model as the modelling and implementation framework for a CBSD methodology. With respect to control, data and computation modelling, this thesis proposes a new component model that focuses on these aspects of modelling. Based on the fundamental concepts of computation models that we presented in the previous section, in this section, the desirable features of a component model that is both control-driven and data-driven are elaborated upon. These features are explained in turn as follows.

#### 3.2.1 Explicit Control Flow and Data Flow Modelling

In Section 3.1, we explained how control flow and data flow express the two main computation models which are based on von Neumann’s machine and the data...
3.2. DESIDERATA OF CD&G MODELLING

flow architecture machine, respectively. That is, the sequential execution of a program’s instructions sharing global data became the foundation of von Neumann’s execution model, and parallel execution of all program’s instructions that have their inputs ready and local data values shared only by their producing and consuming instructions underlie the dataflow execution model. Thus, naturally we can model the two main computation models if we have a model that expresses control flow and data flow as explicit entities alongside computation.

In general, a model is a form of formalisation of allowed functions and their dependencies. The common approach in specifying a system’s functionality is to perform decomposition of functions into smaller parts, and to describe the relationship between the parts. This relationship involves control flow and data flow, in which the modelling of these two elements effectively captures the dependency between functions or computations. The explicit modelling of control flow, data flow and computation as separate entities in a model influences the ability to understand and define the system functionality. For example, in the Structured Analysis and Design Technique (SADT) [10], the model that is used as a tool in specifying the system functionality expresses the data flow, control flow and data transformation (or function) as explicit entities that influence the system analysis and design. The methodology structures a decomposition approach with the goal of describing the overall system’s functionality in smaller pieces of functions that are dependent on each other with the passing of control and data. Figure 3.7 shows an example of a system decomposition using SADT where a system is decomposed up to two levels. At level 0, the function $S$ is modelled as receiving input in the form of data flow and control flow, and sending control flow and data flow as outputs. The function $S$ is decomposed into smaller functions at level 1, which are further decomposed at level 2. The dependencies between functions are clearly described with the explicit modelling of control flow and data flow.

For the CBSD approach, it is desirable to have these characteristics in software systems modelling using component model elements, where the control flow, data flow and computation are made explicit as modelling entities. By doing this, functions and relationships between functions can be described clearly.

As explained in Section 3.1, it is evident that explicit control flow and data flow modelling in programming languages help in describing the behaviour of software systems. For example, data flow languages express regularity in the system behaviour while control flow languages express the discrete features of the
Figure 3.7: The decomposition of functions in SADT.

system behaviour. In addition, control flow and data flow are the two elements that describe the dependencies between computations. This means that the overall system behaviour can be described with the explicit modelling of control flow and data flow. For the CBSD approach, a component model should be able to specify how the system works at a reasonable level of abstraction. Therefore, having explicit modelling of control flow and data flow in a component model is beneficial for clearly describing the overall systems behaviour.

3.2.2 Control-Driven and Data-Driven System Modelling

The data-driven and control-driven computation models presented in the previous section illustrate the basis of modelling a spectrum of software systems. The two extremes of the spectrum are control-intensive systems, most of whose specifications are formed by control structures (sequencing, branching, looping), and data-processing systems, which essentially consist purely of data transformation operations. Although every software system contains some part dedicated to control and some part dedicated to data (processing), different kinds of systems exhibit different proportions thereof, creating the whole spectrum.

Figure 3.8 illustrates this spectrum, with the two extreme ends respectively being the data processing intensive systems and the control intensive systems. On the data processing systems end, the focus is on modelling data flow and data
transformations that involve processing the data input to produce data output. The computation model of this type of system is data-driven. An example of a data processing intensive system is a signal processing system that consists of many data processing functions, transforming streams of input data values to produce streams of output data values. On the control-driven end, the focus is on control flow modelling, where the ordering of data transformations or function execution is of prime importance. The modelling of data flow in this type of system is not important. Control flow modelling is of prime importance which makes the system a control-driven system. An example of a control intensive system is a vehicular system that requires the system to comply to a behaviour that meets the safety requirements.

In the middle of the spectrum lies the type of system that combines both computation models, making it a control-and-data-driven system. An example of such a system is an embedded system. Most embedded systems require modelling for both control flow and data flow [108, 105, 104, 13]. This is because the requirements imposed for embedded systems usually involve explicit modelling for both discrete and regular behaviour, which are captured by these two flows. Another example is real-time systems, which are considered a subset of embedded systems with timing requirements. In the analysis and design methods for real-time systems, it is important to model both control flow and data flow as exemplified by the literature [10, 110, 107, 55].
In this thesis, we create a new component model that covers the whole spectrum of software systems modelling. That is, a component model that will support the modelling for control-driven systems, data-driven systems, and data-and-control-driven systems.

3.2.3 Explicit Separation of Control and Data in a System Architecture

With the concept of modelling data-driven and control-driven systems that we introduced in the previous section, we have shown the importance of capturing control flow and data flow explicitly in software models. Here, the aim is not only to distinguish between the two flows in the system model, but also to have a clear separation of control and data in the system architecture. What we really want to achieve is a separation of control and data like in SCADE (Safety Critical Application Development Environment) [43], as described next.

SCADE is the standard tool for the avionics domain\(^1\). It integrates the usage of two languages: Esterel [21] as the imperative language with the program structure reflecting the control flow, and Lustre [27] as the data flow language that is purely functional. Having both languages enables the use of SCADE to construct systems that contain cooperating continuous and discrete control parts. In other words, SCADE provides the means to seamlessly couple both data flow and state machine specifications. However, SCADE does not propose any design methodology in using the tool. Users are free to construct systems specification that mix both control and data flow. This may result in difficulties understanding the model, which in turn makes the model not reusable and difficult to verify [62].

In [83], the concept of mode-automata was introduced to effectively manage the complexity of modelling the states for software systems. One of the main aims of mode-automata is to clearly separate control from signal processing. Based on this, the concept of running modes for SCADE specifications is proposed in a new design methodology [63]. This methodology places emphasis on a clear separation of control and data flow to achieve the following benefits.

1. Increased readability and understandability of the model — it is easy to read the architecture model, since it contains parts that are dedicated to control, and parts that are dedicated to data.

\(^1\)This domain requires compliance with the DO-178B standard.
2. Increased modularity in system development, providing better facilitation of reusing existing applications, adding new models, modification of existing models and deletion of models.

3. Easier verification — smaller parts of the system can be verified independently, and this is more efficient than verifying the whole system’s architecture model.

Figure 3.9, taken from [62], shows a Climate Control System modelled in SCADE. It shows how the separation of control and data flow is achieved by having a state machine defined in Esterel to model the control part and a Lustre block to model the data flow part. SCADE will be surveyed in detail in Chapter 4.

Our interest in designing a component model that separates control and data is also motivated by the important principle of managing complexity in the design of software architecture — in particular, the separation of concerns in the software architecture modelling by separating control and data [96]. We can imagine, for instance, a software architecture build from the smaller pieces of functions that we illustrated in Figure 3.7 with a mixture of control flow and data flow everywhere in the model. Having the control part distinctly separated from the data part in an architecture model would make the architecture relatively less complex than the ones that mix both elements in the architecture. It is possible to have this separation as a feature in the design of a component model since
it is a well-established practice in building software for a certain domain. For example, the classical control theory that clearly separates the *controller* from the remaining *functional* part of the systems \[50\] is widely employed for the design of software in the embedded systems domain. The basic idea is to have two parts in the architecture: control and data, where both parts are clearly distinguished and linked to each other. This can be illustrated by Figure 3.10 which shows an abstract architecture with two distinct parts that represent control and data. The data part should consist of the functions that transform input data to output data, as well as the data flows between the functions. The control part should consist of the hub where all the control required for the system is processed and expressed. Note that the two parts are linked together denoted by the arrows between the control part and the data part. The control part controls the computation that happens in the data part, while the data part feeds the data required for the control processing.

![Diagram](https://via.placeholder.com/150)

Figure 3.10: An abstract architecture that consists of two parts: control and data.

For the new component model, it is highly desirable to separate the portion that is dedicated to control from the portion that is dedicated to data in the architecture model. Therefore, we adopt the concept of separation of concerns for our component model, in particular separation of control and data in building a software architecture.

### 3.3 Chapter Summary

This chapter has served as the background to the fundamental concept which this work contributes. That is, the definition of control-driven computation and
3.3. CHAPTER SUMMARY

data-driven computation with respect to control flow and data flow, and their role in proposing the new component model in this thesis.

The new component model is set out to achieve the integration of control flow and data flow modelling in a model, as it provides the basis of having a component model that is both control-driven and data-driven. Complementary to this aim is the goal to have a clear separation of control flow and data flow in the model for the reasons that we have described in Section 3.2.3.

The desiderata listed in this chapter are the ideal basis for modelling software systems using a component model. However, they are not fully achieved by current component models, which we will show in the detailed survey regarding this in the next chapter.
Chapter 4
Control, Data and Computation Modelling in Current Component Models

In the preceding chapter, we presented the importance of control flow and data flow modelling in building software systems that forms the basis for the computation model. In this chapter, we survey the current component models that have elements of control flow and data flow, but with varying degrees of separation between them. The survey serves to identify the related work for this thesis. The current component models will be examined according to the desiderata described in Chapter 3.

The definition of a software component reviewed in Chapter 2 shows how important it is to define a software component in the context of a software component model. Moreover, component composition is a central issue in every component model since components are meant to be composed. As we described in Chapter 2, there are three categories of how components are defined in the current component models [77, 71]: (i) models that define objects as components; (ii) models that define architectural units as components, and (iii) models that define encapsulated components. For the survey in this chapter, we choose representative examples for models in each of these three categories, instead of choosing all of the models. We survey EJB [41] for the first category, ProCom [97] and SCADE [43] for the second category, and X-MAN [77, 70] for the third category.
CHAPTER 4. CD&C MODELLING IN CUR. COMPONENT MODELS

For these models, we survey the computation model (control-driven or data-driven), the modelling of control flow and data flow, and the degree of separation of control flow and data flow in the model. Details of the survey are presented in Section 4.1, Section 4.2 and Section 4.3. Subsequently, in Section 4.4 a detailed analysis is presented with respect to the desiderata of modelling control, data and computation in a component model. This in turn leads to the motivation of proposing a new component model that is both control-driven and data-driven, as described in Section 4.5.

4.1 Objects as Components

In the first category, components are defined as objects. The latter communicate to each other via direct method calls. Figure 4.1 shows an example of two components, A and B, that are defined as objects and connected to each other. They communicate via a method call, where A is the caller and B is the callee. Component A, via its source code, specifies the callee which is component B, and the method to invoke in B, which is M2. Invocation of method M2 by component A causes the control to flow from component A to component B. In addition, in this method call, component A passes the parameter p to the method M2. The passing of this parameter causes the data to flow from component A to component B. The data flow follows the control flow.

![Control flow and data flow with method calls.](image)

Components are objects that perform computations when they are triggered via method calls. This means that the computation is purely control-driven, and occurs when an object calls the method of another object. Control flow is thus the sequence of method calls. Like control, data is passed from the caller to the callee, and returned by the callee to the caller. Hence, data flow coincides with
control flow. It is therefore clear that there is no separation of control flow and data flow in this model. The two flows are coupled together, with the control playing the role of the main driver.

Examples of component models where components are objects (as in object-oriented programming) include EJB [41], CCM [52] and COM [22]. In the following subsection, EJB is used as a representative model to explain the control flow and data flow modelling for models that defined objects as components.

4.1.1 EJB

Enterprise Javabeans (EJB) [41] are a managed, server-side software component model. EJB belong to the category where components are objects. An EJB component (also referred as enterprise beans) is a Java object that lives in the EJB container. This means that the components can only function inside this container. The EJB container which is part of a J2EE server provides an execution environment for EJB components and manages client access.

There are three kinds of EJB components: (i) entity beans, (ii) session beans, and (iii) message-driven beans. Entity beans represent persistent data for database access. Session beans represent business objects and execute on behalf of external and internal clients. Message driven beans execute messages received from the clients. All components communicate via direct method calls. The method calls are specified in the source code during the design phase. In runtime, the reference between components via method calls is done by the container’s runtime services.

EJB components are specified in the Java class during the design phase where the source code is written for the enterprise beans. Methods are defined in the class and these definitions must be accompanied by code of two interfaces which are the home interface and the remote interface. These interfaces are used by the EJB container to manage and run the bean. The remote interface exposes the bean methods and provide the access to the bean for the client applications. The home interface provides the create, destroy and locate methods of the bean.

Consider a simple example of a library system that maintains a database for the books in the library. The inventory of the new books added to the library collection must be entered in the library system and this is done by a librarian. The library system can be implemented using a set of enterprise beans that allows multiple clients to access and update the database. Figure 4.2 shows an example
of how this scenario is implemented by an entity bean representing the table of books in a database.

The entity bean in Figure 4.2 consists of one class with two interfaces. The Java class is `BookBean`, `BookHome` is the home interface of the entity bean, and `Book` is the remote interface of the entity bean. The client can use the entity bean by first creating the instance of the entity bean. This is done by calling the ‘create’ method defined in the home interface `BookHome`, which returns an instance of the remote interface `Book`. Then the client can call the ‘set’ methods defined in `Book` to insert a record of a new book in the database.

There may also be a session bean in this example, like in Figure 4.3. The
session bean consists of the class `LibraryBean`, the home interface `LibraryHome` and the remote interface `Library` (and the helper class `Books`). In this example, the client can use `LibraryBean` to add details of a set of books into the table of books in the database. The `LibraryBean` class defines the operation performed on the table of books. For instance, the `addBook` method takes the information (title, author, publisher, etc.) for a set of books and adds these to the database. The add operation involves calling the methods in the entity bean. Similar to the entity bean, methods in a session bean correspond to methods in its home and remote interfaces.

In general, EJB components are Java classes and interfaces, and they are composed together by delegation of method calls. The EJB container in Figure 4.4 shows an example of composing the enterprise beans for the library system, where each method call contains control flow and data flow. The client in Figure 4.4 is the `ClientApp` that calls the method `addBook` where it passes the book information as the parameters. The control drives the method call, and the data flows with the control flow. The computation inside the components is driven by the control flow, as illustrated in Figure 4.4. The sequence of method calls involves a call from the client to the `addBook` method, which in turn calls the methods in BookBean to store the record in the database. Data flow coincides with the control flow, and therefore, there is no separation of control flow and data flow in EJB.
4.2 Architectural Units as Components

In this category, a software component is defined as an architectural unit. A classic example of this are Architecture Description Languages (ADLs) [85]. In terms of structure, an architectural unit has input and output ports, denoting the interface where it requires and provides services, respectively. Units are composed together by connecting ports that are compatible. In contrast with objects as components as described in the previous section, architectural units are connected by connectors instead of calling each other directly. An output port of one component is often linked to the input port of another component by using a software connector.

![Diagram](attachment:architectural_units_connector.png)

(a) Architectural units and a connector.  
(b) Control flow and data flow.

Figure 4.5: Architectural units as components.

Figure 4.5(a) shows an example of two components, C1 and C2, that are defined as architectural units. They are connected to each other via a connector X. C1 and C2 communicate indirectly through this connector. The communication mechanism is exemplified by Figure 4.5(b), where the method M1 in C1 notifies the connector to invoke method M2 in C2. The control flow is originated from component C1, and flows to the connector X and to component C2. Components perform computations when they are triggered by control, and therefore, the model with architectural units as components is control-driven. In this example, there is a parameter q, which is the data input for method M2. The data flows between components in tandem with control flow between them. Thus, there is no separation of control flow and data flow. Examples of component models with the above characteristics include PECOS [90], ProCom [97], Koala [103] and SOFA [24].
The above example shows the typical case of component models that defined components as architectural units. However, in SCADE [43] and SIMULINK [1], components are defined as architectural units and the computation is driven by data. This means that there are two types of computation models employed by component models that define components as architectural units: (i) control-driven models and (ii) data-driven models. We will take a close look at the representative for each type of model in the following sub-sections by surveying ProCom and SCADE. ProCom is a control-driven model, while SCADE is data-driven.

### 4.2.1 ProCom

ProCom is a component model that is used for the software design of real-time and embedded systems [27, 25]. The architecture for this model is structured in two layers: ProSys and ProSave.

ProSys is the upper layer that models a system as a collection of active concurrent subsystems that communicate via asynchronous message passing. Figure 4.6 shows an example of subsystems (A, B, C, D and E) that are modelled as ProSys components, where the message ports are connected via message channels. The message passing via these channels are done asynchronously.

![ProSys Subsystems](image)

**Figure 4.6: ProSys subsystems.**

The lower layer, ProSave, models the primitive components for ProSys. This
layer contains components that are passive and represent simple units of functions. ProSave components communicate with the environment via their input and output ports.

There are two types of ports: trigger ports and data ports. Trigger ports are the control ports for the component. Figure 4.7 illustrates the structure of a simple ProSave component. The component \( P \) is defined with two input ports and three output ports. Activation at the input trigger port will cause the component to read the current value at its input data port and perform computation based on this value. After the component is done with the computation, the output value will be written to the output data port, and control is forwarded to the output trigger port.

A group of output ports may form a single service, in which a function of a component can be accessed externally when a group of input ports is activated. In Figure 4.7, there is only one service provided by component \( P \), as it only has one group of output ports. This service is available at the output port when the group of input ports (in this case, the group contains only one trigger port and one data port) is activated. In other words, a service can be activated by triggering a single input trigger port and filling up a set of input data ports required for the service.

The components are connected by connections and connectors that transfer control and data. Connections connect two ports, while connectors may be used to connect two connections. Figure 4.8 shows a simple example of ProCom components \( C1 \), \( C2 \) and \( C3 \), connected by connectors \( X \) and \( Y \). These connectors are used in between connections that connect the components, where \( X \) is a control connector (control fork) and \( Y \) is a data connector (data demuxer).

Consider a simple example of a distance and speed measurement subsystem, that is used in a cruise control system installed in a car. This subsystem requires the current shaft reading and the last distance travelled as input parameters to
calculate the distance and speed. Figure 4.9 illustrates how we can create this subsystem using ProSave components.

Regarding the computation model, execution of functions in components occurs when the input trigger port is activated. Activation of the input trigger port leads to the reading of all data at the input data ports. When a component is triggered and all data at the input ports is read, it switches to the active state, where the internal computation is performed and results are produced at the output port. The output data ports are written at the same time with the output trigger port, and the component will switch back to the inactive state. As exemplified by Figure 4.9 all of the components have an input trigger port, where
the activation of each component is done by the control flow. If data is available at the input data ports of the components, the component will stay passive until the control flow arrives and triggers the control port. The behaviour of a ProSave component that requires the control triggering at the input and output port makes it a control-driven component.

For the upper layer, ProSys, the input message port writes to its output data port and activates the output trigger port in order to transfer the data to the ProSave component or connector. The same behaviour applies to the output message port, which needs to be triggered in order for the data to be sent as a message at the ProSys layer. Similar to the ProSave components, ProSys components need control triggering at the input and output message ports, and therefore ProSys components are also driven by the control.

Even though the ProCom component model distinguishes the control flow and data flow by having explicit entities for control and data, the execution of components is control-driven. The data can only flow in the system due to the existence of control to trigger the movement. This characteristic of data flow in the system clearly shows that the separation of control flow and data flow employed by ProCom component model is not the separation of control flow and data flow that we discussed in Chapter 3. In particular, the separation is not achieved inside the component, since the computation depends on the presence of control flow at all time.

### 4.2.2 SCADE

Developed by Esterel Technologies, SCADE \cite{43} is a well-established tool developed for the avionics domain. It is endowed with a DO178B-level-A code generator, a standard for safety critical applications that has made SCADE a de-facto standard in building software systems for the avionics domain.

The SCADE tool provides the environment to assist the development of critical embedded systems. The tool consists of graphical editor, a model checker, a simulator and a code generator to transform the graphical model into C code. SCADE is often regarded as a graphical data flow programming language that originated from the Lustre \cite{28} data flow programming language.

A Lustre program operates on a flow of data values. Any variable defined in Lustre represents a flow of an infinite sequence of values $x_1, x_2, \ldots$. A component in SCADE is an architectural unit that is defined using a Lustre block. Figure \ref{fig:4.10}
shows an example of a Lustre block named A. A is called a node in Lustre program. In this example, A receives two input flows of data (x and y) and produces one output flow of data (z). A Lustre node performs its computation when all the input data are made available by the flows of values.

![Figure 4.10: A Lustre block.](image)

The Lustre block is used to model a continuous behaviour part of a system. That is, it continuously performs the same computation to the flows of data. An example of a system with regularity in the behaviour is a signal-processing system. It takes a flow of data input and performs the same computation all the time, producing output data based on the input data given.

Apart from the block diagram that models a continuous behaviour, SCADE uses the Esterel programming language [17] to model the part of the system that requires a change in behaviour in reaction to external events. This kind of discrete control is generally represented by state machines. In SCADE, the state machines in use are called Safe State Machines (SSMs), which evolved from the Esterel programming language and the SynCharts state machine [6]. This state machine that models the discrete control part of the system is put seamlessly in SCADE to work with the data flow model by Lustre blocks.

![Figure 4.11: SSM in SCADE.](image)
Figure 4.11 shows an example of a state machine used in SCADE together with the Lustre blocks. C1 is an SSM defined with two inputs, p and q, and two outputs, r and s. These inputs and outputs of C1 are defined as flows of data in Lustre. Note that an SSM block has the structure of a Lustre block that receives data as input and produces data as output. Also, the behaviour of the SSM block is similar to the Lustre block, in the sense that it needs all the data input to be available before it can perform the processing of control. The processing of control takes the input data values and decides on the mode of the system based on the automata. The decision is produced as output data for the SSM in the form of Boolean data values. These Boolean values are used as the activation condition for the Lustre blocks. In this example, both p and q must be available for the control processing to take place to produce the output r and s. The outputs r and s determine the execution of the A1 and A2 blocks, respectively. That is, the value for r and s must be true in order for A1 and A2 to perform their computation.

Figure 4.12: Separation of control and data in SCADE.

Figure 4.12 shows an example taken from [62] that models a climate control system using SCADE. A climate control system is used in a car to regulate the temperature inside, operating based on modes. There are three modes, which the user can select by pushing a button. These three modes are auto, adjust and manual.
The system takes four inputs of Boolean type, which correspond to the physical input button of the system, namely Ok, Climate, Left and Right. As output, the system produces four data values: ClimateMode, Temperature, VentilationMode and Ventilation. The system is modelled using one SSM block (called Control Climate) and three Lustre blocks that model the modes for the system (Auto, Adjust and Manual). A few data operators are used to coordinate the data flow between components, namely S (which stands for Selector), F (which stands for Fork) and J (which stands for Join).

All the lines that connect the modelling entities in SCADE represent flows of data values. These modelling entities within the graphical block-diagram notation perform operations only when the data is available. However, note that in SCADE control flow cannot be defined. Rather, state machines output control signals in a form of data (Boolean values), and these signals are used as data input to the Lustre blocks. This makes SCADE a purely data-driven model, since all modelling elements in SCADE require data availability in order to operate.

The SCADE model clearly separates the modelling of control and data. In Figure 4.12, we can see that the SSM block models the control in the architecture, and the Lustre blocks model the data in the architecture.

### 4.3 Encapsulated Components

In contrast to architectural units defined as components that have provided and required services, encapsulated components are defined as components that have only provided services at the interface. Figure 4.13(a) shows an example of an encapsulated component C that has a provided port which act as the interface to the component. Computations are encapsulated inside the components, and can only be invoked through the interface. Components do not call other components, since they only have provided services and not required services.

Components can be composed together using the exogenous composition connector [69]. The composition connector defines and coordinates the control flow between components. Figure 4.13(b) illustrates an example of composing two encapsulated components (A and B) with an exogenous connector X. Control flow originates at the top of connector X, and it flows to component A and component B via the connector X, returning back to the top of the connector. Computation inside the component happens when it is invoked by the composition connector.
that carries the control flow, making it a control-driven model. For computations
that require data from the environment, the data is passed together with the
control flow, therefore, there is no separation of control flow and data flow in this
model.

X-MAN is the only representative of component model that defined encapsu-
lated components. Thus, we take a closer look at the X-MAN component model
in the following subsection.

4.3.1 X-MAN

X-MAN \cite{77,70} is a generic software component model with encapsulated com-
ponents. Components encapsulate control, data, as well as computation. Figure
4.14(a) shows the structure of an atomic component, which is the primitive
component in X-MAN. It contains a computation unit (CU) and an invocation
connector (IC) that interacts with the external environment. The CU contains
a set of methods that does not invoke methods in another CU of another com-
ponent. This makes the computation encapsulated. The only way to access
the computation unit is via the invocation connector. The invocation connector
passes the control (as well as input data if any) from outside the component to the
computation unit. The control will return (with results) back to where it came
from after the execution of the method inside the computation unit. Therefore,
the component encapsulates the control.

X-MAN is a hierarchical component model that focused on composition mech-
anism. Components are connected or composed together using a composition
4.3. ENCAPSULATED COMPONENTS

(a) An atomic component.

(b) Composing components.

(c) A calculator example.

Figure 4.14: X-MAN components and connectors.
connector. Figure 4.14(b) shows an example of composition of components that involves atomic components A and B, as well as the composition connector CC. A composition connector defines the control flow and can be composed with other composition connectors. In X-MAN, composition connectors are control connectors that are defined to be Turing complete [65].

The composition connector passes the control (as well as data) from outside to the components, and the control will return (with data results) to the top-most connector. Figure 4.14(c) shows an example of connecting components for a calculating function that is used to calculate a percentage, given two numbers X and Y. The composition involves a component DIV to divide X with Y, and a component MULT to multiply the result with 100, and a pipe connector as the composition connector. The pipe connector defines a sequencing control flow between the entities that it connects, and carries data output from one connected entity to another connected entity. In this case, the control flow visits the components in sequence, and the data output from component DIV is carried to component MULT, in which it becomes the input. Figure 4.14(c) shows how the control flow and data flow coincide with each other for this example.

In this model, computations are completely control-driven. Computations are invoked by the presence of control. Even though control and computation are separated [75] in this model, data flows together with control. Thus there is no separation of control flow and data flow in the architecture model for X-MAN.

4.4 An Analysis of Current Component Models

In the previous sections, we presented a detailed survey of selected current component models that represent the state of the art in modelling software systems. Based on this survey, in this section, we present a detailed analysis and discuss the existing component models, with respect to desiderata described in Chapter 3 for modelling of control, data and computation. This, in turn, leads to the motivations of proposing a new component model that is control-driven and data-driven.

The desiderata for control flow and data modelling presented in Section 3.2 provide the reference and criteria for reviewing the related work. The selected representatives of existing component models are reviewed according to the criteria and summarised in Table 4.1.
As we explained in Section 3.2, control flow and data flow are the two important aspects that should be explicitly expressed in a component model. However, this is not supported by component models that define objects as components. The communication between objects is through a direct message passing mechanism, that involves a direct method call of an object by another object. There is no explicit connector involved in modelling the relationships between components, which makes the control flow and data flow implicit in the communication. This is exemplified by the EJB example described in Section 4.1.1. Objects are wired up by the method calls in the source code. Even though the method calls involve control flow and possibly data flow, this mechanism of connecting components itself is not defined as an identifiable entity. Thus, there is no explicit control flow and data flow modelling in models that define objects as components.

In models that define architectural units as components, control flow and data flow can be easily separated by components having distinct ports for control and data, and their associated connectors. For example, in ProCom, control ports and data ports are defined together with their connectors, to explicitly model the control flow and data flow. The existence of these entities makes ProCom a component model that supports the explicit modelling of control flow and data flow. However, this is different in the SCADE model, where only the data flow is explicitly expressed. In SCADE, control is modelled as a state machine, in which there is no visualisation of control flow.

For the X-MAN component model that defines encapsulated components, the control flow is explicitly modelled by connectors that compose components. Even though connectors are defined with the focus of modelling the control flow, the data flow is defined to be in tandem with the control flow. In other words, the
semantics of the connectors are defined with the consideration of the data that need to be passed between components. Because of this association, data flow is not expressed explicitly in the X-MAN model, and is always assumed to follow the control flow.

Another desirable feature that must be supported by a component model is that it should support the modelling of the whole spectrum of software systems, from data-driven systems to control-driven systems. Most of the models in all categories that are surveyed are control-driven models. These models need the control flow triggering in order to perform the computation inside the components. In ProCom, even though data flow is explicitly modelled, the components can only perform the computation when the control flow arrives at the component control port. From Table 4.1 that shows the summary of the survey of related work, SCADE is found to be the only model that is purely data-driven. This is exemplified by the semantics of SCADE, that define all the modelling elements to be driven by data. These modelling elements are connected using the data flow lines and operate when the data is available. This includes the state machine block that models the control in the system. The state machine receives input data and produces output data to communicate with other modelling elements.

Regarding the separation of control and data in system modelling, SCADE is the only model that separates control and data in the system architecture. For models that define objects as components, obviously, there is no separation of control and data, since data flow follows the control flow. In models such as ProCom that separate modelling entities for control flow and data flow, the separation is not achieved since both control and data are mixed inside the components. The computations inside the components require control triggering, making the data flow dependent on the control flow. For the X-MAN model, even though in principle, control is separated from computations performed by components, the data flow is dependent on the control flow, and therefore, there is no separation of control and data in this model.

Also, from the survey we observe some common properties of the component models in relation to the domain that they are targeted for. In particular, we observe that in ProCom and SCADE, both control and data are modelled explicitly. These component models are targeted for the embedded systems domain. This shows that the modelling of both control and data plays a significant role in the domain of embedded systems.
4.5 Motivations and Objectives for a New Component Model

In the previous sections, we surveyed and analysed the features of existing component models with respect to control, data and computation modelling. In particular, we surveyed the models in terms of explicit control flow and data flow modelling, the computation model, and the separation of control and data in the software architecture. The survey shows that the existing component models do not fully satisfy the desirable features for control, data and computation modelling. In particular, none of the models can model the whole spectrum of software systems as discussed in Section 3.2.2 since they tend to be either control-driven or data-driven. Also, they do not explicitly separate between architecture entities dealing solely with data or with control. Thus, this motivates us to propose a new component model that achieves the desirable features for control, data and computation modelling in a component model. The motivations and objectives of the new component model are outlined in Table 4.2. This table is shown as a direct comparison to the existing component models shown by Table 4.1.

<table>
<thead>
<tr>
<th>Component model</th>
<th>Explicit control flow</th>
<th>Explicit data flow</th>
<th>Control-driven</th>
<th>Data-driven</th>
<th>Separates control and data</th>
</tr>
</thead>
<tbody>
<tr>
<td>New component model</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 4.2: Motivations and objectives of the new component model.

The new component model is intended to exceed the existing component models in the following ways.

A component model with explicit modelling of control flow and data flow

We have explained in Chapter 3 the fundamental concepts in defining a computation model. From our survey in this chapter, we identify the deficits of the current component models with respect to providing explicit modelling of control flow, data flow and computation. The importance of having these elements explicit in a software model calls for the need to have a component model with identifiable entities that can represent these elements. These identifiable entities
should exist on their own and be highly independent of each other. We clarify this in the following.

![Diagram](image)

Figure 4.15: Components as computation units connected by a coordinator as the connector.

From the computation models surveyed in Chapter 3, computations exist as the entity that are connected by control flow and data flow. What we aim for in our component model is to have the control flow, data flow and computation be represented as entities in their own right. In Chapter 2 we described the current component models with the view of how components and connectors are defined. Here, we want to focus on the category where components are defined as encapsulated entities. Again, we depict components and connectors defined in this category in Figure 4.15. Here E and F are encapsulated components that perform computations within themselves. The idea of having components as encapsulated entities is particularly appealing for the aim of separating the elements for computations from control flow and data flow. We want to have a component model that defines truly self-contained units for components that have the role of performing computations only. By doing this, we can effectively model the control flow and the data flow. These should be the entities that connect the components. Having self-contained components means that the components need to be coordinated by the connector that lives outside the component, i.e. the exogenous connector. This connector should define the control flow and the data flow. In X-MAN, the exogenous connector defines the control flow structure for a system. The control connectors are mostly composable, which leads to a hierarchy of control connectors that define the control in the system. The constructed system model is expressed clearly in terms of control flow. This leads to the motivation of having an explicit control flow structure for a system architecture.

By defining the component model elements in the way that we described
4.5. MOTIVATIONS AND OBJECTIVES FOR A NEW C.M.

above, we would have explicit modelling for control flow, data flow and computation that are highly decoupled from each other. This in turn, will lead to reusable modelling elements that can be manipulated and stored in a software repository.

A component model that is both control-driven and data-driven

As explained in Section 3.2, one of the desirable features for a component model is to be able to model a spectrum of software systems that contains data-driven systems at one end, and control-driven systems on the other. In Figure 4.16, we use the same spectrum of software systems depicted in Section 3.2.2, this time illustrating the position of component models, including the component model that we intend to propose, with relation to the spectrum. At the data processing intensive end, we have the SCADE component model that is purely driven by data while on the control intensive end, we have the ProCom component model that is control-driven. These two component models represent the state-of-the-art of the current component models in building software systems. There is no component model that can support the modelling of a variety of systems in the spectrum, i.e. that can support the modelling of the whole spectrum. This leads to our motivation for having a single component model that can support the modelling of a spectrum of software systems.

The new component model should define components and connectors that can support data-driven computation, control-driven computation, and control-and-data driven computation. The realisation of this new component model that is both control-driven and data-driven enables the modelling of the whole spectrum of software systems, from data-driven systems that focus on the modelling of data flow, to control-driven systems that focus on the modelling of control flow.

A component model that separates control and data in the software architecture

In Section 3.2.3, we explained a desirable feature of having a clear separation of control and data in an architecture. This has motivated us to design a new component model that contains a separation of control and data in the system architecture. This is different, however, from our motivation of having explicit modelling of control flow and data flow that we mentioned earlier. Having a clear separation of control and data is an aim that focuses on modelling the whole system architecture, and not on individual modelling elements. In other words,
the end product of using the new component model elements should give us an architecture that clearly separates control and data.

According to our analysis in Section 4.4, the ProCom component model achieves the explicit modelling of control flow and data flow, but ProCom does not separate these two elements in the architecture. ProCom shows that the separation of control and data in an architecture cannot be achieved with merely the invention of explicit entities to model control flow and data flow. ProCom components require both control flow and data flow to perform the computations, and therefore, control and data cannot be separated in the system architecture.

For the new component model, not only do we want to have control flow, data flow and computation as explicit identifiable entities in the architecture, but also to integrate them in a way that maintains the separation of control and data for the system architecture. We want to embrace the principle of separation of control and data (or controller and plant) since it is widely accepted practice in building software for embedded systems domain.

Furthermore, we are highly motivated by SCADE in separating control and data. The two aspects of modelling (i.e. control and data) should be decoupled in a component-based approach to achieve the aforementioned benefits promoted by SCADE that we elaborated on in Section 3.2.3. SCADE achieves the separation...
of control and data in the system architecture by presenting the control as a state machine defined by Esterel, and the data part using Lustre blocks that perform data processing functions and data flow. However, SCADE does not model the control flow. The control in SCADE is defined as a state machine block that receives data as input and produces data as output. This is where our aim is different from SCADE. As we explained in Section 4.5, we are motivated by the way that X-MAN defines the control flow in a system architecture. Control flow can be composed together to form a hierarchy of control connectors that define the control flow structure for the system architecture. The control flow is expressed clearly in a visual form and this is better than having a state machine representation for control. Moreover, it is easier and more straightforward to compose two control flow structures rather than compose two state machines. Figure 4.5 shows the difference between SCADE and the new component model that we want to have. We sketch two instances of the abstract system architecture that represents the SCADE model and our model that will be constructed using the new component model. For our new component model, taking out the control (to define the control flow structure) from the rest of the system architecture will leave the data processing functions and data flows to be the data part of the system architecture. The system architecture constructed using the new component model should be able to clearly separate the part of the architecture that represents control from the part that represents data, like in the SCADE model.

![Figure 4.17: Comparing SCADE with our aim for the new component model.](image)

(a) Abstract SCADE model. (b) Our aim for the system architecture built using the new component model.
4.6 Chapter Summary

In the preceding sections, we have treated two major topics. First, we have reviewed the control, data and computation modelling in the current component models. Second, we have analysed the current component models and described clearly the motivations and objectives for proposing the new component model.

The current component models are reviewed based on a categorisation of how components are defined in the context of component models [71]. Using these categories, a survey is performed of the selected component models. The desiderata that we defined in Chapter 2 are used as the survey’s framework. A series of examples are used to illustrate in detail the control flow and data flow modelling, the computation model, and the separation of control and data in those models. Subsequently, we analysed and discussed the survey with the intention of highlighting the motivations and objectives for propose the new component model in this thesis. The explanation of the new component model starts from the next chapter, Chapter 4.
Chapter 5

The New Component Model

In Chapter 4, we surveyed current component models and identified their deficiencies with regards to the modelling of control, data, and computation. None of the models can model the whole spectrum of software systems, since they tend to be either control-driven or data-driven. Moreover, we emphasised the benefits of having a separation of control flow and data flow, which none of the existing component models has achieved, except the SCADE model.

In this chapter, we propose the semantics of the new component model. It is a novel software component model, developed with the intention of improving the existing component models in the aspect of realising the desirable features of modelling control, data and computation. The latter were described in Chapter 3. Therefore in this chapter, first, we introduce the features of the new component model, detailed in Section 5.1. Then, we present the elements of the new component model, which form the basic building blocks (Section 5.2) for building a software architecture, namely the components and the connectors. Consequently, in Section 5.3 we briefly illustrate a construction of a system architecture using the new component model.

5.1 Overview

The new component model aims to achieve the objectives described previously in Chapter 2. Recall that the main objective stated in that chapter is to have a component model that integrates the modelling of control flow and data flow, while maintaining a clear separation of these entities in modelling the system architecture. In the rest of this chapter, an abstract component model is presented.
This model generally defines the elements for building a software architecture as the design entities that we use in modelling software systems.

Computation, control, and data are the three basic elements of software systems. The new component model that serves as the modelling tool for building software systems is defined to address these elements in the model. We establish the correspondence between computation, control and data elements in software systems with the elements in the new component model. To achieve our objective of having control flow and data flow modelling, the elements for control, data and computations are made explicit and independent of each other in the model. Therefore, we define components to be associated with the computation elements, control connectors to be associated with the control flow elements, and data connectors to correspond with the data flow elements.

The key features of the new component model are (i) components perform computations only, (ii) data connectors perform data flow coordination, and (iii) control connectors perform control flow coordination. Concerning (i), the nature of components as black box entities calls for components to be independent units of functionality. They perform computations only and encapsulate their contents. They communicate through their interfaces, particularly, their data port for the data interface, and/or their control port for the control interface. Concerning (ii), complying to the data-driven computation model, the data connectors are responsible for enabling data to flow by transporting it from an output data port to an input data port. Data connectors work independently to transport the data values between computations. Similarly, for feature (iii), the control connectors are entities separate from computations and data flows, and responsible for control flow coordination. Complying to the control-driven computation, the control connectors that define the control flow are used to enable or trigger the computations to happen inside the components.

A common strand running through these three important elements (computation, control and data) in the new model is the notion of being independent of each other. Each component is a self-contained unit of computation, whose operation does not depend on other components. Control connectors and data connectors perform exogenous coordination, which means ‘coordination from outside’, for control flow and data flow respectively. By defining all of these entities in such ways, the components are loosely coupled, which in turn makes both components and connectors modelling entities that are reusable.
In the following sections, we describe all of these features in more detail. Specifically, we present and explain the basic building blocks for the new component model.

5.2 The Building Blocks

The building blocks defined for the new component model serve the computation models that we described in Chapter 3. These are the computation model for data-driven systems, control-driven systems and control-and-data-driven systems. To make the model data-driven, we define components that are purely data flow units of computation, as in data flow programming languages [58], and define connectors for these components as data channels. To make the model control-driven, in addition we define components that are data flow units of computation that also have control ports, and define connectors for these components as control structures, coordinating control flow between the components.

5.2.1 Components

Components are software entities that represent units of data or computation, with input and output data ports. The input and output data ports are the interfaces for components to communicate with the environment. Data ports are typed and practically we used the type specification in Java. A component is self-contained in the sense that it performs just computation within itself, and it does not invoke computation in other units. This characteristic of components makes them independent and reusable. We define three categories of components in our model: (i) source and sink components, (ii) pure data flow components, and (iii) hybrid components. These will be described in turn in the following sub-sections.

5.2.1.1 Source and Sink Components

Source components are data sources and sink components are data sinks. The input and output ports of these components are data ports. A source component does not have any input data ports, while a sink component does not have any output data ports. Figure 5.1(a) depicts a source component SC with one output data port and Figure 5.1(b) depicts a sink component SK with one input data
port. These special types of components model the environment that a system interacts via data exchange.

(a) A source component.  
(b) A sink component.

Figure 5.1: Source and Sink components to model the environment.

**Example:** Consider a room temperature controller system that interacts with the environment by receiving data input from the room’s temperature sensor, and sending data output values to adjust the room temperature by setting the level of a ventilation fan. Figure 5.2 illustrates the data source and the data sink that are used to model the environment for the temperature controller system. The source component **Temp Sensor** supplies the temperature values to the system, and the sink component **Vent Fan** receives the ventilation fan setting from the system.

![Diagram of temperature controller system](data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAABCAAAAMCAYAAABmMvEAAAAGXRFWHRTb2Z0d2FyZQBBZG9iZSBJbWFnZVJlYWR5ccllPAAAAyFpVFh0WE1MOmNvbS5hZG9iZS54bXAAAAAAADw/eHBhY2tldCBiZWdpbj0i77u/IiBpZD0iVzVNME1wQ2VoaUh6cmVTek5UY3prYzlkIj8+IDx4OnhtcG1ldGEgeG1sbnM6eDwsY3JvdGF highlights...}

Figure 5.2: Source and sink components in a temperature controller system.

**5.2.1.2 Pure Data Flow Components**

A pure data flow component is an independent unit of computation that consists of a data transformation function transforming input data to output data. Such
a component has only data ports: input data ports and output data ports. Figure 5.3 shows an example of a simple data flow component with one input data port and one output data port.

The execution of the function inside the component is data-driven, i.e. it can only take place when all the input data ports are filled up with data. The processing of the data causes the input data ports to be available again for any new data. The execution of the function is done in three steps: reading the input data at the input ports, transforming the input to output by data transformation, and writing the output data to all the output ports at the same time. If new data arrives at the input port during this process, it has to wait for the current function execution to complete, before being transferred into the input ports.

This type of component is suitable for modelling a computation that continuously processes input data values and produces output data values. For example, in the embedded systems domain, a pure data flow component can be used to model a function that processes input data values from the sensors and produces continuous output data values. Another domain that can benefit from the use of pure data flow components is the domain of signal processing systems. An example of a signal processing system is a speech recognition system that converts speech to text. The main function of this type of system involves continuous transformation of a stream of input data values (the speech) to a stream of output data values (the text).

Example: A fuel gauge on a car’s dashboard performs a function to display the level of fuel in the petrol tank. Figure 5.4 shows a pure data flow component that is used to model this function. The data flow component fuel gauge takes fuel-level readings from a sensor in the fuel tank as input data, and transforms the fuel level in centimetres to a percentage. The component has one input data port to model the fuel-level readings parameter, and one
output data port to model the percentage value of fuels in the tank. The component will operate all the time to perform the data transformation, as long as fuel-level readings are available. The data availability drives the computation inside the component.

![Fuel gauge component](image)

Figure 5.4: Fuel gauge component: an example of a pure data flow component.

5.2.1.3 Hybrid Components

The processing of data by data flow components can be managed by imposing an execution order on their operations. This can be done using control in the systems. Therefore, we define hybrid components to work under the influence of both data flow and control flow. These are pure data flow components, but with the addition of control ports. A control port is activated when control flow reaches it. The control port does not have a type but it can only be connected to the parameter of a control connector and can be connected to multiple control connectors. The execution of a hybrid component’s data transformation function is both data-driven (like a pure data flow component) and control-driven. There are two types of hybrid components: E/D components and TR components. E/D stands for Enable/Disable, while TR stands for Trigger. These components have states, which change when control flow reaches their control ports. Figure 5.5(a) depicts an E/D component that has one input data port, one output data port and a control port. It has states \{Enabled, Disabled\}, and its behaviour is described by the state transition diagram in Figure 5.5(b).

An E/D component switches between these two states when its control port is activated by control, and when in the Enabled state it will execute its data transformation function $f_1$ whenever data is available at all its input data ports. If control is available at the same time as data when a component is in the Enabled state, the data transformation function executes first, then the component
switches to the Disabled state. This type of component is suitable for modelling computations that continuously transform input data values to output data values, but can be switched on or off. For example, in the domain of automotive systems, there are functions that involve continuous behaviour of receiving input data values from sensors and performing computations continuously to produce data output, with these functions being switched on/off depending on the mode of operation.

**Example:** A cruise control system in a car performs a function that requires adjusting the throttle value based on the speed that is set by the driver. The function involves taking the current speed produced by the speed sensor and performing the calculation of adjusting the throttle value according to the speed that is set by the driver. There are two data inputs to the function, which are the current speed and the set speed, and one data output which is the throttle value. When the cruise control function is switched on, it adjusts the throttle value, according to incoming data from a speed sensor. The computation continues as long as input data is available, until the cruise controller is switched off. Figure 5.6 shows the component cruise controller with parameter current speed and set speed as the data input, and parameter throttle value as the data output.

The structure of a TR component is depicted in Figure 5.7(a). The component C2 that performs function f2 has one input data port, one output data port and a control port. This type of component has only one state, {Passive}, and its
behaviour is described by the state transition diagram in Figure 5.7(b). A TR component is always in the Passive state. However, when its control port is activated by control, it executes its data transformation once, but only if data is available at all its input data ports. It is suitable for modelling a function that performs computation once when it is required. In a way, it is possible to make the computation control-driven if the data is made available all the time at the input data ports (this will be explained in Section 5.2.2.1). A TR component can be used to model devices in which the operation requires control triggering, e.g. a coffee machine that consists of devices that dispense coffee, sugar and hot water. These devices need to be controlled by triggering the dispensers to dispense the appropriate amount of coffee, sugar and hot water to make a cup of coffee.

Example: A comfort function in a car includes setting the car seat, mirrors (side and rear) and the radio tuner according to the driver preferences. These preferences are stored in the memory together the driver ID. The driver can be identified by the ID that is stored on the car’s ignition key.
When the driver inserts the ignition key, the preferred settings for mirrors, driver’s seat and the radio for the driver are read from memory. Figure 5.8 shows a TR component comfort function that is used to model this function. It has three data inputs to model the preferred settings for mirrors, seat, and radio, and three data outputs to model the adjustment value required for the mirrors, the seat and the radio. The component will perform the computation of calculating the adjustment for the mirror, the seat and the radio based on the setting when the control flow triggers the control port, and the data is available at the input data ports.

![Comfort function component](image)

Figure 5.8: Comfort function component: an example of a TR component.

### 5.2.2 Connectors

A collective form of functions can be achieved by connecting components using connectors. Connectors are the component model entities that are used to establish connections between two ports. They are exogenous connectors that are used for composing software components, and also as indicated by the name, they are separate from software components. There are two main types of connectors in our model: (i) data connectors and (ii) control connectors, for connecting data and control ports respectively. We will introduce each of these connectors informally, in turn. In addition, we also define data coordinators that coordinate data flow between data connectors.

#### 5.2.2.1 Data Connectors

Data flow coordination in our model is performed by data connectors. A data connector is a directed edge which connects two data ports — an output data...
port to an input data port. Connected data ports must have compatible types. A data connector is a pipe that transfers data from its source (the output data port) to its sink (the input data port). Whenever a data item is produced at the output data port, it is transferred by the data connector to the input data port (that it is connected to).

We define three main types of data connectors using the semantics of data channels in the REO coordination language [8]:

**Unbounded FIFO** This type of data connector has an unbounded FIFO (first in first out) buffer between the source and the sink. This means that it can store an unlimited number of data items in the buffer. An unbounded FIFO queue enables the connected entities to work independently, consuming and producing data asynchronously. It is called a **FIFO** channel in REO.

Figure 5.9 shows an example of two connected components: component C1 and component C2 that perform function A and function B respectively. Component C1 has an output data port p2 that is connected to the input data port p3 of component C2. The data connector is the edge between p2 and p3 that actively transports the data. The unbounded FIFO data connector used between port p2 and port p3 ensures that all data produced by component C1 will be consumed by component C2 in order. Once consumed, the data is deleted from the FIFO buffer.

![Unbounded FIFO data connector](image)

Figure 5.9: An unbounded FIFO data connector connecting p2 and p3.

This type of data connector is suitable for modelling asynchronous data production and consumption between components. Since data can be stored infinitely in the channel, there is no issue of losing data if the consumer component does not read the data immediately when it is produced by the producer component.
5.2. THE BUILDING BLOCKS

Size one FIFO The behaviour of this type of connector is similar to the unbounded FIFO, except that the FIFO queue for the data connector has a buffer of size one. Such a connector might incur data loss when a new value from the source replaces the value in the buffer that is waiting to be consumed by the entity at the sink. It is called a \textit{shift-lossy FIFO} channel in REO. Figure 5.10 shows an example of two connected components; component C3 and component C4 that perform function C and function D respectively. The size one FIFO data connector is used to connect components C3 and C4 via port p6 and port p7. Similar to the unbounded FIFO connector, the latest data value is put into the buffer, and once it is consumed at the sink, the data will be deleted from the buffer.

![Size one FIFO data connector](image)

Figure 5.10: A size one FIFO data connector connecting p6 and p7.

This type of data connection is suitable for asynchronous data passing between components, in which only the latest value for the data parameter is taken into account. The example in Figure 5.10 ensures that component C4 will always get the latest data value from component C3, since the data connector can only store one data value.

Non-destructive read size one buffer In contrast to a FIFO channel, that deletes the value in the channel once it is copied to the sink (component input port), the non-destructive read size one buffer does not delete the data value when it is copied to the input data port at its sink. The data value remains in the buffer until the new data arrives at the source and the current data value is overwritten with the new value. It is called a \textit{variable} channel in REO.

Figure 5.11 shows an example of two connected components: component C5 and component C6, that perform function E and function F respectively. The non-destructive read size one buffer data connector is used to connect
components $C_5$ and $C_6$ via data ports $p_{10}$ and port $p_{11}$. The size one FIFO data connector ensures that $C_6$ will always receive the latest value produced by $C_5$. Since the value is not deleted from the buffer when it is consumed, the data value persists as a constant value if component $C_5$ stops producing any output values.

![Non-destructive read size one buffer data connector](image)

Figure 5.11: A non-destructive read size one buffer data connector connecting $p_{10}$ and $p_{11}$.

Data connectors behave according to their semantics regardless of the types or states of components to which they are connected. For instance, a FIFO data channel connected to a disabled E/D component still stores all incoming data in its buffer. The stored data will be processed by the E/D component after it is enabled. Different behaviour can be achieved by changing the data connector. For example, a size one FIFO channel, which does not buffer old data, is suitable for processing only the most up-to-date data.

One data output port can be connected to multiple data input ports. There is no explicit construct to model the data forking connections besides connecting these ports using the data connectors. The same principle is applicable for many-to-one connections as a result of connecting multiple data output ports to one data input port. However, this implies a non-deterministic choice of which data is written to the input data port if multiple data items arrive at the same time. As a result, data loss may occur. To prevent unintended data loss, it suffices to avoid such many-to-one connections. In addition to data connectors, we define data coordinators (this will be explained in Section 5.2.2.2) as entities that perform data coordinations between data connectors.

### 5.2.2.2 Data Coordinators

Data coordinators coordinate data flow between data connectors. A data coordinator forwards data, according to some condition. It has input data ports
and output data ports that are connected to data connectors. The type of a connected data port must be compatible with that of the port at the other end of the data connector. Figure 5.12 illustrates the two types of data coordinators that we have defined for our model: a data switch and a data guard.

![Data coordinators](image)

**Figure 5.12: Data coordinators.**

**Data switch** This coordinator has two data input ports, of which one is connected to a source that provides decision data, and the other is connected to another source which provides the data that needs to be forwarded. A data switch has multiple output data ports, but exactly one of them will be selected to be filled with data. The selection is based on the data condition that it receives. The data condition input is a value of type integer that represents which data output port (of the data switch) should be active in order to forward the data input. Figure 5.13 shows an example of a data switch used to forward data selectively based on a data condition. In this example, we have a data switch SW1 that has two data output ports (p17 and p18), connected to component C9 and component C10. The input data ports of SW1 are connected to component C7 and component C8. Component C7 produces a data value for the data condition of SW1, and component C8 produces a data value that should be consumed by either component C9 or C10. Based on the behaviour of a data switch, either port p17 or port p18 will be filled with a data value. The data switch SW1 will select which one of the data output ports is to be written to, based on the data condition received at the input data port p15 of data switch SW1.

**Data guard** The data guard has a similar behaviour to a data switch, except that it has just one data output port. It only forwards the data to the output port if the data condition is met. In the case of a data guard, the data condition is of Boolean type such that when the value is true, the data input will be forwarded, or else the data will be deleted. Figure 5.14
shows an example of a data guard used to forward data only when the data condition is met. In this example, we have a data guard GD1 that has one data output port p25 that is connected to component C13. The input data ports of GD1 are connected to component C11 (at input data port p23) and component C12 (at input data port p24). Component C11 produces a data condition for GD1, and component C12 produces a data output to be consumed by component C13. Based on the behaviour of a data guard, the data at input port p24 will only be forwarded to the output data port p25 when the data condition received at p23 is true.
5.2.2.3 Control Connectors

Control flow coordination is performed by the control connectors. The main characteristics of control connectors are that they define control flow in our model and they are exogenous. They stand as separate entities that compose software components, and they initiate control in the system. The control connector is entirely responsible for handling the control flow between components, making them truly independent and loosely coupled.

A control connector has a control port and can connect an arbitrary number of control ports (of hybrid components or of other control connectors). Some control connectors have a data input port where they can receive decision data input for control flow routing. The data input port must be connected to an output data port of compatible type. Figure 5.15 depicts the structure of a basic generic control connector.

![Figure 5.15: A basic generic control connector.](image)

A control connector defines control flow as follows: it receives control through its upper control port, then passes it to the (lower) control ports it connects, and finally passes control back out through its upper control port. This is exemplified by Figure 5.15 where the control flow starts from the control port of the control connector CC, passes to the first and second leg, and returns back to the top where it originates. This shows that a control connector *encapsulates* control.

Control connectors are used to connect hybrid components and control connectors. The fact that our control connectors that can be connected with other control connectors enables the formation of a complex control flow. In other words, control flow can be structured in a system by having control connectors connect to one another. This is illustrated in Figure 5.16 where the control flows of two binary control connectors, CC1 and CC2, are composed to yield a more
complicated control structure.\footnote{Control flow cycles in the control structure are not allowed, i.e. control connectors form a directed acyclic graph. The control structure is detailed in [64].}

Figure 5.16: Composing control flows.

The encapsulation of control by control connectors entails a hierarchical approach to connector definition and construction [64]. A complete modelling of a system that involves control flow modelling will consist of hybrid components that lie flat at the bottom of the system architecture, with a control flow structure standing on top of these hybrid components. The control flow is structured in a form of hierarchy of control connectors. This hierarchy of control connectors can form a composite control connector. These composite control connectors are compositional and reusable, making them suitable for being deposited in a software repository for reuse. However, in this thesis, we do not work extensively with control connector composition.

There are four basic control connectors in our model: (i) sequencers, (ii) selectors, (iii) guards and (iv) loops. These correspond to elementary control flow constructs: sequencing, (conditional) branching and looping. Figure 5.2.2.3 illustrates all the basic types of control connectors in our model.

**Sequencer** This is an $n$-ary control connector that composes components or other control connector, and defines a sequential control flow between the entities that it composes. A sequencer routes control to its connected ports in sequence. An example of a binary sequencer is shown in Figure 5.17(a), together with the direction of control flow. In this example, the sequencer routes the control flow to each parameter that it is connected to sequentially, in an order that is fixed. When control is fed to the control port of the
5.2. THE BUILDING BLOCKS

(a) Sequencer. (b) Selector. (c) Guard.

(d) Loop.

Figure 5.17: Control connectors.

sequencer, the control flow will visit the first parameter and then the second parameter, before it returns back to the control port at the top.

Selector A selector is an \textit{n-ary} control connector that composes components or other control connectors. It has a data input port that receives a data condition of type integer. The selector connector defines a branching behaviour that chooses exactly one of all the connecting parameters for control flow routing. The “switch” or “if-then-else” statement in programming languages is a control structure analogous to the selector behaviour. The routing decision is made based on a pre-defined condition received at the data input port. Figure 5.17(b) depicts a selector connector that has two branches. This connector routes control flow to exactly one of its connected ports, selected on the basis of a data input (an integer denoting the position of the port to be selected). A selector thus imposes a branching behaviour for the control flow. The control flow is routed to the branch that satisfies the condition, which causes the visit of control flow to the branch before returning back to the control port at the top. Selectors require synchronisation, since they are points where control flow and data flow meet. Control flow stops and waits in a selector until data is available at the selector’s data input port.
Guard  A guard is a unary connector that imposes a mechanism to restrict the flow of control. In a programming language, it is normally expressed as a Boolean expression that must be “true” in order for the execution to occur. Figure 5.17(c) depicts a guard that conditionally forwards the control flow to its only connected port. It needs a data port that supplies a Boolean condition. Like in a selector, control stops and waits in a guard until its data input is available.

Loop  A loop is also a unary connector. Figure 5.17(d) depicts a loop connector that represents an infinite loop that keeps routing control to its only connected port. Unlike other control connectors, a loop connector never returns control via its port. Rather, after receiving control, it repeatedly passes control to its only connected port. Therefore, a loop can only be used as the top-level connector, or more commonly used as the top-most control connector in a control connector hierarchy that runs perpetually. Note that there can be more than one control connector hierarchy defined in a system, therefore, it is possible to have multiple loops in a system.

5.2.3 Composing Control Flows

The control connectors are used not only for composing components, but for connector composition as well. This means that the control flow is composable in our model. But not all of the defined control connectors described in Section 5.2.2.3 are composable. Sequencers, selectors and guards are composable, while loops are not. This is because of the behaviour of the loop connector that runs perpetually and does not return the control to its control port.

We use a simple cruise control system example to illustrate the scenarios of composing the control flow in our model. With this example, we aim to show how the control connectors are used to compose components and to construct a control flow structure, by connecting a group of control connectors together. The constructed cruise control system is shown in Figure 5.18. It is modelled using hybrid components, control connectors and data connectors. Note that in this example, we only focus on the control flow part of the system, i.e. we focus on the control connectors only.

There are two modes of operation for the cruise control: automatic and
5.2. THE BUILDING BLOCKS

Figure 5.18: Constructing control flow: a cruise control example.
manual. Only one of the modes can be active at a time. However, there exists a third mode in the cruise control system where the speed is recorded and adjusted when the mode changes between automatic mode and manual mode. In the following paragraph, we describe the details of the system constructed in terms of composing the control flow.

Automatic mode requires the system to perform the operations **adjust throttle**, **maintain speed** and **display speed** in sequence. These operations are modelled using TR components. A sequencing control flow is modelled using a sequencer connector that composes the three components. Similarly, for manual mode, a sequencer connector is used to model the sequencing operation of **clear throttle reading**, **manual cruising** and **display speed**. Note that since the display speed operation for cruise control system is performed for both automatic and manual mode, the display speed component control port is connected to two parameters from two different control connectors. This one-to-many connection for control ports is only applicable to connections between components and control connectors. For the third mode, which is the transition mode between automatic mode and manual mode, the system performs the operations **record speed** and **adjust speed** in sequence. On top of the three sequencers, a selector is used to model the branching behaviour of selecting the mode of operation for the cruise control. Based on the data condition that the selector receives from the data source, there will be exactly one branch that is selected for control flow routing. At the very top of the hierarchy, a sequencer is used to ensure that the **calibrate** function is performed before selecting the cruising mode.

The constructed system shown in Figure 5.18 is ready for execution. The system can be executed by feeding control to the topmost level connector (in this case, it is the sequencer) and loading the data source with data values (the data values are the data input to the system). One possible control flow path in the system execution is denoted by the dashed line in Figure 5.18.

This example shows the result of composing hybrid components with control connectors. That is, the system architecture will consist of hybrid components lying on the flat bottom line of the system architecture in the model, connected by a hierarchy of control connectors on top that form the control structure of the system. From this example, we show that it is possible to construct a complicated control flow structure using our control connectors. In other words, the control flow is composable in our model. Also, this example shows that the
hierarchy of control connectors, formed by the basic control connectors described in Section 5.2.2.3, has become an entity that is specific to a cruise control system. We can make this entity a composite control connector that can be stored in the repository for reuse. This is one of the distinct features of our model, which is viewed as valuable in the context of a domain.

If we want to have a running cruise control system that involves a continuous feeding of control flow to the hierarchy of control connectors, we need to connect a loop connector to the topmost level connector, which in this case is the sequencer connector. The loop connector needs to be fed with control (via its control port) at the system startup, and the control will run perpetually. This control connector hierarchy, however, no longer composable at the topmost port since the loop connector is not composable. It is possible to have such control connector hierarchies at multiple places in a system. For instance, besides the control connector hierarchy for the cruise control system, there exists a fuel system that contains a control connector hierarchy which has its own source of control via its top level loop connector. The cruise control system and the fuel system are the subsystems for a bigger system, which is the engine system for a car.

5.3 The System Architecture Constructed Using the Component Model

Having introduced the building blocks of the new component model, in this section we illustrate the system architecture constructed using the component model. We use the example of an aircraft door indicator system, a hot beverage maker system and a car fuel system for illustrative purposes. Here, the aim is to give an overview of how the component model’s building blocks introduced in this chapter are used to build the software architecture to model various types of systems. Along the way, we recapitulate and relate the modelling of control, data and computation to achieve the objective of having a component model that supports a spectrum of software systems described in Chapter 4.

\footnote{This is discussed in more detail in Chapter 8.}
5.3.1 Data-Driven Systems

Data flow components, data connectors and data coordinators are the component model elements that are data-driven. Figure 5.19 shows an aircraft door indicator system constructed using the data-driven elements of our component model. The system is required to continuously take data inputs from the sensor and process them to turn on/off the respective light to indicate whether the door is open or closed. In Figure 5.19, the source component feeds the door sensor input to the MS component. This input is processed by component MS to produce decision data which are the door locked flag and current status as the output. The current status output data will be set to 1 if the door is locked, otherwise it is set to value 2 if it is unlocked. This data value is fed into the switch (SW) data coordinator as decision data for forwarding the input data direct current value (dc value) to either component SDL or SDUL. If component SDL receives the dc value, then the red light command data is generated and sent to the component RL to produce the red light that indicates that the physical aircraft door is locked. Likewise, if the component SDUL receives the dc value, then the green light command data is generated and sent to the component GL to produce the green light to indicate that the physical aircraft door is unlocked. Additionally, the input from the sensor is processed by component MS to generate decision data to pressurise the cabin accordingly. That is, when the door is locked, then the door locked flag is produced by component MS with a Boolean value TRUE to enable the forwarding of data pressure setting through the data guard GD. The component PC that is responsible for pressurising the cabin will receive the data pressure setting and generate the set pressure command data to component CS. The sink components in this example represent the physical devices producing the lights and pressurising the cabin that take data in the form of commands from the system.

In this example, the computations involve transforming data input to data output and work continuously as long as data inputs are available. Therefore we use pure data flow components for monitoring the sensor reading, generating the command for showing the status of the door, and pressurising the cabin. Although the focus is on the data transformations, there exists ordering and selection in the execution of the computation imposed by the data flow. This means that there is a control in the system that coordinates the operations of the data transformations. The system requires component MS to process the
5.3. SYSTEM ARCHITECTURE CONSTRUCTED USING THE C.M.

Figure 5.19: A door latch system that is data-driven.

door sensor reading first before deciding to forward the input data dc value for components to show the light at the door as well as pressurising the cabin. This ordering is coordinated by the data flow with the help of a data switch and a data guard that wait for the decision data input from component MS.

From the example, it is clear that we are able to build a data-driven system using our component model constructs that are driven by data. Data flow is expressed in the behaviour of the data connector and the components model solely data transformations that wait for the data to be available to perform the computation. This type of system which resides at one end of the software systems spectrum explained in Section 3.2.2 focuses only on the data transformations in the system to transform the input data to output data. It is not important to show the control in the system, although the control exists in the form of the data availability that triggers the computations. We show that it is possible to impose control in data-driven systems using data flow coordination. We don’t express the control flow explicitly since the focus is on modelling the continuous behaviour of transforming data input to data output, where control is not meant to be explicit. Here we achieve our objectives in having explicit data flow modelling in the CBSD approach and provide the modelling elements that can be used to
model a part of the spectrum of software systems that are data-driven.

### 5.3.2 Control-Driven Systems

Control connectors and hybrid components are the component model elements that are influenced by the control flow. Figure 5.20 shows a hot beverage maker system example which upon receiving input from the user dispenses a hot drink that can be one of the following types: coffee, hot chocolate or tea. Since the system only works based on the occasional user input as when it is required to dispense a hot drink, it is constructed as a control-driven system. The steps in making the hot drink consist of putting the drink powder (coffee/cocoa/tea) in the cup and pouring hot water.

The main system behaviour depends on the control flow visits to trigger the operation performed by the components. The control structure for the system is constructed based on the combination of branching for the control flow routing in selecting which type of drink to be dispensed and control flow sequence visits to components to make the drink. The whole system is constructed by composing four TR components (for dispensing the drink powder and the hot water) at the bottom hierarchy of the system using control connectors that are structured according to the behaviour required for the system.

In Figure 5.20 the TR components (CD, HCD, TD and HW) are composed together in a particular order to dispense coffee powder, hot chocolate powder, tea bag and hot water. We use TR components for this type of system since the computation need to be performed once as when it is required (i.e. visited by the control flow). These components receive the input data from the source component S via the non-destructive read size one buffer data connector. This data connector type is used to ensure that data input for these components is always available, therefore making the control flow the main driver for the operations of the component.

Components are triggered by the visits of the control flow that comes whenever there are inputs from the user. In particular, the inputs are the data that are fed into component PI that processes and produces the necessary data for the guard GD and the selector SEL control connectors. Note that the control connector guard is placed on top of the selector connector since we want to enable control flow routing only when the user turns on the machine. The switch on and user selection data represent the user input for turning on the machine and selecting
the drink to be dispensed respectively. Upon receiving these data, component PI processes the input and produces the machine on flag data and the dispense selection data output. The machine on flag data carries the Boolean value which is set to TRUE to indicate that the machine is turned on in order for the control flow to go through the guard GD connector and route the control flow to the lower level connector for the machine operations. The dispense selection data is of integer type that carries the value necessary for the branching decision to route the control flow at the selector SEL connector. Based on the user selection, the control flow is routed to one of the sequencers and continues to trigger components in sequence. In this example, for any selection of drink, the hot water component will be triggered to dispense the hot water after the drink component (coffee, hot chocolate or tea) is dispensed. The sink components that represent the cup that contains the drink receive the data output from the appropriate drink powder dispenser and the hot water dispenser.

Figure 5.20: A hot beverage maker system that is control-driven.
CHAPTER 5. THE NEW COMPONENT MODEL

From the example, it is evident that we can construct a purely control-driven system using our control connectors and TR components. Data are made available for data transformations in order for control flow to drive the execution of the system. Here the order of computation execution is of prime importance, therefore the control flow is defined explicitly using the control connectors.

The hot beverage maker system is an example of a system where data flow is not the primary concern, although the computations (performed by TR components) require the availability of data. To ensure the availability of data at all time to these components, we use the non-destructive read size one data connector. The main focus of a control-driven system is on the ordering and the coordinating of computations.

This example shows that we have achieved the objective of having explicit control flow modelling in the CBSD approach primarily by building the hierarchy of control connectors. We are able to construct a pure control-driven system that is part of the software system spectrum that we explained in Section 3.2.2.

5.3.3 Data-and-Control-Driven Systems

Recall from Section 3.1 that there exist systems that combine non-trivial control and data processing parts. Our component model elements that consist of control-driven elements, data-driven elements and both control-and-data-driven elements can be used to construct such systems. Figure 5.21 shows a car fuel system example that is both control-driven and data-driven constructed using our component model. The fuel system is used in a hybrid car that has two types of engines, i.e. a gasoline engine and an electric engine. It manages the fuel consumption by operating using a combination of both engines. The usage distribution of both engines depends on the driving mode of the car – stop, acceleration and highway driving.

There are three main modes of operation that represent the three possible states of the system. The system receives the following as the input: the car driving mode called mode, the pedal input and the shaft input. These inputs are processed according to the current state of the system. The system behaviour is illustrated by the state transition diagram in Figure 5.22. The input mode is the parameter that causes the change of states of the fuel system. When the system starts, the initial driving mode is the stop mode where component SP is activated and components IA and HD are deactivated. The stop mode changes
5.3. SYSTEM ARCHITECTURE CONSTRUCTED USING THE C.M.

Figure 5.21: A fuel system that is both control-driven and data-driven.
to the acceleration mode when the driver presses the pedal to increase the speed of the car. The transition of the state causes the deactivation of component SP (denoted by D:SP) and activation of component IA (denoted by E:IA). From the initial acceleration mode it changes to the highway driving mode when the speed stabilises. This transition will deactivate component IA (denoted by D:IA) and activate component HD (denoted by E:HD). Then the system will go back to the stop mode when there is a decrease in the speed of the car by deactivating component HD (denoted by D:HD) and activating component SP (denoted by E:SP). In addition to the deactivating and activating of components to represent the modes, the system is required to display the current driving mode on the dashboard and display a red light whenever the system is in the stop mode.

![Diagram showing fuel system operating modes](image)

Figure 5.22: Fuel system operating modes.

The system architecture involves the modelling of both control-driven and data-driven computations. The three modes of operation are modelled with E/D components since the computations are driven by the continuous input data that are transformed to output data with enabling and disabling being done via the control port. The display of modes and the red light should be performed as the consequence of the mode changes, which involves one-time simple data transformations that are modelled by TR components and driven by control flow visits. The control in the system is required to coordinate the triggering of these TR components after enabling or disabling the E/D components.

As illustrated in Figure 5.21, the resulting architecture model contains one part that is dedicated to control and another part that is dedicated to data. The control part consists of a hierarchy of control connectors that defines the control flow in the system. The data part consists of modelling elements for data
5.3. SYSTEM ARCHITECTURE CONSTRUCTED USING THE C.M.

The data processing part is modelled using source and sink components, data flow components, hybrid components, data connectors and a data coordinator. The data source component S supplies the data input to the system. The data are fed to the data flow component MC and to the hybrid components (SP, IA and HD). These hybrid components are used to model the three modes of operation for the fuel consumption. Component SP corresponds to the mode stop, component IA corresponds to the mode initial acceleration, and component HD corresponds to the mode highway driving. A data coordinator GD is used to coordinate the data between the data connectors that supply the data condition to the selector connector. The TR components DDM and DRL are used to generate the commands for displaying devices according to the change in the behaviour of the system (i.e component DDM will display the new mode via the DIS and component DRL will display the red light via the R-L whenever the current mode is stop). G-E, E-E, R-L and DIS are data sink components that receive the data output from the system.

The component MC is responsible for keeping track of the current mode and detecting a mode change from the input. The parameter current mode is forwarded to the selector connector by the data guard GD only if there is a change in mode.

As explained in Section 5.2.3, the control connectors can be composed with each other to form a complex control flow. In this example, the hierarchy of control connectors consists of three sequencer connectors at the lowest level, one selector connector and one guard connector at the same level, another sequencer and one loop connector at the very top level. The hierarchy of control connector defines the control flow structure of the system including performing the operation of enabling and disabling the three hybrid components SP, IA and HD, and triggering of TR components DDM and DRL.

The disabling and enabling is done in sequence by the lowest level sequencers. This is because for each transition of mode (depicted in Figure 5.22), there will be an action of disabling the component that represent the previous mode and enabling of the component that represents the current mode. There are three possible actions for disabling and enabling the E/D components, which the selector connector will decide upon based on the decision data. Subsequently the control flow will return and trigger one or both of the TR components. Component DRL
can only be triggered when the system is in the stop mode. The control flow to this component is filtered by the guard connector. A continuous control flow is received from the top level loop connector, denoting an infinite loop as the source of control for the system.

The selector connector is used to route the control flow whenever there is a mode change which requires the action to perform the disabling and enabling of the E/D components. The data flow component MC sends this decision data to the selector connector and the guard connector. However, for the selector connector, we employ the data coordinator GD to coordinate the data transfer for the decision data. This is because the selector connector can only forward the control flow whenever there is a mode change. If we do not use the data guard GD, the selector connector will route the control flow for each cycle of control flow iteration according to the current mode. This will cause the disabling and enabling of the E/D components for each cycle regardless of the mode, which is incorrect. The defined E/D component in our component model only has one control port that toggles between disabling and enabling state, and this requires the control flow to be filtered before it can be forwarded to the control port of the component.

For data flow, we use all three types of data connectors to fulfil the behaviour required for the data transfer. We use the size one FIFO data connector to ensure that E/D components consume the current data from the working sensor. This means that if any of the sensors (that supply the input data to the system) fail to work, the E/D component will not be able to perform any data transformation operation. Similarly, component DRL is connected to the data sink R-L via the size one FIFO data connector that transfers the display red light command data generated each time the car enters the stop mode. In this example, a FIFO data connector is used for connections that require all data to be buffered and processed in order. All hybrid components are connected to the sink components using FIFO data connectors. The data inputs to both TR components are made available at all time since the control flow should drive the operation of the components. Thus, the non-destructive read size one data connectors are used for input to these components.

The fuel system example shows how control-driven and data-driven elements are used to construct a control-and-data-driven system. In the example, we can
clearly see the importance of data processing (computations) as well as the structuring of control to coordinate the data processing. Although the main part of the system is focusing on transforming input data to output data, the control flow plays an equally important role in coordinating these transformations. We obtain a clear representation of control in the system in a visual form with a structured control flow that is expressed using a hierarchy of control connectors.

These examples show how the building blocks of the new component model are put together to construct architecture models for various type of systems. What is not yet specified is the execution semantics for the new component model elements, which deserves a chapter on its own, namely Chapter 6.

5.4 Separation of Control and Data in the System Architecture

In the preceding section, we have demonstrated the construction of system architectures using our component model. The examples represent various kinds of systems that exist in the spectrum of software systems. To this end, we have shown that our component model consists of constructs and elements that explicitly model control, data and computation. Our aim in this section is to further demonstrate the distinct features of our component model, particularly the separation of control and data in the system architecture built using our component model.

The example in Figure 5.21 shows that the system architecture constructed using our component model can be clearly divided into two parts: control and data. We consider any data transformations, any elements that transfer data (which include data coordinators) together with the source and the sink, as the data part of the architecture. The hierarchy of control connectors forms the control part of the architecture since this is where the control is processed making it the central place for control in the system. In the analogy of having a central place for control, we call the control connector hierarchy the “controller” of the system.

If we take a closer look at the individual elements of the component model, we can make a specific categorisation based on control and data. That is, some of the elements fall under the category of purely control-driven elements that are focusing on control flow, some fall under the category of purely data-driven
elements that are focusing on data flow, and some fall under the category of control-and-data-driven elements that are focusing on both control flow and data flow. For example, a data flow component falls under the category of a pure data-driven elements that together with the data connectors model the data processing function and data flows for a system. A pure data flow component is used whenever we need to create a data transformation function that continuously processes input data to output data as long as the system is up and running. Here, the data availability is enough to act as the control in triggering the computation for the component. On the other hand, we have hybrid components that are control-and-data-driven, where the computation inside the component is driven by both control flow and data flow. These types of components perform data transformation function like the pure data flow component, but since we want to realise the computation based on events that happen occasionally, we employ the use of explicit control flow to enable/disable or trigger the computations. This means that the control not only exists in the form of data availability at the data port but also at the control port of the component. For control connectors, the selector and guard depend on the data availability for the routing of control flow. They wait for the data to arrive at the data port before they can proceed with the routing of the control flow. From these examples, we can conclude that even though control flow and data flow exist independently in the system, they are dependent on each other in defining the system behaviour.

One of the main objectives of the new component model is to have a clear separation of control and data in the software architecture. We have achieved the explicit modelling of control flow and data flow, but this does not mean that we have automatically achieved the separation of control and data in the architecture. ProCom is an example of a component model that has explicit modelling of control flow and data flow, however, it does not have the characteristic of separation of control and data as we discussed in Chapter 4. Therefore, we emphasise on modelling control, data and computation in the design of the new component model since we believe that this can lead to our goal in having the separation of control and data in the system architecture. On this basis, we dedicate components as entities that perform purely computation, control connectors as the entities that compose components (via the control port of components) and perform control flow routing, and data connectors and coordinators as the entities that perform data flow routing and data flow coordination between components.
Moreover, we make the connectors to be exogenous entities to ensure high decoupling of control flow, data flow and computation. By designing them in such a way, we achieve a better separation of control and data in the software architecture.

We could, of course, model a system that is both control-and-data-driven using data-driven elements only and not using any control connector constructs. This would lead to a mix of control and data in the system architecture. Consider again the fuel system example explained in Section 5.3.3 to clarify the idea of separating control and data in the system architecture. If we now use only data flow components, data connectors and data coordinators, then the constructed system architecture is a data-driven one, as illustrated in Figure 5.23.

Compared to Figure 5.21, here we replaced all hybrid components with data flow components and we used data coordinators between data connectors. As we have mentioned earlier in Section 5.3.1, data coordinators can be used to control the computations performed by data flow components. In Figure 5.23 we employ data switches to coordinate the data transfer according to the mode processed by component MC. The data switches will forward the data input to one of the three mode components based on the data that it gets from component MC that refers to the current mode of operation for the fuel system. This means that data pedal input and shaft input act as the control that is responsible for triggering the computation for all E/D components.

The components DDM and DRL in Figure 5.23 are working differently from the ones in Figure 5.21. Here, these components are working all the time as long as the data input is available, instead of working once as when the mode changes or when the system enters the stop mode. As long as component MC processes the data input to determine the current mode of operation, component DDM will keep on receiving the data input of the current mode and perform the computation.

For component DRL, the data input stop mode flag is used as an indicator to check if the current mode is the stop mode in order to generate the command to display the red light. Otherwise, component DRL will generate a dummy command since by definition, the component DRL operates at all time producing output as long as the data input (stop mode flag) is available. This means that the computation performed by component DRL is not needed when it generates the dummy command. A conditional structure that takes the value stop mode flag as the condition data is put inside the component DRL to determine whether the
Figure 5.23: A fuel system that is data-driven.
display red light command or the dummy command is generated as the output. In this case, the control is mixed with the computation inside component DRL. This is different with the control for the data flow components that represent the modes mentioned above since it is not about allowing the trigger to take place, but in the output generated by the component.

From the organisation of the data flow components and the data flows above, we can conclude that it is possible to use data coordinators and data connectors to employ the coordination in the form of control to the components. However, this leads to a scattered control in the system, and it is not possible to compose these control flows. The control and data are mixed in the architecture, and this makes the architecture model harder to read. In this example, there is no single identifiable entity that represents control for the system. Also, a component is not the entity that represents computation only but a mix of computation and control as demonstrated by component DRL.

One might argue that even with the use of a control connector hierarchy to represent a single control in the system, we might need to use the data coordinator to coordinate the data flow to control the arrival of data that plays a role in driving the execution. For example, an E/D component that is enabled is made pending to compute by blocking the data to arrive at its data port. This is an interesting point of discussion regarding the role of control connectors in defining the control for a system. The control connector hierarchy should be the central place where we can define the primary control flow structure in the system. Having a controller in a system is the interesting aspect about this component model where it provides the visual expression for the control flow structure. It is an explicit entity and can be manipulated and extended as the system grows. With regards to the role of data flow coordinators, of course, we use them to coordinate the data flow, especially at the point where data flow and control flow need to be synchronised. Therefore, the data flow coordination is not the primary control in the system as its role is only to coordinate the data flow.

5.5 Chapter Summary

In this chapter, we have defined the new component model elements: the various types of components and the various types of connectors. We explained in detail

---

3More explanation and example of this follows in Chapter 6
each of the elements of the model with examples. Subsequently, we illustrate the construction of an architecture model using our component model with the climate control system example. In the example, it is clear that our model separates control and data in the system architecture.

With the current chapter, we have completed the presentation for the new component model. Even though parts of the execution semantics have been presented by the elements of the model, the overall complete semantics has not yet been specified in the current chapter. In the next chapter, we present an execution semantics of the model to complement the design of our model.
Chapter 6

Realising the New Component Model

The previous chapter (Chapter 5) introduced the new component model and presented the basic building blocks that involved the modelling of control flow and data flow. In this chapter, we present the execution semantics to complete the definition for our new component model described in Chapter 5. We continue to elaborate on how we can build a system using our components and connectors, and explain how the system works in practice.

In Section 6.1, we explain how the component model elements work together to support the computation model that we described in Chapter 3. Consequently, this called for a way to manage the elements of data flow and control flow in one integrated model, and we explain this in Section 6.2. This is followed by the tool implementation that we have developed for the execution semantics in Section 6.3, that demonstrates how the component model is used to construct a running system. We demonstrate the construction of a running system in the tool with an example in Section 6.4.

6.1 Supporting the Computation Model

As explained in Chapter 3, the new component model aims to support the modelling of control flow and data flow. By doing this, we support the modelling of both data-driven and control-driven computation models. In this section, we present the abstract execution semantics of the new component model, by presenting the two kinds of systems that can be built using our component model. Even though
we have presented part of the semantics of the component model in the previous chapter, we ought to explain the semantics for the component model elements to be integrated in one model, forming a system architecture for system modelling. Indirectly, this allows us to relate to the computation model that we aim to support with the new component model.

Using our component model, we can build two kinds of systems: (i) systems that are purely data-driven, and (ii) systems that are both data-driven and control-driven. We describe them in turn in the following subsections. By explaining them in this order, we show how we go about modelling a spectrum of software systems with the elements of data flow and control flow.

6.1.1 Pure Data-Driven Systems

A pure data-driven system consists solely of data transformation operations, that continuously consume data, performing calculations and producing results. An example of this type of system is a signal-processing system that involves highly intensive data transformations operations. These require a continuous behaviour. That is, sampling sensors at regular time intervals, performing computations on their values, and producing output values.

In our approach, to build a system that is purely data-driven, we use pure data flow components, together with data connectors. To model the environment for the system, some of the components should be source components, and some of them should be sink components. Together, these model the data exchanged between the system and the environment, i.e. the input and output data of the system. Figure 6.1 shows an example of a purely data-driven system.

This example is a room temperature controller that continuously controls the temperature in a room, by means of two ventilation fans operating concurrently. The user can set a value for the desired temperature, and the controller will work the fans to achieve it. There are three inputs to the system: two temperature readings from two temperature sensors (S1 and S2) and a desired temperature value set by the user (U1). The system produces two outputs: one is to set the ventilation level for ventilation fan 1 (F1), and another one is to set the ventilation level for ventilation fan 2 (F2). S1, U1 and S2 are the source components for the system, and F1 and F2 (ventilation fan 2) are the sink components for the system. The Adjust Temperature components (AT1 and AT2) are used to perform the comparison of temperature set by the user and the real temperature readings from
the room (from S1 and S2). The output from the comparison is used to decide the level of ventilation required for controlling the temperature according to the user setting. This is the function of the Calculate Ventilation components (CV1 and CV2). CV1 and CV2 will output commands to adjust the ventilation level for F1 and F2, respectively. The components are connected by data connectors that are FIFO channels and non-destructive read channels. The data connectors are labelled with the names of the parameters involved in the data transfer.

For a purely data-driven system, the execution semantics is that of the ‘pipe and filter’ architectural style. The components are the filters, and the data connectors are the pipes that model the data flow. The data-driven semantics of data flow programming languages is implemented by a scheduler. The latter executes the data transformation function in every component, whenever the data is available at the input ports of the component. There can be at most one data value at an input or output data port. When all the input data ports for a component are filled with data values, the component can start executing the function by reading the data values from all the input ports. The input data ports will be empty again where the next available data item, queued in the pipe of the data connector, is transferred to fill up the input data ports. The data outputs are
produced and written to all the output ports of the component at the same time, when execution of the function inside the component is completed. Once the output data is written at the output ports, the component is ready to execute again — provided all the data are available at the input ports.

6.1.2 Data-and-control-driven system

The need to manage the processing of data requires the use of control to impose an execution order, by activating/deactivating data processing in the systems. This is the fundamental concept underlying a hybrid model that combines the data flow model and the von Neumann serial control flow model [88]. A region of data transformations within the running data flow system can be grouped together as a thread, to be executed by the control flow.

Similarly, in our component model, we use the control flow to manage the executions of the data transformations performed by the components. In particular, we use the component model elements that are control-driven (as well as data-driven elements) to build a system that is both data-driven and control-driven. For example, hybrid components are used so that we can control the data transformation operations via enabling or disabling, and triggering at the control ports of such components.

We now extend the example in Section 6.1.1 to introduce control flow into the system. Supposed we want to reduce the energy consumption of the system at night by turning off the ventilation fan \( F_1 \). This can be done by introducing a control flow to the system that is responsible for enabling/disabling component \( AT_1 \) and component \( CV_1 \). One more input is added to the system, namely the input from button \( T \) to turn the ventilation fan \( F_1 \) on or off (we call this the input toggle switch). We replace the data flow components \( AT_1 \) and \( CV_1 \) in the previous example with E/D components. We add a sequencer to do the enabling/disabling of \( AT_1 \) and \( CV_1 \) in sequential order. A guard connector is connected to the top of the sequencer so that control flow will only be forwarded to the sequencer whenever there is an input from button \( T \). The data port of the guard connector is connected to the button \( T \) (modelled as a data source) via a data connector size one FIFO. The FIFO data connector that carries value \( a \) (temperature 1) is replaced by a data connector FIFO of size one to ensure that \( AT_1 \) will always consume the latest data from the sensor \( S_1 \). The newly constructed system is shown in Figure 6.2.
6.1. SUPPORTING THE COMPUTATION MODEL

Figure 6.2: A data-and-control-driven system.
In comparison to the previous example, a part of the temperature controller system that is control-driven can only work when the components $AT_1$ and $CV_1$ are enabled. The enabling and disabling of these components is the result of actions from the toggle switch $T$, that causes the control flow to take action to enable or disable the two hybrid components ($AT_1$ and $CV_1$) in sequence.

For a system that is both data-driven and control-driven, the execution semantics involve both data flow and control flow. These two flows drive the execution of a system by executing different elements of the component model. In the following section, we take a closer look at how these two flows can work together and drive the execution of the system.

### 6.2 Managing Control Flow and Data Flow

The existence of both data flow and control flow in the architecture built using our model calls for a synchronisation mechanism. This is because we have to deal with the two flows running in parallel, and accessing the same information in the model. The examples presented in the previous sections gives an abstract view of the execution semantics for the component model. In this section, we present the detailed execution semantics, by describing the entities involved in realising the data-driven computation model and control-driven computation model. That is, the data flow scheduler for data-driven elements, and the control flow thread for control-driven elements. Then, we describe the problem of nondeterministic behaviour as a result of having two running flows operating in the same model. We solve this problem by having a synchronous execution model to effectively integrate both flows in the same model.

#### 6.2.1 Data Flow Scheduler and Control Flow Thread

The data-driven execution is implemented by a data flow scheduler that is responsible in performing the execution of the component model elements that are data-driven. A data flow scheduler performs the execution of: (i) the pure data flow components, by performing the data transformation function whenever all the data input ports are filled up with data values, (ii) the data connectors, by transferring the data values from the components’ output ports to connected channels and from channels to connected input ports if they are empty, and (iii) the E/D components, by performing the data transformation function, under
the condition that the state for an E/D component is enabled, and the data is available at the input data ports.

The control-driven execution is implemented by a control flow thread, that executes the control flow defined by the control structure of the system. Control flow is responsible for: (i) execution of control connectors according to the semantics, (ii) execution of TR components by performing a data transformation function, and (iii) switching the state of E/D components. A control flow thread is spawned once at system start-up and enters the top-most connector in the connector hierarchy. Depending on the connector type, the control flow either keeps iterating the top-most Loop, or it terminates after one traversal of the control structure if the top-most connector is not a Loop.

The main challenge that we face is to integrate these two flows for a running system. Both flows should work in a coordinated setting, to avoid the nondeterministic behaviour that we discuss in the following.

6.2.2 Nondeterministic Behaviour

Having two separate flows with two threads handling each flow results in a variation of behaviour for the same system model. For example, it affects the behaviour of E/D components that work under the direct influence of both the control flow thread and the data flow scheduler.

In order to explain this clearly, we refer to the example in Figure 6.2. For simplicity, we focus on component AT1, which is an E/D component that requires the involvement of both the data flow scheduler and the control flow thread. Component AT1 receives input data a and b from sensors S1 and U respectively. Suppose AT1 is enabled and the system receives the batch of input data (a and b) in which the T button is pressed to disable ventilation fan F1, there are two possible system behaviours:

1. Behaviour 1 – sensor readings from S1 and U are processed by AT1 if it is not disabled (by the control flow that visits AT1) prior to being scheduled for execution.

2. Behaviour 2 – the readings are unprocessed if AT1 is disabled and not scheduled for execution.

The nondeterministic behaviour is the result of having two independent flows operating over the same model elements. As we have shown in the example, the
order in which an E/D component’s state is changed, and the scheduling of its execution, is unspecified and varies non-deterministically every time a system is run. In general, this can result in unpredictable system behaviour.

In this example, the practical consequences of this asynchronicity may be negligible, but in general it is a serious problem that needs to be addressed, as it results in ambiguous execution semantics. Therefore, we employ one possible execution model that we called a synchronous execution model, which is elaborated on in the following subsection.

### 6.2.3 Synchronous Execution for Control Flow and Data Flow

We employ a synchronous execution model to solve the aforementioned problem. This involves coordinating both flows, in a way that control flow and data flow are coordinated in the execution cycle of a running system.

The execution of a system consists of a series of execution cycles. For data-driven execution, we implement a synchronous data flow execution for our model, where for each execution cycle, there are three execution steps involved for data flow execution: (i) the data are transferred from the output data ports to connected channels, and from channels to connected empty input ports, (ii) the scheduling for execution of data flow components that have all input data available at their ports, and (iii) the actual execution of the components, which happens conceptually concurrently, i.e., in arbitrary order. For control-driven execution, the control flow thread performs one traversal of the control flow structure in each execution cycle. Our proposed synchronous execution only caters for a system that contains one control connector hierarchy. Therefore, in order to use the proposed execution semantics, we need to employ a loop connector on top of the control connector hierarchy of our system. This is to ensure that the control connector hierarchy is visited in each execution cycle.

Figure [6.3] depicts the synchronous execution model for our model. In addition to the data transfer phase, each cycle comprises the scheduling of data-driven components (i.e. pure data flow components and E/D components), their parallel execution, and one control flow iteration. By ensuring that the execution scheduling and control thread iteration are sequenced, we avoid the aforementioned problem.
In fact, this execution model allows us to also execute data-driven systems, and therefore we use it for all systems built using our component model.

6.3 Execution Semantics Implementation

A prototype tool is implemented to demonstrate the execution semantics for our model. The tool is comprised of two parts: a system architecture editor and a system simulator. We briefly describe the tool in this section. In the next section, an example of a system built using the tool will be shown, to demonstrate the execution semantics of our model in practice.

6.3.1 Architecture Editor

The system architecture editor provides a system developer with a graphical environment to create the architecture of a system. A developer can instantiate individual components, compose them using data and control connectors, and configure their attributes. The editor has been developed using model-driven engineering techniques, i.e. it has been generated by the Eclipse Graphical Modelling Framework (GMF) from its meta-model.

The meta-model is shown in Figure 6.4. It captures the entities from the component model definition. The main entity is System, which contains all components, connectors, sources and sinks comprising the system. The Component, ControlConnector, DataConnectorOperator, and DataConnector entities are further subclassed by their particular semantic variants. For instance, Component has DataComponent and HybridComponent subclasses corresponding to pure data flow components and hybrid components, respectively. Figure 6.4 does not show

---

1This is a simplified version.
the attributes of meta-model classes. The attributes are used to fully specify the behaviour of meta-model entities, e.g. specification of a component’s function, or run-time semantics of a hybrid component (E/D or TR).

The system simulator is integrated with the modeller part of the tool (through the Eclipse platform and its plug-in mechanism). Its function is to simulate the execution of systems designed in the system modeller. It takes a system’s specification and simulator configuration\(^2\) and runs the system in accordance with the model’s execution semantics. A simulator configuration associates sources and sinks in the system specification with input and output files. It also contains some initialisation data, e.g. the initial states of E/D components, or initial contents of some channels. The simulator’s output is the files associated with the sinks, as well as a trace of system execution for debugging purposes.

### 6.3.2 The Run-Time Execution Semantics

In order to implement the simulator, we needed to create the run-time counterparts for meta-model entities, realising the semantics prescribed by the component model definition. For instance, a component, when executed, invokes a Java class specified in the system architecture that performs the component’s computation. Hybrid components have to have a state (enabled/disabled) and have to react to incoming control based on their semantics (E/D or TR) and current state. Data ports also have a state indicating whether they are full or empty. Likewise, the data connectors’ semantics had to be implemented (FIFOs, \(^{2}\)Currently, the simulator configuration is specified directly in the system’s specification in the architecture editor.)
non-destructive read channels), etc.

Another significant part of the simulator is a realisation of the execution semantics defined in Section 6.2.3. When the simulator is started, it spawns a thread for control flow and another thread for the data flow scheduler, which is running a data flow loop (Figure 6.5). In each iteration of the loop, the data is

```c
while (running) {
    channelsMoveData();
    componentsToRun = getReadyComponents();
    clockTick(); // sync with control th.
    executeComponents(componentsToRun);
    waitForControl(); // sync with control th.
}
```

Figure 6.5: Data flow loop.

transferred from the components’ output ports to connected channels, and from channels to connected input ports if they are empty (the `channelsMoveData()` method). Next, in accordance with the data flow semantics, the set of ready-to-execute components (pure data flow and enabled E/D components) is determined, and the components are executed in parallel (the `executeComponents()` method).

The loop is executed until the user terminates the simulator. The control thread iteratively goes through the connector hierarchy. The two threads (data flow scheduler and control thread) are running concurrently, and are synchronised according to Section 6.2.3. One period of synchronous execution is delimited by two consecutive calls to the `clockTick()` method. It is a signal for the control thread to make one iteration of the top-most Loop. The `waitForControl()` method waits for the control thread to finish that iteration.

### 6.4 Example

In this section, we show an example of a system architecture constructed using our component model. We demonstrate what the running system looks like in practice, using the architecture editor and the simulator. The example is a simple cruise control system that is used in a car.

---

3The current prototype only supports a system having a Loop as the top-most connector.
6.4.1 System Overview

A cruise control system is a system used to automatically control the speed of a cruising car. It is a function that helps the driver during a long drive, by automatically maintaining the speed of the car without the need to keep pressing the gas pedal. There are two modes of operation for a cruise control system; auto and manual. In auto mode, the system takes over the task of cruising the car by maintaining a fixed speed that is set by the driver. The auto cruising is disengaged when the driver presses the brake pedal, or presses the off button. In manual mode, the system calculates the speed according to the gas pedal input from the driver. Figure 6.6 shows the car dashboard which has the input button on the left, and the display for the cruise information on the right. The buttons on and off are meant to set the cruise control mode, and the display on the right shows the current mode (which will be highlighted) and the current speed of the car. When the driver turns the engine on, the cruise control is in manual mode. The driver can press the on button to switch on the auto cruise control only when the car goes beyond 70km/h. Assuming that the car goes beyond 70km/h, if the driver presses the on button to activate the auto cruise control, the system will read the current speed as the fixed speed that needs to be maintained. The car will continually cruise at this speed, until either the driver presses the off button or the driver presses the brake.

In summary, the system responds to four inputs which are set cruise, current speed, gas pedal and brake, and the system produces two outputs which are throttle value and display mode command. The details of these parameters are as follows:

- **set cruise** is a Boolean value, initially set to false for manual mode,
- **current speed** is an integer in the range of 0 to 130,
- **gas pedal** is an integer in the range of 1 to 150,
- **brake** is a Boolean, initially set to false, and
- **throttle value** is an integer in the range of 1 to 150.
- **display mode command** is a string, “Auto” or “Manual”, initially set to “Manual”.

The cruise control system operates in two modes that are depicted in Figure 6.7. The initial mode is the manual mode, where the system responds to the
inputs from gas pedal for the acceleration, and set cruise for activating the auto cruise control. The transition to auto mode happens when the set cruise value is true, which means that the driver presses the on button. While in auto mode, the cruise control system will maintain the current speed. The auto cruise control is disengaged and goes back to manual mode when the value of brake is true, or the set cruise value is false. Additionally, the system sends a display mode command to the system display.

6.4.2 The System Architecture

We construct a software design using our model for the cruise control system. The system takes four inputs: two Booleans (true when button and pedal are pressed, false otherwise) for set cruise and brake, and two integers representing the current speed and the gas pedal status. The system outputs two values, which are the throttle value and the display mode command. The system is sent inputs periodically, and produces outputs in response with the same frequency. Once started, the system enters the Manual mode.

Figure 6.8 shows the architecture of the cruise control system created using
our component model. On the left, there is a source \( S \) feeding input data into the system. The source’s output ports produce data at the same time periodically. On the right, there are two sinks, \( THR \) and \( DIS \), that receive the \textit{throttle value} and \textit{display mode command} respectively. In the middle are two E/D components (\textit{Auto} and \textit{Manual}), and a TR component (\textit{Display Mode}). Components \textit{Auto} and \textit{Manual} correspond to the modes of the cruise control system, and component \textit{Display Mode} represents the display function for the modes. A hierarchy of control connectors is built with the behaviour that enable the mode switching and triggering of the display mode component every time the system changes to a new mode. Additionally, there is a data flow component \( MC \). This data flow component plays a role in changing the current mode of the system.

![Figure 6.8: Cruise control system architecture.](image)

At any time, exactly one of the E/D components \textit{Auto} and \textit{Manual} is active, and it processes incoming data (current speed in case of \textit{Auto} and gas pedal in case of \textit{Manual}) and computes output values for the throttle. Initially, the
system is in manual mode which requires the component **Manual** to be activated (or enabled). Later in the system execution, when the driver changes mode by pressing the on button, control connectors deactivate the component **Manual** performing the manual mode computation, and activate the component **Auto** corresponding to the auto mode. Information about the set cruise button input (a Boolean value) and the brake are first processed by component **MC**, a data flow component that is based on the button input, the brake and the current mode determines whether the system should switch to a new mode. If so, it sends the new mode to the selector connector. If the system should remain in the same state, the selector connector is not sent any data. This filtering functionality is performed by a data guard (the GD entity in the figure). When the control thread comes to the selector and finds the data ready on its port, it is forwarded to one of the selector’s children (the mode is an integer in \{0, 1\} and is used as index in an array of the children) – two sequencer connectors. Upon arrival of control, they redirect it in sequence to two of the aforementioned E/D components, deactivating one and activating the other, respectively. The data flow component **MC** also produces parameters **auto mode** and **manual mode**, which are Boolean values used to forward the input data **current speed** and **gas pedal** through the data guards. Only one of them will have the value true at any time during the system execution. These values are set to true by component **MC** only when the data input should be forwarded to the respective E/D component. Upon visiting these two E/D components, the control flow will return back to the sequencer connector, the selector connector and another sequencer connector on top of the selector connector. At this sequencer connector the control flow will be directed to the next parameter that is connected to the TR component **Display Mode**. The TR component will perform its computation once which is to set the command to display the current mode. It is connected to the sink component **DIS** using the non-destructive read size one buffer data connector that retains the data of string type. This data contains the message to be printed on the dashboard display to indicate the current mode. We use the non-destructive read size one buffer data connector here since we want to retain the current value for the mode to be displayed. This is because the TR component **Display Mode** will only perform its computation when there is a change in mode since the control flow routing might be pending at the selector control connector waiting for the current mode data that will only be forwarded when the mode change data value
is true.

The example architecture in Figure 6.8 uses three types of data connectors: unbounded FIFO, size one FIFO, and non-destructive read size one buffer. Unbounded FIFO connectors are used to realise connections between two data ports where data cannot be lost and can accumulate in the connector’s queue. Size one FIFO connectors can be used in situations where data cannot accumulate or where they can be rewritten by a newly written values. Non-destructive read size one buffers in this example effectively simulate the state for component MC.

Here we modelled the system architecture for the cruise control system based on the execution semantics that we have defined. Thus, the resulting architecture used different type of data connectors compared to the ones in fuel system example explained in Section 5.3.3. In the fuel system example, we used size one FIFO data connectors to connect the data input to the three components that represent the three modes for fuel system operation to ensure that these components always consume the latest data input. This is different for the cruise control system in Figure 6.8 where we used FIFO data connectors to connect the data input source that carries the current speed and the gas pedal to components Auto and Manual. Here, it is not possible to use size one FIFO data connectors based on our execution semantics. If we use the size one FIFO data connectors, we will lose the data input to components Auto and Manual, since the data input will only be consumed after two cycles of execution\footnote{one cycle of execution is illustrated by Figure 6.3}. The first cycle involves processing the data input set cruise and brake by the MC component and the second cycle involves a control flow iteration that processes the decision data (produced by component MC) to perform the disabling and enabling of the components (Auto or Manual) for the current mode. By using FIFO data connectors, we ensure that no data input will be lost while waiting for the two cycles of execution to complete. However, this has caused components Auto and Manual to consume stale data since the input data is buffered for two cycles before the system can consume the data input based on the current mode and produce the results.

### 6.4.3 System Simulation

Here, we briefly illustrate how our prototype tool can be used for simulating system execution. As mentioned in Section 6.3.1 we need a system architecture
and simulator configuration to run the simulator. Using the architecture editor, we construct the cruise control system in Figure 6.8.

Figure 6.9 shows the system architecture in the architecture editor of our tool. In order to run the system simulator, we need to configure the initial values. This includes initialising the non-destructive read size one buffer data connector and the states for the E/D components. For the cruise control system, the initial operating mode is manual, and this information is initialised for the data connector connecting the data ports \textit{mode} and \textit{curMode} of component \textit{MC}, to represent the current mode that is fed back for the next computation. Also, we initialise the data connector connecting the data ports \textit{display mode command} since the system will display the current mode of operation at all time by sending this data to the sink component. In realising this initial data, the following is the configuration entered in the tool to run the system:

<table>
<thead>
<tr>
<th>Configuration item</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data connector \textit{mode} → \textit{curMode}</td>
<td>1</td>
</tr>
<tr>
<td>Data connector \textit{disModeCmd} → \textit{disModeCmd}</td>
<td>Manual</td>
</tr>
<tr>
<td>Manual.active</td>
<td>true</td>
</tr>
<tr>
<td>Auto.active</td>
<td>false</td>
</tr>
</tbody>
</table>

We have omitted the specification of input and output files for the source and sinks. The configuration describes the initial state of the system: the initial contents of a data connector, and the state of the E/D component responsible for the initial mode. The value stored in the data connector corresponds to the current initial mode, which in this case is the value 1 that represents the manual mode. Furthermore, we need to create an input file which will be read by the system’s source. Such a file corresponds to a possible scenario with particular sensor measurements and a specific sequence of buttons pressed by the user. In this case, the inputs are the button used to set the auto cruise control, the brake press, and the sensor reading that gives the current speed and the gas pedal level.

Table 6.1 shows the expected behaviour of our cruise control system with this specific data input. Each row in the Input column corresponds to one batch of inputs produced by the source at one time. The Mode column contains the current mode. The values in the Output column show expected values to be computed by the system, for the inputs presented on the same row in the table.

When we execute the simulator, it shows the results in a window for immediate
Figure 6.9: Cruise control system in the architecture editor.
6.5 Discussion

Defining the execution semantics in detail is the essential first step in bringing out the practical aspects of the component model. However, what we presented here in this chapter is merely the first step, which we consider experimental, as we believe that the work of defining the execution semantics should be more involved.

In Section 6.1, we presented the abstract execution semantics of pure data-driven systems and data-and-control driven systems. The idea is to present an

Table 6.1: Expected behaviour of the cruise control system for our test scenario.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>setCruise</td>
<td>brake</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

inspection (see Figure 6.10). Comparing Figure 6.10 and Table 6.1, we see that the system has produced the expected results, and therefore passed the test.
overview of the execution of the systems in association with the computation model that we described in Chapter 3. However, during the implementation process, in realising the abstract execution semantics, we subsume the difficulty in putting both control flow and data flow together to work for any kind of system.

From the example of the room temperature controller, modelled as a pure data-driven system in Figure 6.1, it is straightforward to adopt an execution semantics that can satisfy the requirements for a data-driven computation model. We discuss two ways of doing it here. First, we can adopt the synchronous execution model, which is the solution that we implemented for the component model described in this chapter. In a synchronous execution model, computations are performed concurrently in the same step of a cycle. This means that all data transformations that have the data input ready to be transformed are executed simultaneously. We explain this with the example in Figure 6.1. If we implement the model with the synchronous execution for data flow, the components AT1, CV1, AT2 and CV2 are executed simultaneously, if all of the components have the data ready at the input data ports. This means that for every execution cycle, all data transformations (that are due to compute) will produce output at the same time, because they are bound to the same clock cycle. They are synchronised in the cycle that is fixed. This is different from the second approach, where we can implement the data flow execution using the asynchronous data flow. We use the same example in Figure 6.1 to explain this. In asynchronous execution of data flow, the computations are not bound to the same clock cycle. All computations are truly independent, in the sense that they have their own thread of control. This means that they can perform their computation at any time when the data is available. In the example, the components AT1 and CV1 may work at a different rate in transforming the data input and producing the data output.

Both synchronous and asynchronous models can be used to implement a simple data-driven system such as the example in Figure 6.1. The same set of outputs will be produced given the same set of inputs for both execution models, if only we use the unbounded FIFO data connector to transport data between components and we avoid many-to-one connections. The input will go through the same data transformations and the data transformations will produce the output accordingly. In Figure 6.1, the input that goes into AT1 will eventually go into CV1, and the output will be written to the sink. By right, the execution of
these data transformations happens in sequential order, because the data channel that exists in between these two data transformations ensures that the data output from component AT1 becomes the data input to component CV1. Here, the only difference between synchronous and asynchronous execution is that in synchronous execution, we know when to expect the output to be written to the sink, because we know the cycle that it takes to produce the final output. However, it is not always the case that a system architecture consists of linear data passing such as in Figure 6.1 and therefore, in general, it is far too simplistic to assume that both synchronous and asynchronous models will produce the same set of outputs given the same set of inputs.

To manage both control flow and data flow in the same model, we employed synchronous execution that results in a deterministic execution behaviour. We integrated the control flow execution with the data flow execution, by including one iteration of control flow in every cycle of execution of the data flow. This approach works for systems that are designed to operate in mode. This means that for each mode, there should be an E/D component that is designed to perform the computation required for the operation of the mode. The example in Section 6.4 illustrates this situation where there is one component to represent each mode (Auto and Manual) and perform the function of transforming the data inputs for the system, to produce the data output for each mode. The system is designed in such a way that for every control iteration, there is only one mode of operation involved. Whenever there is a mode change, the control flow will deactivate the previous mode, and activate the current one by disabling the component that is working in the previous mode, and enabling the component that should be working in the current mode. However, for the temperature controller system depicted in Figure 6.2 the synchronous execution implemented would not work since component CV1 needs the current data produced by component AT1. This dependency required AT1 and CV1 to perform their computations in the same cycle of execution, in which the control flow iteration takes place to enabled or disabled both of the components. With our current implementation, this is not possible since AT1 and CV1 can be disabled while the data adjusted temperature is travelling through the FIFO data connector to component CV1. In this case, when both are enabled again, component CV1 will consume the old data and not the current data produced by component AT1.

The synchronous execution model that we employed here is not the standard
CHAPTER 6. REALISING THE NEW COMPONENT MODEL

synchronous execution model, with the hypothesis that the execution of a system which takes a set of inputs to produce a set of outputs is infinitely fast [18]. It is worth mentioning that our synchronous execution for the data flow modelling does not follow the synchronous data flow as described in the literature [79]. The synchronous execution model that we implemented only aimed for a systematic coordination of the control flow and data flow execution.

Also, the execution semantics introduced for the model requires some effort from the user of the component model, namely to understand clearly the overall semantics, in order to design a system that can work with the implementation.

As we mentioned earlier, the execution semantics presented in this chapter is an effort to realise the component model in practice. Although the work discussed in this chapter is experimental rather than theoretical, we have established a working component model used to construct a simple cruise control system.

We explain the semantics of the component model in this chapter only informally. We believe that the work requires more elaboration on the theoretical part rather than the practical part. The solution for the implementation part of the component model needs to work in full generality, enabling use of the tool to build any kind of system. We consider this an important direction for future work.

6.6 Chapter Summary

In this chapter, we have defined an execution semantics for the new component model. We show how the elements of our component model are used to construct a data-driven system and a data-and-control-driven system. From this, it is clear that our model supports the two main computation models, namely those that are data-driven respectively control-driven.

We explained why it is challenging to manage the two flows in the execution model. This is done by describing how the data flow scheduler works, and how the control flow thread can be implemented. Having these two entities running without synchronisation causes nondeterministic behaviour of the system. We employed a synchronous execution model as the solution for synchronising the control flow and data flow — so that both flows can be effectively integrated into the system. Finally, we pointed out the issues regarding the implementation in the discussion. The discussion has highlighted the potential for improvement of
the execution semantics in the future.
Chapter 7

Using Our Component Model for the Embedded Systems Domain

In the preceding chapters, we have laid the foundation for the work of this thesis, which is both theoretical and practical in nature. We emphasised the importance of modelling control, data and computation, and we introduced a new component model that serves this purpose. The explicit modelling of control and data is found to be the main characteristic in designing software for the domain of embedded systems [98, 62, 13]. Viewed in this light, we take the important step of evaluating the new component model by demonstrating its usage in the domain of embedded systems. In this chapter, using an embedded system case study, we will show how we construct a system model using our component model. From this model, we demonstrate the advantages of our approach over other component models, particularly ProCom and SCADE, that are intended for the embedded systems domain. This is done by means of constructing the same model for ProCom and SCADE, and performing a close comparison with our model.

This chapter is organised as follows. An overview of the embedded systems domain is given in Section 7.1. This is followed by a detailed description of the Climate Control System case study in Section 7.2, which will be used throughout the chapter for evaluation purposes. Then, we demonstrate in detail how we construct the climate control system using our component model in Section 7.3. We compare the model with the ones constructed using ProCom and SCADE in Section 7.3.1 and Section 7.3.2 respectively.
7.1 The Embedded Systems Domain

An embedded system is a software system that is designed to work continuously with the environment, and is often embedded into enclosing products or devices. These range from simple devices such as TV remote controls, to safety-critical software such as missile-guidance systems. Often, these devices require a control system that consists of a plant, which is the functional part of the system, and a controller (Figure 7.1). The controller continuously monitors and controls the plant, based on the inputs that it gets from the environment.

In the following, we briefly describe the essential characteristics of embedded systems [26]:

**reactive** the system operates continually as follows: it accepts input from its environment, performs computations according to the input, and produces output,

**real-time** often, embedded systems are expected to react and produce output in a time period that is specified explicitly, and

**hybrid** embedded systems are frequently characterised as systems that mix continuous and discrete behaviours.

![Control system model of an embedded system.](image)

The hybrid characteristic for a software model, containing the modelling of
both continuous and discrete behaviour, is particularly useful for the formal anal-
alysis and design of embedded systems [39, 1, 9, 61]. The formal analysis is fo-
cusing on verifying whether the system satisfies the required specification, while
the formal design is concerned with meeting the specification, by performing the
process of synthesising the controllers.

In this chapter, we focus on the hybrid characteristic of embedded systems.
An embedded system also known as a hybrid system contains both continuous
and discrete behaviour [5]. The mixture of continuous and discrete behaviour
involves both data flow and control flow. The modelling of these two elements is
the main focus of the component model that is proposed in this thesis. The part
that is continuous exhibits a regularity in its behaviour, which (as mentioned)
involves consuming input, performing computations and producing results. The
continuous behaviour is that behaviour of the data processing functions that is
modelled as the plant shown in Figure 7.1. The controller in Figure 7.1 behaves
discretely, and it causes a change in the system state as a response to events. Both
parts determine the system behaviour, but their presence should be separated and
made explicit in the model, as depicted in Figure 7.1.

Our component model consists of modelling elements that are well-suited for
hybrid characteristics of embedded systems. Heterogeneity is supported by our
component model through the modelling of data flow and control flow, integrated
but separated. In this chapter, we evaluate our component model against these
criteria for embedded systems. Also, we want to draw attention to the subject of
software reuse in the context of a domain. Embedded systems are considered a
domain of software systems. In building software for a domain, our control con-
nectors have the ability to be designed in the context of a domain that promotes
software reuse. In particular, our control connectors can be composed together
to form a complicated control flow structure, that is domain-specific. We are able
to construct a composite control connector to represent the control flow structure
for a series of systems in a domain, which in turn can be deposited in a repository
for future reuse.

In the remainder of this chapter, based on the characteristics discussed in the
preceding paragraph, we show how our component model is used in the domain
of embedded systems, and which advantages our model provides compared to the
others.
CHAPTER 7. USING OUR C.M. FOR THE EMBEDDED SYS. DOM.

7.2 Climate Control System

In this section, we explain in detail the climate control system case study that we adapted from [62]. A climate control system is a sophisticated form of air-conditioner used in a car, which allows accurate control of a car’s interior temperature. The driver can control the climate of the car using the interface shown in Figure 7.2.

![Figure 7.2: Inputs: buttons on the car’s dashboard.](image)

The system receives four inputs of Boolean type, corresponding to the buttons CLIMATE, L (Left), R (Right) and OK. Using the interface in Figure 7.2, the user can control the system to produce output as depicted in Figure 7.3.

![Figure 7.3: Outputs: display on the car’s dashboard.](image)

This is a display on the dashboard, showing the current operating information of the climate control system. The details of the outputs are as follows.

- **ClimateMode** is the current climate mode selected for the car. There are two climate modes, Auto and Manual. The initial selection is Auto. The user can select the climate mode using the L and R buttons to navigate to the desired selection, and press the OK button to confirm the selection.

- **Temperature** is the current temperature set in the car. The user can tune the temperature using the L and R buttons to specify the temperature in integer form, in the range \{17, \ldots, 27\}. The initial value is 19.
VentilationLevel is the speed level of the fan to work for ventilating the air in the car. The level is expressed as an integer in the range \( \{0, 100\} \), the initial value being 0. The user can control the ventilation level by using the L and R buttons.

VentilationMode displays the current ventilation mode for the car. Again, the user can navigate to the desired selection using the L and R input buttons. The selected ventilation mode will be highlighted. Possible selections are CAR, FACE, FEET, DEFROST and CIRCULATION. The initial value is CAR.

There are three modes to which the user can set the device for controlling the climate in the car. These three are Auto, Adjust and Manual. Figure 7.4 shows these modes, corresponding to the states of the climate control system device.

![Diagram of climate control system modes](image)

**Figure 7.4**: The states for the climate control system.

**The Auto State** This mode allows user to set the temperature. By using the L button, the user can decrease the temperature, by steps of 1, down to 17. To increase the value for the temperature, the user can use the R button for increasing, by steps of one, up to 27.

**The Adjust State** In this mode, the user can navigate with the L and R buttons through the ventilation and climate modes in the following order: CAR, FACE, FEET, DEFROST, CIRCULATION, Auto and Manual. To confirm the selection, the user presses the OK button. Pressing this button when the Auto mode is selected will make the system switch to the Auto state. The same applies to the Manual mode, where confirming the selection by pressing the OK button will make the system switch to the Manual state.
CHAPTER 7. USING OUR C.M. FOR THE EMBEDDED SYS. DOM.

The Manual State  This mode allows the user to set the ventilation level. Using the L button, the user can decrease the ventilation level by steps of 1 down to 0. To increase the ventilation level, the user can use the right button that takes the ventilation level up to at most 100.

For the purpose of demonstrating the feature of separating control and data by our component model, we consider a climate control system in a van that allow the passenger seating at the back of the van to control the climate. This system consists of two sets of input/output device (i.e. the input buttons and output display depicted in Figure 7.2 and Figure 7.3 respectively), but with a simplified version for the passenger input/output device. These input/output devices receive configurations (for accurate display of temperature) and turn on/off command from the system, as well as sending the input from the user to the system. There are two sensors in the van (at the front and at the back) to measure the current temperature in the van. The passenger is allowed to control (using the L and R buttons) the ventilation mode and the climate mode, and this can only be done when the climate control system is in the adjust mode. This means that the system needs to activate the passenger input/output device (input buttons and mini output display) only when the climate control system of the van is in the Adjust mode, and deactivate it when the climate control system is in the Auto or Manual modes. The activation is done by reading the current climate temperature from the first sensor located close to the driver’s seat, calculating the configuration for the passenger input/output device and setting the command to turn on the passenger input/output device. The deactivation is done by reading the current climate temperature from the second sensor located close to the passenger seat (at the back end of the van) calculating the configuration for the driver input/output device and setting the command to turn off the passenger input/output device. The details of additional inputs and outputs are as follows.

\textbf{ts1 and ts2} are inputs to the system representing the current temperature reading inside the van, where \textit{TS1} is the reading from the first sensor located near the driver’s seat and \textit{TS2} is the reading from the second sensor located near the passenger’s seat at the back of the van. Both are of Integer type, in the range \{5, \ldots, 40\}.

\textbf{pdc, ddc and cmd} are the outputs produced by the system to configure the input/output devices. The passenger input/output device receives two data
input which are \texttt{pdc} and \texttt{cmd}, that configure and turn on the device respectively. Similarly, the driver input/output device receives one data input which is \texttt{ddc} that reconfigure the device after the passenger input/output device is turn off. The \texttt{pdc} and \texttt{ddc} are of file type, that contain the configuration for the devices. The \texttt{cmd} is of Boolean type, with value \texttt{true} means turn on and value \texttt{false} means turn off.

7.3 System Modelling with Our Model

This climate control system is a typical example of an embedded system with running modes. The behaviour of the device’s buttons is dependent on the mode, which is being changed at runtime. In other words, a change of mode controls the system’s behaviour. When the system is in a particular mode, it is driven by data — namely incoming \texttt{L} and \texttt{R} button clicks. Therefore, the system possesses characteristics of both control- and data-driven architectures, which can be separated by means of our model.

Figure 7.5 shows the systems architecture of the climate control system constructed using our component model. We use three hybrid E/D components and two TR components. The E/D components are used to model the three modes of operation: \texttt{Auto}, \texttt{Adjust}, and \texttt{Manual}. The TR components \texttt{APD} and \texttt{DPD} are used to model the computations required to perform the activation and deactivation of the passenger input/output device respectively. Since there is only one active mode at a time, there must be only one E/D component being activated at a time. The TR components \texttt{APD} and \texttt{DPD} are triggered whenever the system enters the adjust mode and leaves the adjust mode respectively. On the left, there is a source component that feeds the input data to the system. \texttt{Climate}, \texttt{Ok}, \texttt{Left} and \texttt{Right} are the user inputs that come from either the driver or the passenger, and are not being distinguished in the model\footnote{The handling of input data is done in the source component which is an entity external to the system}. In addition, there are inputs from the two temperature sensors in the van, \texttt{TS1} and \texttt{TS2}. The inputs \texttt{Climate} and \texttt{Ok} are fed to a data flow component \texttt{MC} that is responsible for monitoring the changes in mode. The \texttt{MC} component monitors these changes by comparing the \texttt{current mode} input with the input from the \texttt{Climate} and \texttt{OK} buttons. The \texttt{MC}
Figure 7.5: The climate control system with our model.
component produces three outputs, which are mode change, mode action and current mode. The current mode (of integer type) is fed back to the MC component to monitor the state changes. The mode change produced by the MC component is a data item of Boolean type, which carries a true value when there is a change in the mode. This Boolean value is fed to the data guard (GD1) as a data condition that determines the forwarding of mode action value, of integer type in \{1, 2, 3, 4\}, to the selector connector. The mode action, as the output value computed by the MC component, corresponds to the four possible actions for the state transitions shown in Figure 7.5. These four actions each involve deactivating (disabling) and activating (enabling) a pair of components as follows:

1. from the Adjust state to the Auto state: this requires deactivating the Adjust component and activating the Auto component,
2. from the Auto state to the Adjust state,
3. from the Adjust state to the Manual state, and
4. from the Manual state to Adjust state.

We use a sequencer connector to perform the actions of deactivating and activating. Apart from this, for actions number 2 and number 4, we need to perform the triggering of component APD while for actions number 1 and number 3, we need to perform the triggering of component DPD. The triggering of component APD is to perform the activation of the passenger input/output device after enabling the adjust mode and the triggering of component DPD is to perform the deactivation of passenger input/output device after disabling the adjust mode. There are four sequencers connected to the respective components, where the connections are modelled according to the actions (deactivating, activating and triggering) required. In other words, the first leg of the sequencer (labelled as x) is connected to the component that requires deactivation, the second leg (labelled as y) is connected to the component that requires activation, and the third leg (labelled as z) is connected to the component that requires triggering. These sequencers are in turn connected to the selector connector, which will forward the control flow according to the data that it receives from the MC component. The selector connector is connected to the top level loop connector, which feeds the infinite control flow to the system. At the start of the system, control is fed to the loop connector to initiate a control flow.
CHAPTER 7. USING OUR C.M. FOR THE EMBEDDED SYS. DOM.

The three E/D components receive input data values that denote L and R button presses. The data source S feeds the system with data values for L and R button presses to the data switches SW1 and SW2, respectively. A button press is represented by the integer 1, while no button press is represented by the integer 0. The data switches receive the data current mode from the MC component, used to determine which component is active, and requires the L and R button press input data value. The data switch ensures that only one of the E/D components will receive the L and R button press from the data source. The TR components APD and DPD receive input data values TS1 and TS2 from the data source S respectively. These components are triggered (i.e. perform computations) only during control flow iterations that reach the sequencers when there is a change in the mode. The data output produced by TR component APD are used to configure and to turn on the passenger input/output device modelled as a sink component PD. The driver input/output device (modelled as a sink component DD) receives data produced by TR component DPD.

The system produces output based on the mode selected. In the Auto mode, the user can adjust the temperature using the L and R buttons, and this action is handled by the component Auto. The temperature value is written to the data sink component TMP. The climate mode and ventilation mode are produced in the Adjust mode, where the output values are written to the data sink components CM and VM, respectively. For the Manual mode, output values for ventilation are written to the data sink component V.

The system architecture in Figure 7.5 clearly shows that our model separates the modelling of control flow and data flow. The control part of the system is represented by the control connector hierarchy, which is explicit and separated from the data flow part of the system, represented by the data flow components and E/D components. The two parts are then connected and synchronised together by means of data connectors and data coordinators (data switches and guards). Our model has a variety of modelling elements that can express the modelling of control flow and data flow. Moreover, the control structure defined by the hierarchy of control connectors exists as an entity for software reuse. The behaviour captured by the composite control connector is specific to the climate control system, where it can be reused over and over again to build the climate control system, if this composite control connector is stored in a repository.

It is now possible to demonstrate the advantages of our approach over other
component models intended for the embedded systems domain. In the following subsections, using the same example, we embark on making a close comparison with two component models, ProCom and SCADE, that represent the current state-of-the-art in modelling software for embedded systems.

### 7.3.1 Comparison with ProCom

Figure 7.6 shows the system architecture of our climate control system constructed using the ProCom component model. The climate control system is designed as a subsystem. This subsystem consists of ProSave components that are connected to perform the operation of controlling the climate device in a car. Similar to our approach, each mode is modelled with a component. There are three components: Auto, Adjust and Manual, representing the three modes of operation for the climate control system. In order to model the mode changes, another component is used to handle the inputs to the system. The component Handling Input receives data input from the message ports, and performs the operations to handle the mode changes. The component Handling Input produces a control trigger output and the current mode, used as decision data to route the control flow. The Selection connector forwards the control flow to the respective component based on the current mode decision data. The left and right data values are forwarded to all of the components by the data fork connector. The output ports of all the three components that model the mode are connected to the output message port. Since only one of the modes will be active at a time, not all of the output message ports will have output data. The source of control for the subsystem is a clock that works at a rate of 10Hz. For each clock cycle, the climate control subsystem will produce output based on the mode in which the system is operating.

In ProCom, since all of the components are control-driven, data flow is dependent on control flow. For each path of data flow that requires going through computation operations performed by components, the control flow has to be routed to reach the control trigger port of the components. In this case study, the control flow is routed first to component Handling Input, since the system needs to perform the evaluation of inputs from the climate and ok buttons to determine the operating mode for the system. The component Handling Input is responsible for performing the evaluation of input data against the current state of the system. The output data produced at the output data port, together
Figure 7.6: The climate control system in ProCom.
with the control trigger, will flow to the subsequent components. Control flow that goes through the Selection will be filtered, so that only one of the three components (Auto, Adjust and Manual) will be triggered.

The component Handling Input is an example of a component that only performs a pure data transformation, which involves transforming input data to output data. In ProCom, since data flow is not separated from control flow, the modelling of component Handling Input has to be accompanied by the control flow. This is different from our approach, where we can model a pure data transformation that involves data flow using our data flow component. By doing this, we clearly separate the elements of control from the elements of data in modelling the system architecture.

In ProCom, although we can connect control connectors together, it is not possible to have a single entity that represents a control flow structure for the whole system as a result of composing control flow. This is shown by Figure 7.6 where there are two control connectors (Selection and Control OR) that are not connected to each other. Furthermore, sequencing behaviour is modelled by connecting components one after the other using connections that connect trigger ports. In this example, components APD and DPD which need to be triggered after the triggering of one of the mode components are connected by the trigger port connections.

The control flow between component Auto and component DPD is not connected to the control flow modelled on the left side of component Auto. The control flow is not composable — unlike in our model, this involves connecting a sequencer at the appropriate branch of the control structure to compose the new control flow. The new connector that captures the new control flow is added to the existing hierarchy of control connectors. The hierarchy of control connectors is considered as one composite connector, which can be stored in a repository as a reusable entity. This feature of our model provides an opportunity for software reuse, which is an advantage.

7.3.2 Comparison with SCADE

We recapitulate the SCADE example explained briefly by Figure 4.12 in Section 4.2.2. The same figure is used here for comparison with our model. Figure 7.7 shows the climate control system (without our adaptation) constructed using SCADE, consisting of Lustre blocks and the Safe State Machine (SSM). The
system model takes four inputs of Boolean type, corresponding to the physical input buttons (ok, climate, left and right) of the system, and produces four outputs (ClimateMode, Temperature, VentilationMode and Ventilation) that are sent to the respective device or actuator, to realise the climate control. Since the inputs Ok and Climate determine the change for the system mode, they are fed to the SSM that contains the behaviour modelling of the system. These two data values are evaluated against the current state (i.e. mode), and the result of the evaluation will be the output for the SSM block. There are two outputs from the SSM: (i) a data value to indicate the current Climate Mode, and (ii) a control signal in the form of data that is used to activate one of the Lustre blocks, which should operate in the current mode. The Selector operator S is used to route the Boolean value to the respective Lustre block for activation. The Join operator J is used to forward either one of the ClimateMode values produced by the SSM, or the one that is produced by the Adjust Lustre block. The decision is based on the current mode value received from the SSM.

From Figure 7.7, we can see that all the lines that connect the modelling entities in SCADE represent flows of data values. These modelling entities, with the graphical block-diagram notation, perform their operation only when the data is available, making the SCADE model a pure data-driven model. For this reason, control is expressed in terms of a state machine, modelled inside an SSM.
block. From the external view, the SSM block behaves similarly to the rest of the Lustre block, in the sense that it takes data as inputs and produces data as outputs. This is different from our model, where we are better able to express the control flow in the system. Control connectors define the explicit control flow structure in the model, as compared to in SCADE, where control is defined as state machines. Also, in our model, we differentiate between the modelling entities that are related to control flow and data flow. For example, in this case study, using our model we are able to model computations that involve both data flow and control flow by means of the hybrid E/D components. This situation is different in SCADE, where the modelling of computations that involve both data and control is done using Lustre blocks. These are the same modelling entities used to model pure data transformations.

7.3.3 Comparison with PECOS

In this section, we describe another piece of related work, which is the PECOS component model [90]. The reason for this is that because PECOS is a component model targeted to the embedded systems domain [49], and it uses a data-centric approach to compose components [109].

The PECOS component model is designed to support specific requirements in building software for field devices, which is part of the embedded systems domain. The components in PECOS are fine-grained and specified to encapsulate a computation, and export a data interface. The data interfaces of components are the only way that PECOS components are able to communicate with the environment. The components are either active, passive, or event, used to perform different behaviours and functions. Figure 7.8 shows an example of a PECOS active component ModBus with one data port setFrequency.

![ModBus setFrequency](image)

Figure 7.8: PECOS component.

An active component contains its own thread of control that defines its process. This is similar for event components, except that the execution of the
behaviour is triggered by an event. A passive component, as the name suggests, does not contain a control thread, and is executed exclusively by the process of the composite component that it is part of. Composite components in PECOS are always active.

There is only one type of port for PECOS components, namely a data port which is read or written by a component, and which can be shared with other components. All components have an execute method (denoted as `exec()`) which acts as an entry point to the component. For components that share data ports, the `sync()` procedure is used for synchronisation of reading and writing of the shared data. Each active and event component schedules its own execution procedure, and that of any subcomponents, in a single thread. A composite component schedules the execution of its own `exec` procedure, the `exec` procedures of its passive subcomponents, and the synchronisation procedures for its active and event subcomponents. Component ports are connected to each other using `connectors`. Connectors in PECOS define the shared data between ports, which the components use to communicate.

Here, we use the example in [90] to illustrate how software is modelled using the PECOS approach. Figure 7.9 shows three connected PECOS components: (i) an active component `ModBus`, (ii) a passive component `ProcessApplication`, and (iii) an event component `FQD`.

![PECOS Components Diagram](image)

Figure 7.9: Composing PECOS components to set a valve position.

The three components `ModBus`, `FQD` and `ProcessApplication` are composed together to form a composite component. This composition requires specifying a control flow that is written in the scheduler. The scheduler is the control mechanism used to schedule the reading and writing of data at the data ports of components. Data flows between components depend on the scheduling by the
7.4. ANÁLISIS

La work in this thesis that focused on control flow and data flow modelling is in line with the hybrid characteristics in embedded system modelling. These characteristics are presented clearly by the structure of the control system model depicted in Figure 7.1. The structure, that basically consists of a controller representing the control part of a system and a plant representing the data part of a system, highlights once again the importance of separating control and data in software architecture.

Consider the climate control system case study modelled using our component model, shown in Figure 7.5. If we represent the control part and data part as two black boxes that hide their contents, we would have a structure similar to the
one in Figure 7.1. For convenience of illustration, here we again show the control system model structure in Figure 7.11(a).

![Control system model](image)

(a) Control system model.  

![Control and data boxes](image)

(b) Control (controller) and data (plant) as black boxes for our model.

Figure 7.11: Comparing the structure for embedded systems.

We compare the control system model with our climate control system in Figure 7.11(b) hidden in two boxes. These two boxes represent the control and data parts, also known as the controller and the plant, respectively. The lines that connect the two boxes represent the control flow and data flow between them. We distinguish between lines that represent data flow, and the ones that represent the control flow. From the control box, there are eight lines connected to the data box, representing the connection between the control connector hierarchy at the top and the hybrid components at the bottom. These connections are responsible for carrying the control flow between the control box and the data box. Likewise, there is one line that carries the data flow from the data box to the control box. This one connection is the data input to the selector connector in the control box.

The controller plant structure is visible in SCADE, where the block that contains the safe state machine represents the controller part, and the parts that contain Lustre nodes represent the plant part. The plant part of SCADE, containing the Lustre blocks for data processing, is similar to our plant part that contains components responsible for data processing functions. However, for the controller part, our controller box contains a hierarchy of control connectors, defining the control flow structure for the system — instead of a state machine like in SCADE.
Other than ProCom and SCADE, PECOS \cite{90} and Koala \cite{103} are examples of component models in the embedded systems family. The PECOS component model is designed to meet the requirements of building software for field devices, while the Koala component model is designed for products in the family of consumer electronics. As we have reviewed earlier, the PECOS component model defines the control flow in the scheduler which drives every movement of data in the system. Because of this, control and data are not separated in the PECOS model. In the Koala component model, components that are defined as architectural units are connected by method calls through connectors. The connection carries both control flow and data flow, making the control and data mixed together in the Koala model.

The most interesting aspect of our component model in the context of the embedded systems domain is the separation of control flow and data flow. Having a separation of control and data fulfils the requirement for the hybrid characteristic of embedded systems that contain the modelling of both continuous and discrete behaviour. Moreover, the ability of our model to capture the control flow structure as an identifiable entity in the architecture, provides the opportunity for connector reuse in the domain of embedded systems. The controller box in Figure 7.11(b) is a software entity that is built for an embedded systems domain, representing behaviour that can be reused to build other applications in the same family.

7.5 Chapter Summary

In this chapter, we have performed an important evaluation of the new component model, emphasising the separation of control and data in the system. Other than the ability to model control flow and data flow, the separation of control and data is one of the sought-after features of a model for embedded systems. This is because the separation principle supports the modelling of continuous and discrete behaviour for the embedded systems domain.

A case study of a climate control system is used to perform our evaluation. The design of a climate control system using our model is compared with the design of this climate control system using the ProCom and SCADE models. From the comparison, we conclude that our model can be used in the domain of embedded systems. Additionally, our model has the advantage of offering
software reuse for the control flow structure, which is captured in the hierarchy of control connectors.
Chapter 8

Using our Component Model for a Generic Domain

In Chapters 5 and 6, respectively, we have established the elements of our new component model, and its execution semantics. The practicality of our approach is demonstrated in Chapter 7, by means of a detailed example of a climate control system case study, and a comparison to two important pieces of related work, namely SCADE and ProCom. Using this comparison, we highlighted the suitability of the new component model, to be used in the domain of embedded systems.

Having a component model that models control flow and data flow brings out another possible area of interest in CBSD, that can benefit from the component model proposed in this thesis. Specifically, the interest arises from the domain modelling approach in Feature Oriented Domain Analysis (FODA) [60], which contains the modelling of control flow and data flow in the functional model. The features of our component model, that has explicit modelling of control flow and data flow, are well-suited for the idea of defining a domain model for the CBSD approach. In other words, we want to use the new component model to define a component-based domain model in the CBSD approach, similar to how FODA is used to define a domain model in the software engineering approach.

In this chapter, we first introduce the concept of a domain, and domain modelling, in software engineering (Section 8.1). Secondly, we explain the FODA approach in detail, as it serves as important background for this chapter in Section 8.2. We further explain the concept of FODA with an example of a vending machine. We then introduce the concept of a component-based domain model in
CHAPTER 8. USING OUR C.M. FOR A GENERIC DOMAIN

Section 8.3 Finally, an analysis and discussion for this chapter is presented in Section 8.4.

8.1 Domains and Domain Modelling

The word *domain* can be interpreted in different contexts, depending on the goal that it is intended to serve. In [99], Simos classifies the usage of the term “domain”: (a) a domain as a real-world point of view, and (b) a domain as a set of systems approach. The former definition embodies the knowledge of the problem area (problem space), without the involvement of the software part, as a means to support the problem area. This view is adopted by the object-oriented community, which defines a domain as an area of knowledge or activity characterised by a set of concepts and terminology, as understood by the practitioners in that area [20].

In the software reuse community, which focuses on building reusable software with domain engineering, the term domain includes not only knowledge of the real-world as a problem area, but also knowledge to build software systems in order to support this problem area. This definition of “domain” encompasses both the problem space and the solution space, and it corresponds to the “domain as a set of systems”. Not only are both views important, but the knowledge to map from the real-world view to the set of systems view is also considered an essential part of the domain. This is exemplified by the literature [7, 51, 101, 19, 14, 59, 89], which is concerned with mapping the real-world view of the problem, in the problem space, into the set of applications view in the solution space. For example, Czarnecki formulates the definition of a domain as follows [37]:

Domain: An area of knowledge

- scoped to maximise the satisfaction of the requirements of its stakeholders
- including a set of concepts and terminology understood by practitioners in that area
- including the knowledge of how to build software systems (or parts of software systems) in that area.

Under this definition, a domain is perceived as a complete area of knowledge with a clear link between the problem and the solution space. We regard a domain
as the knowledge that encompasses the real-world view of the system, and the solution of building the software to support this real-world view. In order to capture the domain knowledge in the context of building the software systems, we need to perform domain modelling.

Domain modelling is one of the activities in the domain analysis process. Domain analysis is the process of identifying, capturing, and organising information that is used to develop a class of related software systems. This way, we make it reusable in the future for creating new systems in the domain [93]. The main objective of domain modelling is to increase reuse of domain knowledge, which includes not only the code but also the requirements, specifications and designs. Domain modelling produces a domain model, which represents the commonalities and variabilities among the members of a family of software systems. This applies to not only existing applications in the software systems family, but also to future applications in the same family.

There are numerous domain modelling methods, all proposing various kinds of modelling aspects, tools and steps in order to perform the domain analysis activity. This variation exists due to the various intentions for reusing software elements, as surveyed in [94, 44]. In our work, we aim to achieve reuse at the level of function and architecture, as it has been targeted by the CBSD community [31]. We closely follow the FODA approach, for the target of reuse in FODA is also at the level of function and architecture. By “follow”, we mean taking the FODA domain model as the reference and benchmark in proposing a component-based domain model.

8.2 Feature Oriented Domain Analysis (FODA)

Feature Oriented Domain Analysis (FODA) [60] is a domain analysis method that defines a domain analysis process and describes the products of the analysis. The methodology focuses on designing a family of software systems where reusable software is constructed, in such a way that they will be useful in new contexts that have not yet been specified. The domain is modelled to accommodate the future development of the software in the same family. This is done via analysis of the commonalities and variabilities in the domain. A commonality analysis identifies useful abstractions that are common to all members of the system family. The common parts will be the main source of reuse. Variabilities are discovered within
the common parts, indicating potential sources of change over the life-time of the software family.

There are three main phases in FODA, namely context analysis, domain modelling, and architecture modelling. The context analysis ensures proper scoping for the domain and produces a context model. A domain is described in a domain model produced from the domain modelling activity. The domain modelling describes the problem space in the domain that is addressed by the software. The architecture modelling involves creating the software architecture, for implementing the solution for the problem domain. In our work, we focus on the domain modelling part of FODA, as our intention is to introduce the usage of the new component model for domain modelling in the CBSD approach.

8.2.1 A Domain Model in FODA

Domain modelling produces a domain model that is described in a context defined during the context modelling (context analysis) phase of FODA. We focus on feature modelling and the functional modelling activities for creating a domain model. Feature modelling produces a feature model, and functional modelling produces a functional model. The central element of the domain modelling is the feature model, since it describes the commonalities and variabilities of the applications in the domain. The functional modelling models the function and behaviour for the domain. The following is an introduction of feature models and functional models.

8.2.1.1 Feature Models

A feature model captures the end-user’s understanding of the application capabilities in a domain. It models the domain requirements in a structured form, appearing as a tree-like modelling for features. A feature model documents the commonalities and differences of applications in a domain, and the dependencies among variable features [60, 37]. The usage of a feature model is important in domain modelling, because it generalises and parametrises all other models in the domain analysis. A feature model sketches the system attributes as how they are perceived by the end-user. In essence, it represents a hierarchy of properties of domain concepts. There are three kinds of features in FODA, namely mandatory, alternative and optional.
An example of a feature model created from the FODA domain modelling activity is shown in Figure 8.1. At the root of the hierarchy, there is a feature called ATM, which is the concept feature in the feature model. The concept feature represents the whole class of solutions for the domain. Sub-features are hierarchically structured below the concept feature, describing the refined properties. In Figure 8.1 the ATM has three mandatory features (card reader, authentication unit and money slot) and one optional feature, which is the receipt printer. There are two optional features for the authentication unit: PIN check or fingerprint check. These two optional features are alternatives, which means that only one can be chosen at a time.

![Figure 8.1: An example of a feature model.](image)

### 8.2.1.2 Functional Models

A functional model captures the data flow and control flow, and the processing for both data and control. In general, a functional model specifies the functional and behavioural aspects of a system. The modelling of the functional aspects includes the specification of input data, output data, data transformations, the logical structure, and the data flows. The behavioural aspects include the specification of the events, states, conditions, and state transitions, the focus being on control flow. The functional part is specified using the data flow modelling technique, and the behavioural part is specified using the finite state machine modelling technique.

FODA demonstrated the use of Statemate Activitycharts and Statecharts to model the functions and the behaviour, respectively. These models are rooted from the Structured Analysis and Design Technique (SADT) [40][110]. The SADT emphasises the functional view of a system, where the data flow is the
major focus for analysis of the requirements. Later, the SADT technique was extended to include explicit modelling of control specifications involving control flow modelling [95, 107, 55, 111].

Figure 8.2: The functional model.

Figure 8.2 shows a summary of the modelling elements for the functional model. The solid lines represent the data flow, and the dashed lines represent the control flow. Starting from the context models for both the data model and the control model, decomposition of functions is performed. This results in the creation of Data Flow Diagrams (DFDs) for each level of decomposition in the data model. The Control Flow Diagrams (CFDs) that are identical to the DFDs are sketched in the control model. For each level in the CFD that involves interaction with the data model, a control specification is documented to specify the behaviour. All of these modelling entities are explained in detail in the following.

Data Model

The data model specifies the functional parts of the system. It consists of a Data Context Diagram, a number of Data Flow Diagrams (DFDs), and Process Specifications (PSPEC) for the DFDs.

1. A Data Context Diagram (DCD) models the data flows between the system and its environment. The main purpose of this diagram is to model the external entities (often called terminators) that communicate with the system. The system is modelled as one data transformation and it represents the most abstract view. Figure 8.3 shows an example of a DCD. Here,
T1 and T2 are the terminators that interact with system S via data flows f1, f2 and f3.

Figure 8.3: A Data Context Diagram (DCD).

2. Data Flow Diagrams (DFDs) refine the data model in a DCD. Starting from the DCD, DFDs define a hierarchy of views by decomposing data transformations, until reaching the primitive level. Each DFD is constructed by using the elementary building blocks shown in Figure 8.4. There is a data transformation, a data flow, and a data store.

- Data transformations or data processes denote a transformation from an arbitrary number of input data values to an arbitrary number of output data values. A data transformation is modelled as a solid circle line.

- Data flows are represented as unidirectional arrows, connecting one data transformation to another. They are denoted as solid line arrows.

- Data stores provide temporary storage for data in data flows. They are denoted as double solid lines.

3. Process Specifications (PSPECs) document the details of the functional specification for primitive data transformations (the leaf data transformations in the hierarchy).

Figure 8.4: Elementary building blocks for a DFD.
CHAPTER 8. USING OUR C.M. FOR A GENERIC DOMAIN

Control Model

The control model specifies the control aspects of the system. In particular, it describes the behaviour of the system by the modelling of control flow. This model interacts with the data model through control prompts and data conditions. The control model consists of a Control Context Diagram (CCD), a number of control flow diagrams and Control Specifications (CSPECs).

Control Context Diagram (CCD) similar to a DCD, this diagram describes the system interaction with the environment. But instead of data flow, it shows the control flow exchange between the system and the terminators.

Control Flow Diagrams (CFDs) augment the DFD hierarchy with control information. The data transformations in DFDs are modelled in CFDs, with the modelling of control flow. Figure 8.5 shows an example of a CFD. CFDs contain control modelling which comprises of the control flow modelled as dashed arrows, the Control Specifications (CSPEC) bars modelled as solid lines, and the control store modelled as a horizontal double solid line. Similar to the data stores in DFD, the control stores are used in the control model to store control information. Data transformations are enabled/disabled or triggered by the control process that is hidden in CSPEC bars. CSPEC bars represent control processes that coordinate data transformations by enabling/disabling or triggering them. The control process is known as control transformation, hereafter referred to as the CT. It transforms control signals to control prompts for controlling the data transformations.

Control Specifications (CSPECs) document the behaviour of CSPEC bars in detail. The behaviour documented in a CSPEC is in the form of an automaton that shows how the CT reacts to control signals and data conditions.

In the remainder of this thesis, we adopt the notation from [107, 55, 111] for the functional model. This notation show the interaction between the CT and the DT, rather than modelling them separately in DFDs and CFDs. The behaviour of the CT is described using a state transition diagram (STD). In other words, the state transition diagram is the control specification for the CT.
Figure 8.5: An example of a control flow diagram.

Figure 8.6: Interaction between control transformation (CT) and data transformation (DT) in a functional model.

Figure 8.6 depicts the interaction between a control transformation and a data transformation. A control transformation that is hidden in the bars from the control flow diagram interacts with the data transformations (DTs) by sending a prompt signal to the DTs. A CT can receive any control signal in the form of an event, and transform this control signal to another form of control signal. The latter can be either a control prompt, or a response that will be an event to another CT. In addition, a data transformation may generate signals to control transformation, in the form of data condition that results in reaction to control coordination. Control transformation coordinates control in the system by controlling (via prompt) the enabling/disabling and triggering of data transformation.

A data transformation being enabled by the CT is basically being “switched on”, and will perform the data transformation function as long as it receives data from the data flow. Switching on the DT will make the DT ready to receive
the input data, and transform it to produce the outputs. Once this data transformation receives the disabled prompt from the CT, the DT is no longer able to transform its input, even when the data is available from the data flow. A data transformation that receives a trigger prompt will perform the data transformation function once at the immediate instance and taking zero time to complete.

8.2.2 Hierarchy in Functional Models

The methodology in SADT is based on the philosophy of functional decomposition, which involves a top-down approach to refining the model. Functional decomposition in a functional model yields a hierarchy of modelling, beginning from the context diagram at Level 0. The decomposition process refines the function as we go down the hierarchy. The functions at Level $n+1$ are more detailed than the ones at Level $n$. Figure 8.7 shows an example of a functional model that has four levels in the hierarchy. The three columns illustrate three ways of representing the model: (i) the data model represents the data aspects, (ii) the control model represents the control aspects, and (iii) the data and control model sketches both data and control aspects in the same diagram.

Starting from the context model at the very top (Level 0), the functional structure and the behaviour are specified in the data model and control model, respectively. The data transformations ($DTs$) represent functions that are connected by data flows. The functional decomposition stops at primitive data transformations ($PDTs$). In this example, there are two PDTs at Level 2 (1.2 and 2.1), four PDTs at Level 3 (1.1.1, 1.1.2, 1.1.3 and 2.2.1), and three PDTs at Level 4 (2.2.2.1, 2.2.2.2 and 2.2.2.3). Identical to the DFD hierarchy, the Control Flow Diagram (CFD) models the control flows (dashed lines). There can be a CT associated with the DFD at each level. This is shown in the third column, which contains the modelling of both control and data. The CT receives input signals in the form of control flows. It is responsible for processing or transforming these control signals, and enabling/disabling or triggering the data transformations. In the data and control model in Figure 8.7, there are two CTs at two levels: (i) ct-A at Level 2, and (ii) ct-B at Level 3.

Note that at any decomposition level of the functional model, a PDT may produce a feedback control flow. That is, a PDT may output a control flow signal in the form of a data condition to a CT at the same level, or it may arrive at a CT at another decomposition level, bypassing the CT at its own level. For
Figure 8.7: Functional model decomposition.
example, in Figure 8.7, the PDT 2.2.2.2 at level 4 generates a data condition to ct-A at level 2, bypassing the CT at level 3.

8.2.3 Vending Machine Example

In order to explain the FODA domain model, we use a vending machine example adapted from [55]. Figure 8.8 shows a simple feature model that we constructed for the domain of vending machines. The feature model shows that a vending machine carries two main functions, namely accept payment and dispense product. There are two ways of payment: cash and credit card. Cash payment is a mandatory feature, which means that all vending machines must be able to accept cash payment. Accepting payment by credit card is an optional feature of a vending machine. From this feature model, we know that in the domain of vending machines, there are two applications or systems that can be developed. One system carries the mandatory feature, which serves the most basic function of a vending machine, i.e. a vending machine that can dispense products paid for in cash only. Another possible system is a vending machine which carries the basic function plus the additional function to accept credit card payment.

![Figure 8.8: Feature model for vending machines.](image)

For simplicity, we present the functional model that covers the mandatory feature of a vending machine, which is a vending machine that dispenses products paid for by cash only. Figure 8.9 shows the context model for the vending machine, the environment being the customer. The latter interacts with the vending machine by putting objects into the coins slot, making a selection, and requesting coins to be returned. The vending machine interacts with the customer by dispensing the product selected, returning the change (if the amount deposited is greater than the product price, or the customer cancels the transaction) and
notifying the product availability.

Figure 8.9: Data and control context model for the vending machine.

The first level of decomposition is shown in Figure 8.10. It shows the modelling of the DTs and the CT that refine the context model at Level 0. At this level, $\text{ctA}$ coordinates three DTs via a control prompt, in response to the control signals that it receives from a few DTs and the environment. Specifically, $\text{ctA}$ coordinates data transformations $\text{Dispense Change (2)}$, $\text{Get Valid Selection (5)}$ and $\text{Dispense Product (6)}$. The rest of the DTs, namely $\text{Get Customer Payment (1)}$, $\text{Validate Payment (3)}$ and $\text{Get Product Price (4)}$ are working independently and are driven by data. There are five control signals, namely $\text{coin detected}$, $\text{coins return request}$, $\text{sufficient payment}$, $\text{product dispensed}$ and $\text{product available}$. The CT reacts to these control signals by enabling/disabling and triggering the relevant DTs. In this example, once a coin is detected, the data transformation $\text{Get Valid Selection (5)}$ is enabled and ready to get the selection from the customer, modelled as data flow. Subsequently, the data transformations $\text{Get Customer Payment (1)}$, $\text{Validate Payment (3)}$ and $\text{Get Product Price (4)}$ will calculate the payment against the selection to decide whether it is sufficient. If the payment is sufficient, then a control signal in the form of a data condition $\text{sufficient payment}$ is sent to $\text{ctA}$, and this causes disabling of the data transformation $\text{Get Valid Selection (5)}$, and the triggering of data transformations $\text{Dispense Change (2)}$ and $\text{Dispense Product (6)}$. The behaviour that we described here is summarised in the STD in Figure 8.10.

The data transformations $\text{Get Customer Payment (1)}$ and $\text{Dispense Change (2)}$ are further decomposed at the next level, where each of them contains a control transformation (with a state transition diagram) and PDTs. The rest of the DTs in Figure 8.10 — namely $\text{Validate Payment (3)}$, $\text{Get Product Price (4)}$, $\text{Get Valid Selection (5)}$ and $\text{Dispense Product (6)}$ — are primitive data transformations that are not decomposable. Figure 8.11 and Figure 8.12 show the decomposition for the data transformations $\text{Get Customer Payment (1)}$ and $\text{Dispense Change (2)}$, respectively. Each of them has a CT where the behaviour is defined in the state
Figure 8.10: Level 1 decomposition.
transition diagram. In Figure 8.11, \( ctB \) will trigger a data transformation **Clear Payment** (1.2) only when the payment is sufficient. This is to reset the amount of payment stored for a new transaction. In Figure 8.12, \( ctC \) will trigger data transformations **Get Change Coin** (2.1) and **Get Payment Coin** (2.2), based on two control signals that it receives. The triggering will ensure that an appropriate amount of coins are returned.

![Figure 8.11: Get Customer Payment (1) at Level 2 decomposition.](image)

Figure 8.11: Get Customer Payment (1) at Level 2 decomposition.

![Figure 8.12: Dispense Change (2) at Level 2 decomposition.](image)

Figure 8.12: Dispense Change (2) at Level 2 decomposition.

To get an overview of the decomposition process, we lay out Figure 8.10, Figure 8.11, and Figure 8.12 in one diagram as depicted in Figure 8.13. In Figure 8.13, the data transformations **Get Customer Payment** and **Dispense Change** are replaced
by their internal models, shown in Figure 8.11 and Figure 8.12 respectively. In doing so, we have all primitive data transformations in one diagram.

8.3 Component-Based Domain Models

As explained in the previous sections, FODA establishes a process for domain analysis, by introducing the use of a feature model together with the context and functional models. The functional model captures the data flow and control flow, and the processing for both data and control. This is achieved by using DFDs and CFDs with the control specified in the form of state transition diagrams. We want to introduce similar concepts for domain modelling in the CBSD approach, where we use the component model that we defined in this thesis to be the tool to perform the functional modelling for a domain.

The FODA approach is generic, in the sense that it can be used to build software systems for any domain. Similarly, it is our intention to introduce the approach of using the new component model to construct a domain model for any type of domain. We have shown the suitability of the new component model in the domain of embedded systems in the previous chapter. However, due to our component model’s features being similar to the features in FODA domain modelling, we foresee the suitability of the new component model for being used for any domain.

This chapter evaluates the suitability of the new component model, to be used for domain modelling in component-based approaches. Furthermore, by performing this evaluation for the component model, we take a first step of demonstrating a new way of performing domain analysis by using components and connectors. Components and connectors are the basic building blocks for software architecture. By producing components and connectors in domain analysis, we create an instant software architecture for a domain. This is different in FODA, where the architecture is created by performing a mapping of domain model to a generic design in the form of an architecture [91]. For the component-based domain model approach, we create the software architecture early in the analysis stage. By doing this, we provide an opportunity to formally verify the software architecture against the requirements in the analysis phase.

Before we can demonstrate the usage of the component model to build a domain model in the CBSD approach, we need to show the equivalence of the
8.3. COMPONENT-BASED DOMAIN MODELS

Figure 8.13: Laying out Figure 8.10, Figure 8.11 and Figure 8.12 in one diagram.
elements in the component model with the elements in the FODA domain model. The reason for this is twofold. First, it is important to show that the elements of the functional model in FODA and the elements of our component model are semantically equivalent. Second, the software designer who knows about the FODA approach can be introduced to the concept of using the new component model to construct a domain model in the CBSD approach.

8.3.1 Relating the FODA Functional Model to the Component Model

For each modelling element in the functional model, we tried to find the suitable corresponding modelling element in the component model. In the following, we present this near-equivalence by describing each modelling instance in the FODA functional model together with the corresponding modelling instance in our component model.

8.3.1.1 Terminators

Terminators are sources and sinks that show the external entities interfacing with the system. In FODA, control and data flow to and from the terminators. In our model, the environment can only be modelled using source and sink components that exchange data with the system. This means that in our model, there is no exchange of control flow with the environment like in CCD. Any communication with the environment needs to be modelled using the data flow that goes into the system and out from the system. Figure 8.14(a) shows an example of a context diagram that models three terminators, namely T1, T2 and T3. These terminators exchange data with the system namely d1, d2, d3 and d4. Using our component model, the same example can be modelled as illustrated in Figure 8.14(b). The data source S1 models terminator T1, whereas data sinks S2 and S3 model terminator T2 and T3 respectively.

8.3.1.2 Data Transformations

As explained in Section 8.2.2, the FODA approach is a top-down problem decomposition approach, where at the end of the process, we arrive at having all primitive data transformations for the model. This is different from a CBSD approach, in which we build systems using a bottom-up composition approach
where at the end of the process, we arrive at having a constructed system. For this reason, we consider the equivalent instance only for primitive data transformations to have the corresponding instance in our model. Therefore, hereafter, when we refer to a data transformation, it will mean a primitive data transformation.

Data transformations involve taking information (data), performing some functions, and outputting information (data). The operations of data transformations suit the definition of components as units of functionality, which transform input data to output data.

Figure 8.15: A simple data transformation and its equivalent data flow component.

In CBSD, we model a simple data transformation (that takes input data and produces output data consumed by another data transformation) as a pure data flow component. Figure 8.15(a) shows an example of a primitive data transformation \texttt{pdt1}, with one input data \texttt{a} and one output data \texttt{b}. Equivalently, using our
component model, the data transformation is modelled using a data flow component $f_1$ (with one input data port $a$, and one output data port $b$), depicted in Figure 8.15(b).

![Figure 8.15: An example of a PDT with an E/D control prompt and its equivalent E/D component.](image)

Figure 8.16: An example of a PDT with an E/D control prompt and its equivalent E/D component.

In FODA, there exist data transformations that are controlled by a control transformation (via the control prompt). These types of data transformations work under the influence of control in the system that activates/deactivates and triggers data transformations. In CBSD, we model this type of data transformations using hybrid components. Figure 8.16(a) shows an example of how a data transformation with an E/D control prompt $pdt2$ with one input data $c$ and one output data $d$ is modelled in FODA. Equivalently, using our component model, we model the data transformation with an E/D control prompt using our E/D component depicted in Figure 8.16(b). Component $f_2$ performs the data transformation whenever it is enabled by the control flow at its control port, and data
is available at its input data port. Similarly, the data transformation triggered by the control prompt in FODA is modelled as a TR component in the CBSD approach. Figure 8.17(a) shows an example of a data transformation \textit{pdt3} that is triggered by the control prompt modelled in FODA. The corresponding model in CBSD is depicted in Figure 8.17(b).

### 8.3.1.3 Data Flows

A complete data flow diagram consists of a set of data transformations interconnected by data flows. A data flow line must be uniquely identified, and it may represent one item of data or a group of individual items. There exist data stores for modelling persistent data flows. The data store is used in between two data flow lines (where the action is writing and reading) or as a source of data for a data transformation where the action is to read only.

Connecting two data transformations with a data flow line suggests that the output data from the first data transformation is directly transferred as the data input to the second data transformation. Figure 8.18(a) shows an example of two PDTs, \textit{pdt1} and \textit{pdt2}, that are connected by a direct data flow in FODA. Figure 8.18(b) shows the equivalent of modelling the same example using our component model for the CBSD approach. We used a FIFO size one data connector to connect two components \textit{f1} and \textit{f2}.

![Figure 8.18: An example of direct data flows between two PDTs, and its equivalent representation in our model.](image)

Although the direct data flow line in FODA means data is transferred directly between the two PDTs, in our model we use a size one FIFO connector. This is because we do not have a data connector that can support this semantic of data flow, which requires a synchronised communication between two components where the data produced by the first component can immediately be consumed.
by the second component. Therefore, we choose to model the direct data flow semantics using our size one FIFO data connector.

The existence of a data store in between data flow lines that connect two data transformations suggests that the data value travelling in between the two data transformations persists over time. In this case, data transformations can work independently for producing and consuming data. Figure 8.19(a) shows an example of two PDTs, \( \text{pdt3} \) and \( \text{pdt4} \), connected by a persistent data flow in FODA. Here we can use either an unbounded FIFO data connector or a FIFO of size one data connector to model the persistent data flow. Figure 8.19(b) shows one possible way to model the same example using our component model for the CBSD approach. We use an unbounded FIFO to connect two components \( f3 \) and \( f4 \).

![Diagram](image)

(a) Two PDTs with persistent data flow. (b) Two components with an unbounded FIFO data connector.

Figure 8.19: An example of persistent data flows between two PDTs, and its equivalent representation in our model.

There exists a data store in the FODA model where the function is to supply data to data transformations. A data flow from the data store to data transformations implies read-only operation by the data transformation. Figure 8.20(a) shows an example of two PDTs, \( \text{pdt5} \) and \( \text{pdt6} \), that retrieve data \( h \) from the data store. There is no writing operation for value \( h \), which implies that it is a constant value existing in the data store. Equivalently, in our approach, the data store is modelled as a data source (Figure 8.20(b)) that supplies data to the component. The non-destructive read size one buffer data connector is used to model the connection between the data source and the component. This connector suits the modelling for data values that are available all the time, like a variable in a programming language.

In addition, a data connector is used to connect between a component’s output data port to a control connector’s input data port. This is to model the data condition sent by a DT to a CT. The type of data connector used depends on
8.3. COMPONENT-BASED DOMAIN MODELS

(a) Two PDTs retrieving data from a data store. (b) Two components connected to data source via a non-destructive read size one data connector.

Figure 8.20: An example of retrieving data values from a data store by PDTs, and its equivalent representation in our model.

the behaviour intended for the data flow.

8.3.1.4 Control Transformations and Control Signals

As described in Section 8.2.1.2, control transformations (CTs) from the control flow diagram interact with the data transformations (DTs). The control transformation coordinates control in the system by controlling the enabling/disabling and triggering of the data transformation. This is done via the control prompt: the E/D prompt does the enabling/disabling of DTs, and the TR prompt does the triggering of DTs. A control transformation prompts DTs as the reaction to control signals that it receives. The control signal is in the form of either data conditions received from DTs, or events. The behaviour for a CT is specified in terms of a finite state machine, modelled as state transition diagram. Figure 8.21 shows a simple example of a CT specified with an STD. The control transformation ct1 reacts to the data condition dc1 that it receives from the data transformation dtZ. Here, there are two states involved, namely S1 and S2, with initial state S1. Upon receiving data condition dc1, the control transformation ct1 will trigger the data transformation dtA, and causes the state to change to state S2.

The role of a control connector in our component model corresponds to the CT (with the specification written in a STD) that defines the behaviour in the FODA functional model. The CT manages the processing of the data transformation by enabling/disabling and triggering of the data transformations. Therefore, equivalently in the CBSD approach using our model: (i) hybrid components are used to represent the data transformation that can be enabled/disabled or
triggered via the control port, and (ii) the control connector is used to define
behaviour similar to the control transformation with the STD, that defines the
behaviour in the FODA functional model.

Our control connectors express the behaviour in terms of control flow struc-
ture. In order to show that our control connectors are equivalent to control
transformation and its STD, in the following, we sketch the behaviour of our
control connectors in state transition diagrams. By doing this, we show that our
control connector can be used to model the behaviour in the CBSD approach,
as how CT and STD are used to model the behaviour in the FODA functional
model.

1. **Selector**: Figure 8.22 shows a selector with an STD that describes the
   behaviour of the selector. In this example, this selector has two parameters
   (there could be more; this is just to illustrate the general idea). These
   parameters are connected to A and B, both of which are either components
   or control connectors. In the corresponding state transition diagram, if we
   are at the initial state S1, then the control flow has arrived at the selector.
   Two transitions are then possible, depending on whether the condition c
   evaluates to 1 or 2 (the values 1 and 2 are just particular examples). In case
   of the former, the new state is S2, and the action A represents the control
   flow visiting the first parameter of the selector. In case of the latter, the
   new state is S3 and the action is B.

2. **Guard**: Figure 8.23 shows the state transition diagram for a guard con-
   nected to a parameter A. Again, the initial state is S1, but this time, the
   condition c should simply evaluate to true. If this happens, a transition to
8.3. **COMPONENT-BASED DOMAIN MODELS**

S2 is made, along with the action A. The latter represents the control flow visiting the only parameter of the guard.

Figure 8.22: An example of a selector with its STD.

3. **Sequencer**: Figure 8.24 considers the case of a sequencer with two parameters A and B. As in the previous case, the initial state is S1, but no transition conditions are present. This is because the transition is purely initiated by virtue of the control flow visiting the sequencer. First, the state is changed to S2, and action A is performed, meaning that the first parameter is visited. Then, as dictated by the control flow, a transition to state S3 is made, and action B is taken, representing a visit to the second parameter.

These example state transition diagrams are meant to further illustrate the behaviour of control connectors, and to show that our control connectors are equivalent to control transformations. However, here we do not address how a complete hierarchy of control connectors would be represented as a (big) state transition diagram. In addition, as a loop connector is always a top-level connector for such a hierarchy, we do not express a loop connector as a state transition diagram either.
Example 1

Consider an example of a FODA model in Figure 8.25(a) that shows a control transformation ct2 with the behaviour defined in an STD. There are two data transformations, dtB and dtC, that are controlled by ct2, which reacts to data condition dc2 (this is described in the STD). If the data condition dc2 is true, the actions of triggering dtB and dtC will take place in sequence, and the initial state S1 will change to state S2. Figure 8.25(b) shows how the same example is modelled using our component model in the CBSD approach. There are two tr components (component B and component C) that are composed by a sequencer. We connect a guard connector on top of the sequencer to ensure that triggering of component B and C in sequence happens only when the condition dc2 is met. The data condition dc2 is produced by component Z.

Example 2

Consider an example of FODA model in Figure 8.26(a) that shows a control transformation ct3 with the behaviour defined in an STD. There are two data transformations, dtD and dtE that are controlled by ct3 based on the data condition dc3. The state transition diagram described the branching behaviour for ct3 where the value of dc3 determines the triggering of either dtD or dtE. There will be a change for initial state S1 if dc3 evaluates to 1 or 2 to state S2 or S3 respectively. Figure 8.26(b) shows the same example constructed using our model in CBSD approach. Component D and component E are modelled as tr components, and they are composed by a selector connector which receives data.
8.3. COMPONENT-BASED DOMAIN MODELS

(a) Example 1 in FODA functional model.

(b) Example 1 in Component-based domain model.

Figure 8.25: Example 1.
input dc3 from component Z. The semantic of Selector connector that defined a branching behaviour ensures that either component D or E are triggered at a time based on condition dc3.

(a) Example 2 in FODA functional model.

(b) Example 2 in Component-based domain model.

Figure 8.26: Example 2.

8.3.2 A Component-Based Domain Model for the Vending Machine Example

Here, we demonstrate the construction of a component-based domain model using the vending machine example described in Section 8.2.3. The feature model remains as the model that expresses the end-user understanding of the features of the system developed for the domain. To show the idea of using the component-based domain model, we focus on the functional model in FODA. Given a FODA functional model, we should be able to reconstruct that model using our component model, to create a component-based domain model.

The CBSD approach in nature is a bottom-up approach, since the basic idea
is to compose a larger system from smaller parts. That is, starting from the smallest unit, the architecture is built incrementally by composing components. In contrast to the FODA approach, which employs a top-down way of refining the system, our approach employs a bottom-up way of building a system.

In this section, using the vending machine example, we illustrate the process of constructing a component-based domain model for a domain of vending machines. Based on the feature model shown in Figure 8.8, here we construct a component-based domain model for a basic vending machine that contains all the mandatory features for the vending machine. To clarify the process of constructing the component-based domain model, we construct the system by parts. Also, we start from the bottom level hierarchy of Figure 8.13, since we do the bottom up composition of components.

We start by creating components to perform the basic functions of getting the customer payments. Figure 8.27 shows the components and connectors that are involved in building the “get customer payment” function. There are five components and one control connector.

We now explain each of the modelling entities constructed to perform the get customer payment functions.

1. Component (1) performs the validate coins function — the operation of the vending machine starts when the user enters an object into the coin slot. This component detects the object entered by the user, and validates it against the coin parameter. It will output the physical coins labelled as coins entered, the coin value and the slugs. For example, if the object entered is a 20-pence coin, the output value for this component will be 20 for coins entered and coin value, and 0 for slugs. If the object entered is not a coin, the output value for this component will be 0 for coins entered and coin value, and 1 for slugs.

2. Component (2) performs the clear payment function — this component is responsible for clearing a payment by producing a clear flag value that will be consumed by component (A). This value is of Boolean type, always carrying the value true when it is produced as the output for component (2).

3. Component (3) performs the accumulate payment function — payment entered by the user is accumulated and calculated by this component. This
Figure 8.27: Get customer payment components and connectors.

Figure 8.28: The STD for the guard connector in Figure 8.27.
component has coin value, payment and \( x \) as the input. The coin value is the current value of a coin entered by the user that needs to be accumulated. The payment value is the feedback input for the component, so that the value can be added and accumulated as the user enters new coins. For input \( x \), if the value of \( x \) is \text{true}, the component will reset the payment to the value 0, and set the value \( y \) to \text{false}. The \text{false} value of \( y \) is needed to set the value of \( x \) to \text{false}, so that the next payment input will be accumulated again. Note that the value of \( y \) is forwarded only when the data coordinator (GD) receives \text{true} input for the clear flag.

4. Component (4) performs the accumulate coins function — this component will accumulate the physical coins entered by the customer. Since in our model, we do not have a persistent storage for data like a table in a database, we use this component to accumulate the data. The data which is the coins entered are accumulated by saving the value of coins entered in a list. There are two inputs to this component: coins entered and held coins. The data coins entered represents the physical coins entered into the vending machine at any time. The component will store information about the coins, such as the coins’ value and the total number of coins in a list. This list is produced as the output data held coins, which is fed back to the component as an input. This ensures that the list accumulates and stores the information over time about the physical coins held in the machine.

5. Component (A) is an additional component created to handle the reset signal in the form of a Boolean value sent by component (2). The input to this component is the clear flag data of Boolean type received from either component (2) or the data guard that forwards the just reset value from component (3). This component will produce output \( y \) of integer type, which is a signal to component (3) to reset the payment accumulated. The existence of component (A) is to ensure that whenever the clear flag signal is sent to component (3), it will reset the payment only once. Since component (3) is a data flow component, it requires the value \( x \) all the time for performing its operation. We ensure this by putting a non-destructive read size one data connector to carry the signal value for resetting. Since component (2) produces only a \text{true} value for the clear flag, component (A) ensures that component (3) will always receive the value \text{false} otherwise.
CHAPTER 8. USING OUR C.M. FOR A GENERIC DOMAIN

6. Control connector **Guard** — the function of the guard is to drive the execution of data transformations for component (2), by triggering the component only when the data **sufficient payment** of Boolean type is true. Figure [8.28] shows the STD to describe the behaviour of the guard in Figure [8.27].

The next part is to construct the model for the dispense change function. Figure [8.29] shows the components and connectors that are connected to perform this function. There are four components and one control connector. In the following, we explain each of the modelling entities involved in Figure [8.29].

1. Component (5) performs the get change coin function — this component is responsible for getting the change coin by retrieving the stored coins in the list of **held coins**. Based on the **change due** input, the appropriate **coins** value that represents change coins output is produced.

2. Component (6) performs the get payment coin function — the current payment made by the customer is read as the input **payment** and based on this, the stored coins in the list of **held coins** are retrieved. The output **coins** is the payment coins produced to be dispensed.

3. Component (7) performs record return coins — this component is responsible for recording the returned coins before it dispenses **returned coins** from the machine. There are two data connectors connected to the input port **coins**. They are linked to both components (5) and (6). Since only one of these components is active at any particular time, there will only be one **coins** value written to the **coins** data port of component (7). Note that we choose a data connector FIFO of size one to model this behaviour of data flow, to ensure that only one of the data connectors contains the data for the **coins** value.

4. Component (B) performs the evaluation of control signals for dispensing the change. There are three control signals involved in the functional model for dispense change that need to be evaluated to decide how the coins are returned to the customer. In other words, there are three data conditions that are evaluated in order for the control to decide which components are to be triggered. These data conditions are **product available**, **coins return request** and **sufficient payment**. The output **c1** is the decision data, of integer type, that will have a value of either 1 or 2. Regardless of the data conditions **coin**
8.3. COMPONENT-BASED DOMAIN MODELS

return request, component (B) will set the value of c1 to 1 if product available equals true and sufficient payment equals true. This means that component (5) is triggered to return the change coins to the customer. The value for c1 is set to 2 if coin return request equals true or/and product available equals false.

5. Control connector Sel is a selector connector that is used to route the control flow to either component (5) or (6). The selector routes the control flow to either of the branches, based on the data condition c1. If the value of c1 equals 1, the get change coin function (component (5)) will be triggered. If the value of c1 equals 2, the get payment coin function (component (6)) will be triggered. Figure 8.30 shows the STD which describes the behaviour of the selector in Figure 8.29.

Figure 8.29: Dispense change components and connectors.

Figure 8.30: The STD for the selector connector in Figure 8.29.

The rest of the primitive data transformations and the control transformations are constructed using our model, as in Figure 8.31. We describe in detail the function for each modelling entity.
1. Component (8) performs the validate payment function. This is a data flow component which takes two input data items: payments and price. The current accumulated payment input by the customer is received from component (3). The payments data is compared against the price value received from component (9). There are two outputs produced by component (8): sufficient payment of Boolean type, and change due of integer type. The sufficient payment value acts as a flag for the guard connector in Figure 8.27. Component (8) calculates the change that needs to be returned and writes the value at the change due output port.

2. Component (9) performs the get product price function. This component reads the list of product prices and retrieve the price of the product selected by the customer. This is done by comparing the data input price table with the data input valid selection. The output is the price which represents the price for the product selected.

3. Component (10) performs the function of getting a valid selection from the customer. It is a hybrid E/D component that operates only when it is enabled by the control flow. This component receives two data inputs: customer selection and products. The customer selection input is the selected product entered by the customer, and the products input is the product information including the product availability in stock. Once the component is enabled, it is ready to consume data at the input data port. In this case, component (10), in its enabled state, waits for the customer to enter the product selection. Once both customer selection and products are entered and available, the component will check whether the selected product is in stock. If it is available, then output product available is set to the true value and valid selection is set to a particular product number. The latter refers to the product that has been selected by the customer.

4. Component (11) performs the dispense of product function. The physical product is dispense by this component, that is modelled as a hybrid trigger component. It receives two data inputs: valid selection and products. Based on the valid selection value, the products that is retrieved from the stock is dispensed.

5. Component (C) performs the evaluation of control signals product available,
coin return request and sufficient payment. The output c2 produced by component (C) is used as the selection condition for the selector connector. If the value of c2 equals 1, the control flow is routed to the first branch on the left, which in sequential order performs the following: disabling the get valid selection function of component (10), triggering the act of dispensing the product of component (11) and routing the control flow down the hierarchy to perform the dispense change function. If the value of c2 equals 2, the control flow is routed to the second branch on the right. This, in sequence, will first disable the get valid selection function (component (10)) and then route the control flow down the hierarchy to perform the dispense change function.

6. Control connectors: a control flow structure is constructed using control connectors based on the behaviour described by the STD for ctA in Figure 8.10. Figure 8.31 shows the composite control connector constructed based on the behaviour of ctA. We draw dashed line boxes to represent the triggering of Dispense Change. On top of the hierarchy of control connectors, there is a loop connector that is used to feed the control iteratively for the vending machine. We sketch an STD to describe the behaviour of the connected control connectors in Figure 8.31 and this is shown in Figure 8.32.

Finally, we connect the partial architecture models together, to form the complete architecture model that represents the component-based domain model for the vending machine. The final architecture model is depicted in Figure 8.33. The bottom part of the model consists of components that perform data transformation functions, while the upper part consists of the hierarchy of control connectors. Equivalent to the functional model shown in Figure 8.13, we capture the modelling of control flow and data flow for a domain model using a CBSD approach.

8.4 Discussion

The resulting architecture in our model consists of a layer of components at the bottom, with a hierarchy of control connectors at the top. The structure of our architecture model clearly separates the control flow and data flow modelling.
Figure 8.31: The rest of the components and connectors for the vending machine example.

Figure 8.32: The STD for the hierarchy of control connectors shown in Figure 8.31.
Figure 8.33: The component-based domain model for the vending machine.
This is different in the FODA functional model, where the control flow and data flow are intertwined.

With the example of the vending machine domain model that we used for presenting the component-based domain model in this chapter, we have shown that our component model is feasible for constructing a domain model, similar to how the FODA functional model is used to construct the FODA domain model. What is not shown here is the usage of the feature model, with respect to our component-based domain model. In FODA, the variability expressed in the feature model is used in parametrising the functional model. As the model is refined, the variability is incorporated into the model. Based on the feature model shown in Figure 8.8, the variability for the vending machine involves an additional function, which is the feature for payment by credit card. We can see that this requirement involves both functions and behaviour. The payment by credit card is a step that needs to be coordinated, and the processing of payment function requires data transformation operations. If we introduce this additional function into our architecture modelled in Figure 8.33, this means that we are adding possibly new hybrid components and control connectors. This is because the new coordination requires additional control connectors, and the new data transformation function needs additional hybrid components.

The FODA approach pays less attention to the design and implementation for a domain. In [91], the mapping of a domain model to an architecture is demonstrated with the focus on the process of creating a generic design, which can be reused in the product family. With the approach of mapping the domain model, the architecture is not created in the analysis phase, but in the design and implementation phase. What we want to achieve with our component-based domain model approach is to construct the architecture during the domain analysis phase. Instead of having to come up with a design for the architecture based on the domain model created during analysis, we are able to construct the architecture during domain analysis, using the component-based domain model. Although in this chapter, we show the construction of the component-based domain model that is based on the FODA domain model, the ultimate aim is to be able to construct the component-based domain model from the domain requirements.

There exist a plethora of other domain related approaches, such as Domain-Specific Languages (DSL) [102, 86], Domain-Specific Software Architecture (DSSA) [101] and Generative Programming [37]. These approaches are different from our
approach in the sense that our approach is focusing on using a component model in the context of CBSD.

So far, the component-based domain model that we have shown is capable of capturing the modelling of control flow and data flow, in the same way as the FODA domain model captures the modelling of control flow and data flow. Although this chapter aims to present a practical evaluation for the new component model to be used in a generic domain, it provides for an important research direction, which is to introduce a CBSD approach for domain modelling. In other words, the work presented in this chapter serves as a good starting point towards a new domain approach for CBSD.

8.5 Chapter Summary

This chapter has been dedicated to the practical application of the new component model to a generic domain. It started with the introduction of the concept of a domain, by giving the definition that is used in the context of this thesis. The FODA domain analysis approach was presented in detail, as it is important to understand the approach to relate to the component-based domain model that we proposed in this chapter. The FODA domain model was explained with a vending machine example.

Afterwards, the chapter introduced the component-based domain model by describing the corresponding component-based modelling instances for every FODA modelling instance. We demonstrated the use of the component-based domain model for the vending machine example. Finally, the strengths and weaknesses for the approach were discussed.

This chapter marks the end of our work to date concerning control flow and data flow modelling in a component model. In the following chapter, we present an overall evaluation of our work.
Chapter 9

Evaluation

In this thesis, we place emphasis on the control, data and computation modelling in building software systems, and we develop a new component model for this purpose. We have set out to work towards having a component model that is both control-driven and data-driven for the component-based software development approach in Chapter 3. Chapter 4 surveys related work, and our analysis shows that none of it has the characteristics of the model that we aimed to develop. Our new component model is introduced, and explained in detail, in Chapter 5. It contains modelling elements that clearly cater for control flow and data flow modelling, while integrated in the same model, the separation of these two modelling elements is maintained. The execution semantics of the new component model is explained in Chapter 6, together with a possible implementation to realise the new component model. Then, we demonstrate the practicality of the component model by exploring the suitability of the component model in the domain of embedded systems in Chapter 7. This is because in the domain of embedded systems, it is desirable to explicitly model control and data in the same model, with a clear separation of these entities. We have also shown the use of the component model in defining a component-based domain model in Chapter 8. The modelling of control flow and data flow provided by our model is comparable to the FODA domain analysis approach. Our model is well-suited for defining a domain model in the CBSD approach.

In this chapter, the overall aspects of the new component model proposed in this thesis are evaluated. The goal is to discuss the strengths and weaknesses, as well as emphasising the originality and significance of the research work in comparison to related work.
CHAPTER 9. EVALUATION

9.1 Control, Data, and Computation Modelling

The idea of control flow and data flow modelling for software systems has already been considered in the context of programming languages [3, 12, 15, 58]. The reason for this is that these two aspects are important in program analysis, to check for logical correctness and errors. Also, control flow and data flow in programming languages carry the fundamental concepts of the two main computation models, which are von Neumann control-driven computation models and dataflow architecture data-driven computation models. The basis of these two computation models lies in the modelling of control, data and computation. Viewed in this light, these two computation models represent the two ends of the software systems spectrum. Having a component model that supports this spectrum of software systems means that we have a model that can be used to model any type of system.

Thus, it is important to make the modelling of control flow, data flow and computation explicit in a software model, while maintaining the separation between them. This is what we achieved with our component model, which expresses control flow, data flow and computation explicitly, and separates them clearly in the system architecture. This is done by defining elements that focus on data flow only, control flow only, and both data flow and control flow together. They are defined with the two computation models in mind (control-driven and data-driven), that drive the execution of the elements in the model.

Current component models do not possess the mentioned characteristics of control, data and computation modelling. Most of them are control-driven with the focus on control flow modelling. Control flow and data flow are tightly coupled in most of the models, causing data flow to be dependent on control flow. This means that it is impossible for the model to be data-driven, since data is not able to flow on its own. This is where the component model proposed in this thesis is original and unique, by virtue of a clear separation of control and data in the system architecture. This enables data-driven modelling, control-driven modelling and both data-and-control-driven modelling. In the following, we elaborate on these features by comparing the current component models with the proposed one, with control flow, data flow and computation modelling explained in detail, and show how we achieved the aim with our proposed component model.

In component models such as EJB, CCM and COM, components are defined as objects that communicate with each other via method calls. A component
9.1. CONTROL, DATA, AND COMPUTATION MODELLING

that initiates a method call to another component passes the control and data in order to complete the computation. In this case, the control flow is initiated by the caller to perform the method calls that cause the computation to take place. Method calls with parameter passing involve control flow and data flow coinciding with each other. This clearly shows that control flow and data flow are not separated in this type of model, and the computation model is control-driven. In general, a component model where components are defined as objects can only be used to model control-driven systems. Data flow is implicit and highly dependent on the control flow.

PECOS and ProCom are examples of port-connector component models that define components as architectural units, and ports are connected by connectors. Components communicate indirectly with each other via message passing through the connectors. Control flow and data flow can be easily separated by components having distinct ports for control and data, and their associated connectors. However, this separation alone does not determine whether the computations are control-driven or data-driven. The latter depends on the semantics of the component model. For example, in ProCom, a component has a control port and a data port and connectors for control ports and data ports. However, despite this separation, a ProCom system is strictly control-driven, and not at all data-driven. In a component, data flow is control-driven, i.e. it is triggered by control signals on the control ports.

In PECOS, even though components are defined with data ports only, the control flow defined in the scheduler is responsible for the data flow. It synchronises the reading and writing of the shared data variables between components. Without control, data is not able to flow on its own. This makes the PECOS system a control-driven system.

In SCADE, the underlying component model uses architectural units as components. It separates control from data flow by means of having state machines defined in an Esterel block to model the control part, and Lustre blocks to model the data flow part. Control flow cannot be defined in SCADE. Rather, state machines are used to model the control, output control signals, and these signals are used as data input to the Lustre blocks. All blocks (Esterel and Lustre) in SCADE are driven by data, which makes SCADE a data-driven model.

The X-MAN model is the only model with encapsulated components. With such components, control is separated from computation. A software component
is defined as a software unit that encapsulates computation and data. The separation of control from computation is achieved through exogenous connectors defined to encapsulate the control flow between components. It is these connectors that originate the control, and coordinate control flow among components. Computations are purely control-driven, and data flows together with control flow.

In the component model proposed in this thesis, control flow and data flow are modelled explicitly. Control connectors and data connectors coordinate control flow and data flow between components respectively. Various types of components are defined to support the computation model. Data flow components with data ports support data-driven computations, and hybrid components with both data ports and control ports support data-and-control-driven computations. Connectors exist to coordinate control and data between components that perform computations. All components and connectors are loosely coupled and independent of each other. This results in a clear separation of control and data in the system architecture.

Another advantage of our control connectors is that we can compose them to form a complicated control flow structure. Control flow is composable with the exogenous connection scheme [64, 69] employed by our component model. In models other than ours and the X-MAN model, it is not possible to compose control flow.

For data connectors in our component model, we adopted a basic data channel semantics defined in the REO coordination language. Although in REO, we can compose data channels to form a composite data channel, for our component model, we do not consider composing the data connectors. All data connectors used in our model carry the basic semantic data channel of REO in transporting the data.

The modelling elements of the component model presented in this thesis show that our component model expresses control flow, data flow and computation explicitly in a model. Clearly, in modelling software systems, the list of objects to be modelled is neither exhaustive, nor should the list of modelling elements that represent these objects be. In modelling control flow, there exist various other control structures that can be defined as control connectors in our model, such as cobegin to model concurrency and finite loops. Similarly for data flow, there are many data channel definitions in the REO coordination language that
we can use to define data connectors for our model. The use of these connectors with components requires more elaborate work. For example, the semantics of synchronous data channels in REO requires both parties on its two ends to be synchronised, in order to be able to pass the data through the channel. The synchronous data channel is the one that is responsible for ensuring that both parties work simultaneously to pass the data. This is an interesting area for future work, however, it requires detailed work for the execution semantics.

In our model, we support a hierarchical composition of control connectors, but not data connectors. The data channels are composable in REO, which means that it is possible to compose data connectors in our model. However, we have not elaborated on this matter, since we are more interested in composing control flows.

So far, in our component model, we have focused on control, data and computation modelling. We do not consider the topic of composition of components. That is, to build a hierarchy of composite components, which is often the focus of component models. The work of defining composite components in our model is not easy and trivial, as we need to consider many aspects — especially with regards to the execution semantics that involves managing both control flow and data flow. Further work would be required to investigate this.

Although we emphasised the fact that our component model distinguishes between elements of control, data and computation, it is not always the case for components to perform merely function that transform input data to output data. Components are also used to keep track of the current state of the system as shown by a few examples in this thesis. This means that the components are involved in processing the data input for the control. One way to resolve this problem in a future work is to have a composite control connector that keeps track of the system state (i.e stateful connector) and is able to process input data on its own.

Our component model contains a variety of modelling elements that have their own specifications and behaviours. Users of the component model would be confused when constructing a system that contains, say, more than eight components where some of them might exchange data with each other. This, however, can be overcome with a well-defined execution semantics embedded in a tool that can help the users in visualising the working system architecture.
9.2 Separation of Control and Data

Separation of control and data in system architecture is an established principle that is based on the “separation of concerns” concept in software system modelling. The clear separation of control and data leads to, among other things, a better reuse of parts that form the architecture. That is, the hierarchy of control connectors can be the reusable part that represents the control flow structure and the data processing part can be the reusable part that represents the common functions in a domain.

Given a set of requirements, system developers are faced with many possible ways of modelling the system architecture using the component model. Thinking about the system spectrum helps in making the design decisions. However, as modelling is a creative activity, the concept of separating control and data by the component model would complicate a simple model. Consider the simplified version of the fuel system example explained in Section 5.3.3. The resulting architecture could, for example, be constructed as depicted in Figure 9.1.

Figure 9.1: A fuel system example that operates in three main modes.

In this example, the system is required to perform the continuous transformation of input data to output data in one of the three modes. The three E/D components represent the three modes of operation while the data flow component MC performs the changing of modes based on the user input to the system.
Data switches SW1 and SW2 are used to ensure that only the component that represents the current mode will receive the input data. At the same time, the architecture contains the control part where the control connectors perform the action of enabling and disabling components to make sure that only one of the component is active at a time. Note that the control part performs the same role as the data switch in controlling the execution of computation for the E/D components. Technically, the system can still work without the hierarchy of these control connectors. If we take out the control part, we still have the control in selecting the component by enforcing the data coordinator to forward the data based on the current mode. In a sense, the use of a control connector hierarchy constitutes the central controller for a system, while the option is there to use the data flow coordination as the control. For the example in Figure 9.1, it is also possible to employ the control connector hierarchy without the usage of the data switches. Yet, this would rather depend on the execution semantics to ensure that the data input for monitoring the mode changes is synchronised with the data input for the E/D components.

Figure 9.2: A fuel system with a control connector hierarchy.

Obviously, a system that requires a more complicated form of control flow would benefit from the use of control connector hierarchy. Again, we use the full
Figure 9.3: A fuel system without a control connector hierarchy.

version of the fuel system example explained in Section 5.3.3 in Figure 9.2\textsuperscript{1}. The fuel system example shows the use of a control connector hierarchy in modelling the sequence behaviour of disabling and enabling, and triggering the hybrid components. We compared this with the fuel system model that uses only data flow to coordinate the control in executing the components as depicted in Figure 9.3\textsuperscript{2}.

In Figure 9.2\textsuperscript{1} the control flow is required to do more than just enabling and disabling components based on the mode. In particular, it is used to do the occasional triggering of TR components, and these actions should be done in sequence after the disabling and enabling for the mode. From here, it is obvious that having a control connector hierarchy as the central controller is better than having the data flow coordination to control the computations.

Having control and data parts integrated but separated in a system architecture is non-trivial to implement. From the series of examples presented, normally the input data that is required for the decision to route the control is processed by a data flow component that keeps track of the current state (or mode) of the system. This is a major disadvantage for the component model, especially for defining the execution semantics. To explain this, again, we consider the example described in Section 6.4 here in Figure 9.4\textsuperscript{2}. Using the execution semantics described in Chapter 6, we have to use the FIFO data connector to transport the

\textsuperscript{1}The complete diagram is in Figure 5.21
\textsuperscript{2}The complete diagram is in Figure 5.23
data current speed and gas pedal to the E/D components since one of the components will only be enabled based on the current input after two execution cycles. This has caused the consumption of stale data by the E/D components. Concerning the implementation of the execution semantics that we presented in Chapter 6, we reiterate the fact that the work of defining a proper execution semantics to manage both control flow and data flow should have a solid theoretical foundation. Therefore, what we did is merely experimental and exploration-based, leaving more theoretical work for future research.

Figure 9.4: A cruise control system architecture.

9.3 A Component Model for Domains

The subject of a component model for a specific domain is not new. There exist a number of component models that are designed for specific application domains. In such cases, the specific requirements for the domain are considered in the design of the component model. This causes a limitation of the component model to be used in a different domain that carries different requirements.
In the context of component models for domains, we show that our component model is not only suitable for the embedded systems domain, but also in a more generic domain where we can define a component-based domain model based on the FODA functional model. We demonstrated this in Chapter 6 and Chapter 7. In the following, we elaborate on the assessment of the usage of our component model for the domain of embedded systems and the generic domain.

9.3.1 The Embedded Systems Domain

The modelling of control and data has been suggested for the domain of embedded systems in the literature [98, 62, 13]. The characteristics and behaviour implanted in the systems built in this domain requires explicit specifications for control and data. For example, the standard plant controller structure of an embedded system shows a clear separation of control and data in the system structure. Also, the continuous and discrete behaviour of embedded systems is expressed by data flow and control flow. Clearly, this indicates that the modelling for embedded systems not only requires control and data to be distinct and separated in the architecture, but also the flows for both control and data are explicitly expressed in the architecture.

![Figure 9.5: A separation of control and data with our model.](image-url)
We achieved the aforementioned modelling requirements for embedded systems domain with our model. The distinct control and data partitioning in the architecture is achieved by having a hierarchy of control flow structure defined by control connectors for the control part, and a set of components connected using data connectors for the data part. This setting allows for clearly expressing the control flow and data flow in the model. Again, we show the separation of control and data in the system architecture built using our model in Figure 9.5.

Figure 9.6: No separation of control and data in the system architecture with ProCom.

In Chapter 7, we compared our model closely with the two main related works, namely ProCom and SCADE. These component models are designed to be used in the domain of embedded systems. In ProCom, control flow and data flow are explicitly modelled by components having distinct ports for control and data, together with connectors to connect these ports. However, control flow is not separated from data flow, since components depend on the control triggers to perform their computation. And because of this, data is not able to flow in the system without the presence of control flow. This dependency makes control and data not distinct in a system architecture built using ProCom. Figure 9.6 shows the climate control system modelled using ProCom. Control and data parts are mixed, therefore, it is not possible to draw a clear box to partition the control part and data part in the architecture, like we did in Figure 9.5.

The separation of control and data is clear in SCADE, where the control part is explicitly modelled by the state machines defined using Esterel, and the data part is modelled by Lustre blocks, clearly expressing the data flow. However, it is not possible to express control flow in SCADE, since control defined by Esterel
is in the form of state machines. In Figure 9.7 the Esterel block receives input
data and produces output data, which behaviour is similar to Lustre blocks that
also receive input data and produce output data. Thus, it is impossible to see
the control flow in the system, as we can only see data flow between the Esterel
blocks and Lustre blocks.

Based on the comparison of the models shown in Figure 9.6 and Figure 9.7,
specified respectively using ProCom and SCADE, with our model shown in Fig-
ure 9.5, we can see clearly that:

- our model achieves separation of control and data like a SCADE model,
- our model defines control flow better than both SCADE and ProCom, and
- our model is organised in the form of a plant controller structure for an
  embedded system.

Other than ProCom and SCADE, the KOALA [103] and EAST-ADL2 [10] component models belong to the family of embedded systems. Koala is a com-
ponent model designed by Philips, which focuses on software product-line develop-
ment for consumer electronic devices. EAST-ADL2 is a modelling language that
provides an information framework which intends to support the development of
components (hardware and software) in the automotive domain. Both KOALA
and EAST-ADL2 are control-driven models, therefore, there is no separation of
control and data.
9.3.2 Generic Domains

The benefit of using the FODA approach in performing domain analysis for any kind of software development is that it leads the way towards software reuse. The FODA methodology is designed to be used for software development in any kind of domain, and thus it is an approach for a generic domain. The approach uses a collection of modelling tools, with feature models being the most important modelling entity, capturing the domain commonalities and variabilities.

The FODA domain analysis method is a well-established method in domain engineering, and therefore, serves as a benchmark for our proposed component-based domain model using our component model. That is, for our approach, we constructed a component-based domain model from a FODA functional model.

In FODA, a possible solution would also be to map the FODA domain model to a chosen architecture for the implementation [91]. During this process, the decision on a suitable domain architecture is made, and the mapping operation is performed for the domain model. The architecture resulting from the mapping is used to develop applications for the domain. It is possible to choose a component model as the domain architecture, and perform the mapping to yield a domain-specific component model [73]. We do not perform this mapping: rather, we constructed a component-based domain model from the functional model. The reason for this is twofold. First, the hierarchical nested structure of a functional model with the decomposition of data transformations does not easily match with the nature of our architecture model, which is built bottom-up through composition of components. Secondly, unlike our control connector that encapsulates control, the control transformation in the functional model does not encapsulate control. Control and data are intertwined in the hierarchy of the functional model, and therefore, it is not feasible to map this structure to our component model.

Figure 9.3.2 shows a graphical comparison between a FODA functional model and a component-based domain model. The data transformation 1.1.1 at level 3 in Figure 9.8(a) outputs a control signal to control transformation ct-A, which resides at level 1. The feedback control flow output by the data transformation is an example where control flow is not encapsulated in the functional model. Also, note that in Figure 9.8(a) there is no clear separation of control and data in the model. In contrast with our component-based domain model in Figure 9.8(b), the control flow and data flow are clearly separated. The architecture is flat,
and components lie at the bottom of the hierarchy. Control connectors can be composed together to form a complicated control flow structure.

![Diagram](image)

(a) A FODA functional model.

![Diagram](image)

(b) A component-based domain model.

Figure 9.8: FODA and component-based domain models.

The significant difference between our approach and the FODA approach for functional modelling is that our approach is bottom-up instead of top-down. Even though it is natural to do problem decomposition (top-down) in analysis, it is also possible to build the architecture during analysis from the primitive elements (bottom-up) via composition. Both approaches have advantages and limitations \[92, 34, 82\], and in our case, constructing a system architecture using components requires a bottom-up approach. Our approach to building a component-based domain model for domain modelling promises scalability in the software design.
9.4 SOFTWARE REUSE

Scalability here refers to the ability of the system architecture to support large numbers of components, and interactions among components at any time within the analysis and design phase.

9.4 Software Reuse

One of the aims of this thesis is to achieve software reuse with the design of the new component model. Software reuse is the use of existing software or software knowledge to construct new software [45]. In this work, software reuse is supported in a general context, and in a domain-specific context. It is worth mentioning here that our arguments in the evaluation of software reuse are based on qualitative criteria.

In a general context, our components and connectors are highly reusable. By definition, our components are self-contained units that perform functions when the data are available at their input ports. Components communicate with the environment through ports that only specify the required data, the provided data, and possibly the control interface for enabling/disabling or triggering the functionality. Components do not know about other components. Communications between components are handled by the connectors. In particular, connectors are responsible for coordinating the control flow and data flow. They are exogenous connectors, that compose software components through data ports and control ports. As indicated by the name exogenous, connectors are entities that exist totally separated from the components.

We now show an example of code that is written for a component that performs a function calculating the ventilation for a room temperature controller, in Figure 9.9. The component CalculateVentilation is a data flow component that communicates with the environment through an input data port and an output data port. The component does not call other components to perform its function, and it does not receive any share of what is happening outside of the component. This piece of code for calculating ventilation for a room temperature controller is highly reusable, as it does not depend on other entities.

Similar to the connectors, the written code for implementing the connectors can be reused, for it represents the most common control flow structure and data flow structure for building software systems. In the implementation for our component model described in Chapter 6, the control connectors and data
public class CalculateVentilation implements ComponentFunction {

    public void run(Map<String, Object> inputs, Map<String, Object> outputs) {
        Integer adjustedTemp = (Integer) inputs.get("adjustedTemp");

        Integer vLevel = 0;

        if (adjustedTemp == 8) {
            vLevel = 20; // tuning the fan to work at 20
        } else if (adjustedTemp == 4) {
            vLevel = 10; // tuning the fan to work at 10
        } else { // no adjustment needed
            vLevel = 0;
        }

        outputs.put("ventLevel", vLevel);
    }
}

Figure 9.9: The code for a Calculate Ventilation component.

Connectors are the fixed modelling entities with implementations that are reused over and over when constructing software systems. Viewed in the general context of building software systems, the basic control connectors might be seen as similar to the common control constructs that exist in imperative programming languages. However, our basic control constructs (i.e., control connectors) offer more advantages in the sense that we can use these basic control constructs to define new control constructs. These basic control connectors are highly reusable for building many new control constructs, which realise new behaviours that can be reused again as they are or be further composed to build more new control constructs. This is possible using our component model since the basic control constructs exist as identifiable entities and can be composed with each other.

For components, reuse can be achieved through the existence of a repository that can store a set of components created, that potentially will be used again in constructing systems in the future. This will depend to a large extent on the domain, since the same components are only reusable in the context of a domain. For example, the calculate ventilation component will not be reusable if the code
is stored in the repository of bank systems. Similarly, the concept of reuse is
applicable to control connectors. As we mentioned earlier, we can build many
new control connectors from the basic control connectors. Since the basic control
connectors can be composed together to form a control flow structure with various
behaviours, it is highly desirable to store the composite control connectors in the
repository of a domain. In a domain that normally consists of a series of systems
that share a similar behaviour, the composite control connectors are definitely
desirable, being the reusable entities that represent the common behaviour. They
can easily be used multiple times if they are stored in a repository. An example
of a component model that emphasises connector reuse is described in [106].

KOALA and ProCom are examples of component models with repositories of
components. Connectors in these models are not intended to be reused, rather,
they are entities with code written to be used for specific sets of components.

In this thesis, we do not focus on repositories of software components and
connectors. Software design with repositories is an interesting subject, especially
in the context of the domain of software systems. We demonstrated the benefits
of the component model in the domain of embedded systems, and the generic
domain with the component-based domain model in Chapter 7 and Chapter 8
respectively. This serves as a source of further improvement for achieving software
reusability, using both components and connectors in the CBSD approach.

9.5 Architecture Expressiveness

The design of a component model with explicit control flow and data flow, integ-
rated but separated in a model, causes the software architecture to be expressive,
in the sense that the behavioural aspects of the system are highlighted. The
control flow part models the discrete behaviour of the system, whereas the data
flow part models the regular behaviour of the system.

A control connector forms the control flow structure for coordinating compon-
ents. This is realised by having various types of control connectors for connect-
ing components. Our control connectors are inspired by the X-MAN component
model. This model offers Turing completeness through sequencing, branching and
looping control structures for control connectors. Control connectors that form
the control flow structure are the explicit entities for composing software compo-
ents [63]. They are presented and discussed in various contexts in [74, 68, 67, 73].
Even though these connectors model the control flow, as in a von Neumann architecture, they do not separate the data flow. Data flows with control in the system. For example, the selector connector in X-MAN forms a branching structure which requires the presence of decision data to route the control flow. X-MAN selectors combine the control flow and the data flow, with the data flow following the control flow.

Control flow is clearly expressed using our control connectors. Compared to SCADE, we have the benefit of creating a more expressive architecture, as the control connectors provide the visualisation and constraints of the control flow. The execution of the control flow can be traced in the system. The basic control connectors proposed in this thesis contain the sequencing, branching and looping control structures that are Turing complete. In SCADE, the visualisation of control is in the form of state machines. The control output from the safe state machine block modelled by Esterel is treated as another flow of data, that carries a data condition to activate a Lustre block. ProCom has a set of control connectors to express the control flow, but they are not Turing complete. In PECOS, control flow is written in the scheduler every time a system is built. Control flow is not an entity that is made explicit, and therefore it is not expressed in the architecture.

A complex control structure can be constructed from these basic control flow connectors, which in turn create a composite control connector that behaves like a pattern. In the Software Engineering community, design patterns are defined as reusable solutions for commonly occurring problems [46]. An example of a pattern that can be created using our basic control connectors is a chain of responsibility pattern, which involves a sequencer connector. There exist patterns that are defined in the context of component models that express the behaviour of control flow which can form explicit entities for software reuse [68].

Data connectors in our model are defined to express the behaviour of data flow in the system. Similar to the control flow described, the expressiveness of data flows provide the visualisation for the flow of execution in the system. We have defined various types of data connectors, which can be manipulated to model different behaviours of data flows. This is different in SCADE, where the data lines that connect Lustre blocks are defined as a sequence of values. Each value represents the parameter value captured at a particular time. These values are manipulated or retrieved by associating the data parameter with the time.
9.6 Chapter Summary

In this chapter, we have evaluated the work in this thesis. A detailed analysis and evaluation of the work has highlighted its strengths and weaknesses. The main strength of the work is that it provides control flow and data flow modelling in a CBSD approach, demonstrating practical applications in the context of a domain. By contrast, the main weakness of the work is in the execution semantics of the component model, which to date is not based on a solid theoretical foundation. Consequently, the range of systems that can be modelled with our implementation is correspondingly limited.

Since the work in this thesis focuses on modelling the fundamental aspects of software systems — control, data and computation — the strengths outweigh the weaknesses. In the next chapter, we conclude this thesis and discuss possible future work suggested by our findings.
Chapter 10

Conclusions and Future Work

In this thesis, we have studied and proposed a component model that is both control-driven and data-driven. This chapter brings the thesis to a conclusion. Here, we summarise and conclude our research work, and give directions for further work.

10.1 Conclusions

The main goal of the work presented in this thesis was to advance the current state of the art in the design of a component model that is both control-driven and data-driven. Control, data and computation are the basic elements of any software system, and form the fundamental computation model for modelling software systems. It is therefore important to capture these elements explicitly in a software model.

In CBSD, the modelling of control, data and computation should be considered in the design of a component model. Not only do these elements need to be explicit in the model, but also they need to be separated from each other. Compared to the existing literature on control flow and data flow modelling, it is original and significant to investigate and motivate the desiderata for control, data and computation modelling in a component model. This was highlighted and presented in Chapter 3.

Based on these desirable features, a thorough examination of modelling the control, data and computation in the existing component models is made in Chapter 4. This is original work, which categorises and analyses the related work with respect to the desiderata. The survey presented in Chapter 4 clearly shows
that there is no existing work that can fully realise these objectives.

Thus, a new component model that is both control-driven and data-driven is proposed in Chapter 5. The modelling elements are defined to support both control-driven and data-driven computations. The component model is unique in this sense, since it is the only model that can model both control-driven and data-driven systems. The control connector defined for the component model is also unique, as it allows the control flow to be composed (via control connector composition) to form a complex control structure. Also, the connected control connectors that capture a specific control flow for a domain can be made into an entity that is identifiable for the purpose of reuse. This means that the composite control connector can be deposited in a repository for future reuse. Doing so brings further additions to the CBSD goal of maximising software reuse by reusing both components and connectors.

Based on the proposed component model, its execution semantics are defined in Chapter 6, with the intention of realising the component model. This involved managing both control flow and data flow, where leaving the two flows running without any synchronisation causes nondeterministic behaviour in the model. This leads to an implementation that employs a synchronous execution for both control flow and data flow. The implementation is working for systems that operate based on modes. It is deemed unsuitable for other type of systems in practice. The work of defining the execution semantics for the component model should be based on a solid theoretical foundation, requiring a thorough investigations of the topic of execution models and model formalisation.

The fact that the component model separates control flow and data flow modelling makes it well-suited for embedded systems modelling. The use of the component model in constructing an embedded system was illustrated in Chapter 7. An in-depth evaluation of the component model was presented, with a clear comparison to related work. The analysis derived from this evaluation provides important insights into the criteria of separating control flow and data flow modelling for the domain of embedded systems.

The practicals aspect of the component model are also tested for a domain modelling approach in CBSD. This is a novel approach to using a component model for performing domain modelling for domain analysis. The idea is to provide an alternative to the FODA domain modelling approach. The component-based domain model was demonstrated in Chapter 8, in which the modelling
counterparts of the FODA functional model are identified for the component-based domain model. The approach is demonstrated to be practically feasible, with promising future directions.

10.2 Future Work

We have established the foundations for control flow and data flow modelling in a component model. The work presented in this thesis offers a solid basis for further research, which is discussed below.

Component Model Extensions

New modelling elements can be added to the component model to enrich the modelling expressions for control flow and data flow. Possible additions of new entities include other data connectors (REO \[8\] provides a rich source of various channel semantics), data coordinators (various operators from data flow process networks \[80\], such as data selectors and delays) and control connectors (COBE-GIN for adding concurrency, and finite loops).

Another possible enhancement concerns the component definition. One possible extension would be to add state to components. A more challenging enhancement is introducing a notion of composite components. It is already possible to build composite components from data flow components and data connectors, by means of port delegation, as is the current practice in ADLs. However, defining composite components combining both control flow and data flow would be more involved and requires elaboration.

Execution Semantics

The execution semantics for the component model presented in Chapter 6 clearly shows that it is non-trivial research work to manage the separated control flow and data flow in a model. This invites further research in the area that involves formalising the component model, and extensive study of models of computation. A further development in this aspect of the model can increase the practicality of the component model with a solid theoretical foundation.
CHAPTER 10. CONCLUSIONS AND FUTURE WORK

Embedded Systems Modelling

The suitability of the component model for the embedded systems domain, shown in this thesis, should give rise to interesting research work for embedded systems. An embedded system is a hybrid system that exhibits continuous and discrete behaviour. Integrating the two different behaviours in the same model creates many research questions that need to be answered. Research in formal verification, building a mathematical framework for the model and model synthesis are examples of the possible research areas that can help answer those questions. Another possible future direction in this context concerns the tool development that can facilitate the modelling of embedded systems using the component model.

Domain Modelling

In this thesis, we demonstrated that it is feasible to perform domain modelling for domain analysis using the component-based domain model. This shows the important role of a component model in domain engineering with CBSD. The component-based domain model could be used as an alternative to the FODA domain model for performing domain analysis. Harnessing this potential, however, is a major challenge especially when dealing with the variabilities in a domain. There are two potential research directions with regards to this potential. Firstly, a reasonable extension to the component model is required to support the variabilities that exist for any domain model. Secondly, a design of software repositories for components and connectors is highly desirable to realise the practical aspect of software reuse with component-based domain models.
Bibliography


Appendix A

Constructing a Component-Based Domain Model for a Window Control System

All modern cars have a Window Control System to control the operation of car windows. The window control system controls and coordinates the operation of windows in the car, responding to inputs given by the car passengers. For simplicity, we model a window control system that is made for a car with two doors.

A.1 Problem Description

The control software operates the window pane by opening and closing it. The opening and closing movements are performed by a motor which is subject to the position of a control switch. There are three inputs from the user: up, down and stop. The window lifting movement has to be stopped automatically when the lower or upper bound is reached.

The comfort function in a car requires extra functionality for the basic window operation. This includes the ability for the driver to control the passenger window, protecting children by disabling the window movement for passenger door when the protection is activated, and integration with the central locking system that requires the window to be fully closed when the car is locked from outside.
A.2 A FODA Model for the Window Control System

A.2.1 The Feature Model

Based on the problem description, the corresponding features are described in the feature model shown in Figure A.1. To demonstrate the modelling of the window control system, we consider a window control system that performs all the features, except the integration with a central locking system.

![Feature Model](image)

Figure A.1: Feature model for the window control system domain.

A.2.2 The Context Model

Figure A.2 shows the window control system context diagram, in particular the external entities that the system has to interface to. These are described next.

**Driver keypad** This is a window switch button for the driver side window. It is a special keypad switch for the driver that can control the driver window and also the passenger window. The keypad contains a button allowing the driver to press up/down for both the driver and passenger windows. If neither of these buttons is pressed, the keypad regards this input as stop input.

**Passenger keypad** This is a window switch button for the passenger side window. The keypad contains a button for the passenger to press up/down for the passenger’s window only. If neither button is pressed, the keypad again regards this input as stop input.

**Driver window motor** This is the physical motor that operates the driver window going up or down. It interacts with the system by feeding the current
drain reading (for d-top limit resistance and d-bottom limit resistance), and receiving the d-window motor speed from the system to move the window accordingly.

**Passenger window motor** This is the physical motor that operates the passenger window going up or down. It interacts with the system by feeding the current drain reading (for p-top limit resistance and p-bottom limit resistance) and receiving the p-window motor speed from the system to move the window accordingly.

**Comfort function** This entity handles the integration and management of the comfort function feature in the car. This includes storing the comfort function setting, required for activating the child protection system to disable the window movement. The child protection feature for the window control system only involves the passenger window. Also, this entity stores the window status for both the driver window (d-window status) and the passenger window (p-window status) that it receives from the window control system.

**D-shaft** Each window in the car has a motor sensor which senses the actual physical motor behaviour. This includes the driver window where the shaft reading is performed by the so-called D-shaft. The d-shaft input is the shaft rotation reading.

**P-shaft** Similar to the D-shaft, except that this is for the passenger window.
A.2.3 The Functional Model

The window control system context model, shown in Figure A.2, is decomposed to lower level functions. Figure A.3 shows the first level of decomposition, where most of the data transformations involve operating the driver window. The behaviour of the driver controller CT is described by the STD.

The data transformation Move Passenger Window is the only data transformation that can be further decomposed. The rest of the DTs in Figure A.3 are primitive data transformations. Figure A.4 shows the second level of decomposition for the move passenger window data transformation. The behaviour for the passenger controller CT is described by the STD.

We replaced the data transformation Move Passenger Window with its internal model, to get an overview of the decomposition process that finally arrives at
Figure A.4: Window control system: second level decomposition.
primitive data transformations. Figure A.5 shows all primitive data transformations for the window control system.

### A.3 A Component-Based Domain Model for the Window Control System

We reconstruct the domain model in Figure A.5 using components and connectors in our model. The construction of the component-based domain model starts with the move passenger window function modelled in Figure A.4. Figure A.6 shows the component-based domain model equivalent to the move passenger window function modelled in Figure A.4.

Going bottom up in composing components, Figure A.7 shows the rest of the model which involved moving the driver window function. We labelled the move window passenger function as a dash line box in Figure A.7 for the reason of simplicity and clarity in reading the model. The whole model with the passenger and driver window operating functions is shown in Figure A.8.
Figure A.5: All primitive data transformations for the window control system.
APPENDIX A. CONSTR. A CBDM FOR A WINDOW CTL. SYS.

Figure A.6: Move Passenger Window in Component-based Domain Model
## A.3. A CBDM FOR THE WINDOW CONTROL SYSTEM

### Evaluation of Input for Disabling D-Window Up

- **f5**: Flag to forward disable d-window up (T/F)
- **c5**: Condition to disable d-window up (T/F)

### Evaluation of Input for Disabling D-Window Down

- **f6**: Flag to forward disable d-window down (T/F)
- **c6**: Condition to disable d-window down (T/F)

### Evaluation of Input for D-Window Movement

- **f4**: Flag to forward d-keypad input (T/F)
- **c4**: Condition to move up (1) or move down (2)

### Update D-Window Status

### Move D-Window Up

- **d-max reach**
- **d-window status**

### Move D-Window Down

- **d-min reach**
- **d-window status**

### Detecting Driver Input

- **d-shaft input**
- **d-top-limit resistance**
- **d-bottom-limit resistance**

### Move Passenger Window (2)

- **p-window motor speed**
- **p-window status**

### Forward Keypad Input

- **driver keypad input**
- **passenger keypad input**

### Limit Sensors

- **d-window max sensor**
- **d-window min sensor**

### Forward Sensor Input

- **d-window distance**
- **d-window motor speed**

### Additional Variables

- **f5**: Flag to forward disable d-window up (T/F)
- **c6**: Condition to disable d-window down (T/F)

---

**Figure A.7**: The rest of the functions in the component-based domain model.
APPENDIX A. CONSTR. A CBDM FOR A WINDOW CTL. SYS.

Figure A.8: The component-based domain model for the window control system.