Model-based Decentralised Automatic Management of Product Flow Paths in Processing Plants

From the Faculty of Georesources and Materials Engineering of the RWTH Aachen University

submitted by

Gustavo Arturo Quirós Araya, M.Sc. in Computer Science

from San José, Costa Rica

in respect of the academic degree of

Doctor of Natural Sciences

approved thesis

Advisors:  Univ.-Prof. Dr.-Ing. Ulrich Epple
          Univ.-Prof. Dr. Ir. Joost-Pieter Katoen

Date of the oral examination: May 3, 2010

This thesis is available in electronic format on the university library website.
This research has been partially funded by the DFG Research Training Group 1298 “Algorithmic synthesis of reactive and discrete-continuous systems” (AlgoSyn).
Model-based Decentralised Automatic Management of Product Flow Paths in Processing Plants
Quirós, Gustavo

**Model-based Decentralised Automatic Management of Product Flow Paths in Processing Plants**

220 Seiten, 46 Bilder, 3 Tabellen.
ISBN 978-3-18-518308-9, ISSN 0178-9546,
€ 65,00 / VDI-Mitgliederpreis € 58,50.

**Keywords:** Process control – Model-based software engineering – Object-oriented software engineering – Product flow paths – Decentralised systems – Distributed systems – Distributed algorithms – Distributed control – System synthesis – Safety

This work introduces the model of product flow paths as a formal framework for the correct execution of product flow operations in processing plants. A product flow path is a software object responsible for controlling, monitoring and documenting the movement of products along a determined route in the plant, and represents a temporarily and spatially isolated area for the safe and correct transport of products. A formal model provides a guide for the implementation of decentralised object-oriented product flow path management systems. A model-based synthesis technique replaces the engineering of a flow path management system for a plant with the simpler task of creating and verifying a model of the plant. A prototypical implementation of this technology has been developed as a proof of concept and tested with real-life plants, and can be seen as a reference implementation for industrial-strength flow path management systems.

**Bibliographische Information der Deutschen Bibliothek**
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliographie; detaillierte bibliographische Daten sind im Internet unter http://dnb.ddb.de abrufbar.

**Bibliographic information published by the Deutsche Bibliothek**
(German National Library)
The Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliographie (German National Bibliography); detailed bibliographic data is available via Internet at http://dnb.ddb.de.
Preface

This thesis is the result of the research that I conducted at the Chair of Process Control Engineering of RWTH Aachen University during the period from 2006 to 2009 under the guidance of Prof. Dr.-Ing. Ulrich Epple, and while being a member and scholarship holder of the DFG Research Training Group 1298 “Algorithmic synthesis of reactive and discrete-continuous systems” (AlgoSyn) under the direction of Prof. Dr. Dr.h.c. Wolfgang Thomas. Furthermore, Prof. Dr. Ir. Joost-Pieter Katoen acted as a second advisor for this work.

I am immensely grateful to Prof. Epple for his mentoring, his encouragement and his support during my time as a doctoral student and research assistant at his chair, for having offered the initial motivation for this research, for his precise insight into the formal principles and practical aspects of the field of process control engineering, and for being committed to offer an amenable working environment that encourages close collaboration and the sharing of knowledge. I also thank Prof. Katoen sincerely for accepting to act as a second advisor, for his careful revision of this work and for his keen indications. Furthermore, I am very grateful to Prof. Thomas for his commitment as a director of AlgoSyn, and to Prof. Dr.-Ing. Wolfgang Bleck for presiding over my doctoral examination. Finally, I sincerely thank Ursula Bey for her invaluable assistance.

Many people have contributed to the research that is presented in this thesis with their comments and thoughts. For this I thank Prof. Dr. Alexander Fay, Christina Haus, Holger Jeromin, Reiner Jorewitz, Dr. Łukasz Kaiser, Dr. Sebastian Klabes, Tina Kraußer, Kai Krün- ing, Henning Mersch, Martin Mertens, Dr. George Mertzios, Prof. Dr. Martin Polke, Markus Schlüter, Bernd Schmidt, Dr. Stefan Schmitz, Liyong Yu, and all the members of AlgoSyn. I also thank Ursula Bey, Johannes Gregor, Milena Gueorguieva, David Kampert, Tobias Li- etke, Adela Marsikova, Martina Uecker and Khai Vinh Vi for their collaboration at the Chair of Process Control Engineering.

I am ever so grateful to my parents Arturo and María, my sister Alicia, my brother Roberto, my parents-in-law Edgar and Sonia, and my brother-in-law Edgar for their love and their unconditional support in every possible manner. I also thank my family and my friends for their encouragement and support. Finally, I am infinitely grateful to Catalina, for walking the path of life with me.

Gustavo Quirós
Aachen, September 2010
“You cannot travel the path until you have become the path itself.”

Prince Siddhartha Gautama, the founder of Buddhism (563–483 B.C.)
# Contents

Abstract VIII

Kurzfassung XI

1 Introduction 1
   1.1 Motivation 1
      1.1.1 Piping Systems 2
      1.1.2 Example: A Filling Station 4
      1.1.3 The Problem 5
   1.2 An Analogy from Railway Control 6
   1.3 The Concept of Product Flow Paths 8
   1.4 Goals of this Work 9
   1.5 Structure of this Work 9

2 Definition of Requirements and Tasks 10
   2.1 Product Flow Path Model 10
      2.1.1 Composition of Product Flow Paths 10
      2.1.2 Life Cycle of Product Flow Paths 11
   2.2 Product Flow Path Management 13
      2.2.1 Product Flow Path Management Tasks 13
      2.2.2 Product Flow Paths as Objects 14
      2.2.3 Product Flow Path Management Object Model 16
      2.2.4 Assignment of Product Flow Path Management Tasks 19
   2.3 Decentralised Product Flow Path Management 20
      2.3.1 Advantages of Decentralisation 20
      2.3.2 Decentralised Railway Control 21
      2.3.3 Decentralised Product Flow Path Management Systems 22
   2.4 Synthesis of Product Flow Path Management Systems 24

3 The State of the Art 25
   3.1 Process Control Engineering 25
      3.1.1 Engineering of Plants and Control Systems 26
      3.1.2 Continuous Processes and Batch Processes 29
      3.1.3 Decentralised Control Systems 30
      3.1.4 Control of Material Transport 30
   3.2 Computer Science 34
      3.2.1 Fundamental Theories 34
      3.2.2 Decentralised Systems 39
      3.2.3 Object-oriented Software Engineering 39
   3.3 Railway Control 40
3.3.1 Geographical Railway Control .......................... 41

4 Formal Models .................................................. 43
4.1 Modelling Approach ........................................... 43
4.2 Abstract Plant Model .......................................... 45
4.3 Flow Allowance .................................................. 48
  4.3.1 Boolean Flow Allowance ................................. 49
  4.3.2 Flow Allowance Settings ................................. 50
  4.3.3 Flow Allowance Model ..................................... 53
  4.3.4 Flow Allowance States ..................................... 53
4.4 Flow Routes .................................................... 54
  4.4.1 The Structure of Flow Routes ............................. 54
  4.4.2 Flow Steps .................................................. 55
  4.4.3 Flow Routes ................................................ 56
  4.4.4 Open Flow Routes ......................................... 59
4.5 Enclosed Flow Routes ......................................... 60
  4.5.1 Product Leaks and Product Mixtures .................... 60

5 Algorithms ........................................................ 66
5.1 Decentralised Component Model ............................... 66
5.2 Decentralised Algorithms ..................................... 67
  5.2.1 Syntax of Algorithms ...................................... 68
  5.2.2 Component Environment .................................... 69
  5.2.3 Flow Path Analysis ........................................ 70
  5.2.4 Flow Path Monitoring ...................................... 76
  5.2.5 Flow Path Allocation ...................................... 86
  5.2.6 Examples of the Algorithms .............................. 111
  5.2.7 Termination Properties .................................... 114
5.3 Model-based Synthesis ......................................... 116
  5.3.1 Synthesis Environment ..................................... 116
  5.3.2 Component Synthesis ...................................... 117
5.4 Computational Complexity ..................................... 118
  5.4.1 Complexity of Flow Path Analysis ...................... 119
  5.4.2 Complexity of Flow Path Monitoring .................... 120
  5.4.3 Complexity of Flow Path Allocation .................... 121
  5.4.4 Complexity of Component Synthesis .................... 121

6 Analysis of Resources and Tools ............................... 123
6.1 Programming Languages ....................................... 123
  6.1.1 C .......................................................... 123
  6.1.2 Tcl ........................................................ 124
6.2 ACPLT Technologies ............................................. 124
  6.2.1 ACPLT/KS Communication System ....................... 124
  6.2.2 ACPLT/OV Object Management System .................. 125
  6.2.3 iFBSpro Function Block System ......................... 126
  6.2.4 fbkslib ................................................... 126
  6.2.5 Tks ....................................................... 126
### Contents

6.3 Formal Plant Meta-models .................................................. 127
  6.3.1 CAEX ............................................................... 127
  6.3.2 RIVA .............................................................. 128

7 A Prototypical Implementation ............................................ 130
  7.1 System Architecture .................................................... 130
    7.1.1 Class Libraries ................................................. 130
    7.1.2 Instance Model ................................................. 143
    7.1.3 Execution Model ............................................... 147
  7.2 System Synthesis ...................................................... 151
    7.2.1 Synthesis Program ............................................. 152
    7.2.2 Parameter Files ............................................... 153

8 Case Studies ............................................................... 156
  8.1 Petrochemical Plant ..................................................... 156
    8.1.1 Description of the Plant ..................................... 156
    8.1.2 Plant Model ................................................... 156
    8.1.3 System Synthesis .............................................. 157
    8.1.4 Operation of the Flow Path Management System ............ 158
  8.2 Pumping Station ....................................................... 158
    8.2.1 Description of the Plant ..................................... 158
    8.2.2 Plant Model ................................................... 159
    8.2.3 System Synthesis .............................................. 162
    8.2.4 Access to Online Plant Data ................................ 163
    8.2.5 Tests .......................................................... 165
    8.2.6 Distributed Flow Path Management System .................. 172
  8.3 Performance Analysis .................................................. 172
    8.3.1 Space .......................................................... 173
    8.3.2 Execution Times .............................................. 174

9 Conclusions ....................................................................... 177
  9.1 Summary ................................................................. 177
  9.2 Discussion .................................................................... 179
    9.2.1 Methodology ..................................................... 179
    9.2.2 Application Domain ............................................. 181
    9.2.3 Related Work .................................................... 182
  9.3 Future Research Directions ............................................ 182

A ACPLT/OVM Models ......................................................... 185
  A.1 Decentralisation.ovm ................................................. 185
  A.2 FlowAllowance.ovm .................................................... 188
  A.3 FlowPathAnalysis.ovm ............................................... 190
  A.4 FlowPathMonitoring.ovm ............................................. 191
  A.5 FlowPathAllocation.ovm ............................................. 193
  A.6 FlowPathManagement.ovm .......................................... 196

Bibliography ....................................................................... 203
Abstract

The present work introduces a new concept for conducting product flow operations in processing plants: the model of product flow paths.

As plants become more complex and more widespread, the engineering tasks required for designing and implementing control functionality become increasingly complex, costly, and harder to verify. The task of engineering the control of product transport operations in processing plants suffers from this problem as it is commonly accomplished manually, based on experience and common sense, and without the use of a model that guides the design. This incurs in greater engineering costs, and in a greater chance of introducing errors in the control logic.

The model of product flow paths is introduced as a novel attempt to provide a formal framework for the correct execution of product flow operations in processing plants. This model is centred around the idea of a product flow path, which is a software object that is responsible for controlling, monitoring and documenting the movement of products along a determined flow route in the plant, and which represents a temporarily and spatially isolated area for the safe and correct transport of products along this flow route. Product flow paths have a well-defined life cycle that characterises their operation and aims to guarantee their safe and correct use, and this is inspired by the common procedures used by railway systems for achieving the safe movement of trains along the tracks of a railway network. The management of product flow paths comprises the automated tasks that are necessary for the adoption of the product flow path model within a process control system. It includes the discovery of flow routes in a plant, the creation and deletion of product flow path objects, the allocation, locking and activation of product flow paths, the monitoring of active product flow paths, the issuing of flow path alerts to report problematic situations and the documentation of the complete life cycle of every product flow path in the system. An object model for a product flow path management system is defined, which provides a guide for the implementation of object-oriented product flow path management systems.

The automation of product flow path management tasks requires precise information about the possibility of product flow in the plant, and this is provided by means of a formal model that represents the structure of the plant and the flow allowance of the elements in the plant. Based on this model, a formal definition of the concept of a product flow route in a plant is given, considering the possible displacement of material through the structure of the plant. Also, the properties of openness and enclosure of a flow route at a given flow allowance state of the plant are formulated in a way which may be used by algorithms to detect potentially hazardous situations such as product leaks and product mixtures, or to determine the absence of these situations for a given flow route. This formal framework conforms the base of an algorithmic solution for the automation of product flow path management tasks.

Modern process control systems operate in a decentralised manner because this offers many advantages over a fully centralised deployment of the control functionality. In accordance with this principle, this work presents a decentralisation scheme that may be applied to the tasks involved in the management of product flow paths in a plant. Inspired by the
decentralised operation of geographical railway control systems, a model for designing and implementing decentralised product flow path management systems has been developed by following a decentralised component-based approach, where the structure of the decentralised system follows the structure of the corresponding plant. The objects of a decentralised product flow path management system may be distributed across multiple controllers in a process control system, and in a way that follows the logical sectioning of the plant.

Based on the formal model of flow routes, and on the decentralised component-based approach that is followed in this work, algorithms are presented which accomplish the tasks of a product flow path management system in a decentralised manner. The algorithms consist of procedures that execute within the different decentralised components of the system, which communicate with each other by exchanging messages that are delivered along the interconnections of the components. Flow path analysis is an algorithm that discovers flow routes in a plant by performing a decentralised breadth-first search over the graph of the plant. Flow path monitoring is an algorithm that determines the open condition of a flow route, which additionally detects and reports potential product leaks and product mixtures that affect the flow route that is being monitored. Flow path allocation is an algorithm that assigns plant elements to a product flow path in an exclusive manner, and which also constrains the settings of some plant elements in order to guarantee the enclosure property of the flow path, and therefore its safe use. The objects of a product flow path management system make use of these algorithms in order to provide a unified access to the functionalities of discovering, monitoring and allocating product flow paths in a plant.

In order to provide a way of reducing the complexity and the cost of constructing a decentralised product flow path management system, and at the same time, a way of reducing the chance of introducing errors in the system, an automatic synthesis approach for this kind of systems is presented, which is based on a formal model of the structure of the plant and on the flow allowance model of the elements in the plant. By accessing the information in these models, a synthesis algorithm automatically creates and parametrises the objects of a product flow path management system, which is then ready to begin its operation. This model-based synthesis technique is advantageous because it replaces the task of constructing a flow path management system for a plant – which is relatively complex and error-prone – with the simpler task of creating and verifying a model of the plant.

A prototypical implementation of the technology presented in this work has been developed as a proof of concept. A decentralised product flow path management system is realised as a collection of function block servers. These servers host function blocks that correspond to the flow path objects, the decentralised components, the algorithm objects and the rest of the objects of this system. Furthermore, the decentralised components communicate with each other and across the servers of the system. The automatic synthesis approach described in this work is implemented for this prototype by means of a synthesis program that explores a model of the plant and based on the information contained therein, it creates and parametrises the objects of a decentralised product flow path management system for the modelled plant.

The prototypical product flow path management system has been tested with the models of real-life plants. For these plants, a flow path management system was automatically constructed based on a corresponding model of the plant’s structure, and on a description of the flow allowance model of the plant’s elements. The decentralised algorithms which perform the discovery, monitoring and allocation of product flow paths were tested with these systems, as well as the usage of the product flow path object model according to the life cycle of product
Abstract

flow paths that is presented in this work. Also, the flow path management system was connected to a process control system in order to allow the decentralised algorithms to operate with online plant data. The case studies that are presented in this work show the feasibility of the product flow path management technology. Concretely, the model-based synthesis approach, the decentralisation of the product flow path management operations and the usage of the product flow path object model have all been applied in a successful manner in these cases. Because of this, this prototype can be seen as a reference implementation for industrial-strength flow path management systems.
Kurzfassung


Mit dem Produktflusswegmodell werden mehrere Ziele verfolgt: die automatische Erkennung von Flussrouten in einer Anlage, die Belegung und die Sicherung von Flusswegen ähnlich wie aus dem Eisenbahnbetrieb bekannt, die Überwachung von Produktleckagen und von unerwünschten Produktmischungen, die Alarmierung von potenziell gefährlichen Situationen, die automatische Dokumentation des Lebenszyklus aller Flusswege sowie die Dokumentation aller Transportvorgänge und die Kontextbereitstellung für intelligente und adaptierende Überwachungsfunktionen. Die Verwaltung von Produktflusswegen entspricht den automatischen Aufgaben, die für die Nutzung des Produktflusswegmodells in Prozessleitsystemen notwendig sind.


Das vorgestellte Dezentralisierungskonzept verfolgt einen komponentenbasierten Ansatz, wobei die Struktur des dezentralen Systems die Struktur der Anlage abbildet. Die Objekte des dezentralen Systems können über mehrere Steuerungseinheiten und entsprechend der
Organisationsstruktur der Anlage verteilt werden. Die dezentrale Flusswegverwaltung bietet eine technische Lösung für eine robuste, flexible und selbstskalierende Realisierung.


1 Introduction

Processing plants are fundamental for the way we live. Most of the food we eat, the clothes we wear, and the items that we use every day for work, travel, communication and amusement, owe their existence directly or indirectly to products that come from processing plants. The last century has seen such an advance in industrialisation, that it is difficult to imagine a normal life without the plethora of modern-day objects that this phenomenon has generated.

As processing plants become more widespread, they also become larger and more complex, and they are able to host more complex processes and to yield a greater variety of products. This, in addition to the permanent need to increase the efficiency, quality and safety of the production, has pushed processing plants to become more and more automated. This has given birth to the field known as process control engineering [17, 45], which is a hybrid of chemical, mechanical, electrical, computer and software engineering that studies the development of systems that control the operation of processing plants. The current work presents a new theoretical and technological approach which has the potential to improve some of the functionalities currently offered by process control systems, namely, those that deal with controlling, securing, monitoring and tracking the flow of products in processing plants. This chapter serves as an introduction to the topic, it gives an overview of the main technological issues which this work aims to solve, and presents the goals which will be pursued throughout the following chapters. A motivating example is discussed, which should help the reader in grasping the main intentions of this work.

1.1 Motivation

Processing plants are technical facilities that are designed and built with the purpose of operating one or more determined processes. These processes consist of procedures by which materials called products are produced from other materials called raw materials by applying one or more physical or chemical transformations to them. For this, processing plants are equipped with devices that perform these transformations such as heaters, grinders, mixers, reactors and distillers. In addition, the operation of processes requires the movement of materials through the plant, as well as the storage of both raw materials and products. For this, processing plants usually comprise devices like pipes, pumps, compressors, conveyors, tanks and silos. Finally, the movement of these materials must usually occur in different ways at different times, and devices like valves are commonly employed to enforce this.

The different aspects of the operation of a process within a plant must all be managed by a process control system. The transformation of materials must be controlled by, say, regulating the temperature of a heater or the pressure in a reactor. Also, the movement of material must be controlled by performing tasks such as activating and deactivating pumps, monitoring the levels in tanks, or regulating the aperture of control valves. Throughout the years, several methodologies have been developed with the purpose of providing solutions for these engineering tasks. For instance, the mathematics and engineering field of control
1 Introduction

theory has yielded a good amount of theoretical results which may be used in practise by control engineering to design techniques for controlling the dynamic behaviour of processes [6, 18]. However, many of the techniques used in process control engineering are not based on a well-defined theory, but rather on years of experience. The techniques used at present for controlling the movement of products in a plant mostly fall into this category. This is due in part to the fact that these movement operations are simple to understand and to carry out, when compared with the more complex dynamic control tasks that spawned the development of control theory. Whereas regulating the different parameters of a chemical reaction requires the analysis of a complex dynamic system, performing a pumping operation that fills a tank to its top may basically be described as “setting valves to correct positions, activating the pump, waiting until the tank is full, and then deactivating the pump”. Simple control sequences like this one may be easily designed by a process control engineer, and thus, engineers have been accustomed to developing such solutions by hand, without the support of a formal theory other than the knowledge about the plant in question, and common sense.

The main disadvantage of this manual engineering approach is that it does not scale well with the size and complexity of the plant. Effectively, designing proper control sequences for large and complex plants can prove to be a daunting task that must be handled by teams of engineers with an adequate amount of experience. Even then, the correctness of these designs is difficult to assert. Also, if any changes are made to the plant, then the entire design of the control logic must be revised and updated, incurring in more engineering effort.

It is precisely this point which motivates the matter of the present work. Just as control theory and control engineering have enabled the steering of complex dynamic processes by providing a model of these processes and their controllers and a methodology that engineers can apply in order to solve these control problems, a model for the transport of material in processing plants would equally ease the tasks of designing and implementing the components of process control systems that automate the movement of products in a plant. The present work introduces the model of product flow paths as an novel reference model for performing product transport operations in a correct and safe manner within processing plants. Thereby, the management of product flow paths is defined as a collection of tasks which enable process control systems to embrace the model of product flow paths and apply it to its operations, and the use of a formal model further permits the automation of many of the engineering tasks that are necessary for this.

1.1.1 Piping Systems

The kinds of processing plants that are directly considered in this work are those which consist mainly of piping systems as shown in Figure 1.1. Thus, the kinds of products that will be regarded are mainly fluids, that is, those materials that are able to move through these systems by flowing [20]. This covers a large portion of all processing plants and their products, for instance in the chemical, petrochemical, pharmaceutical and food and drink industries, and even covers other kinds of installations where piping systems are used outside of processing plants. Moreover, the model of product flow paths that is presented in this work is designed to be very general, and because of this, some of its ideas may be directly applied to other forms of product movement, such as solids travelling along conveyors.
1.1 Motivation

Figure 1.1: Examples of piping systems.
1.1.2 Example: A Filling Station

There are many details that must be considered when controlling the flow of products in a plant. In order to show this, we will now examine an example of a simple plant that, in spite of its size, presents some of the same problems that engineers must face when designing control systems for large and complex plants.

Our sample plant is a simple filling station that serves for the storage of products; its diagram is shown in Figure 1.2. Diagrams similar to this one are used when designing processing plants, and the symbols that are used therein represent the different types of plant devices that make up the plant and the way in which these devices are connected to each other to permit the flow of products. Furthermore, the plant devices are labelled with names that identify them. This filling station consists of four tanks, two pumps, seven valves, four pipe joins, and multiple pipe segments that interconnect all of these elements. The pipe joins J₃ and J₄ on both sides of the plant represent two product input nozzles that may be used for feeding product into the filling station. The pumps are depicted with internal lines that form an angle, and this angle shows the direction of the product flow that the pumps produce. The valves allow or disallow the flow of products through them by opening and closing, and the tanks store the material that flows into them. Finally, the pipe joins J₁ and J₂ are distribution points that allow products to flow from any direction to any other direction.

Thanks to the flexible design of this plant, each of the four tanks may be filled from any of the two product inputs J₃ and J₄ by using the corresponding pump P₁ and P₂. Flexible piping structures such as this one are not uncommon in modern processing plants. For instance, they are used in the petrochemical industry where large quantities of oil derivatives must be stored in tank farms, or in the food and drink industry where different kinds of products are produced at different times, and both these products and their corresponding raw materials
must be stored and retrieved in a flexible way. However, great care must be taken when performing a filling operation in plants such as this one, because undesired situations could otherwise occur. For instance, two different products could mix unexpectedly somewhere in the piping, or in one of the tanks. Depending on the nature of these products, the outcome of this situation could be product loss – which incurs in a loss of revenue – or even a hazardous situation where the products react causing explosions, fire, or escape of liquids or gases – which apart from product loss can also cause equipment damage, injuries or deaths of workers, and environmental damage. Other problematic situations that may occur are when a product flows into the wrong tank, or when a product flows through an unblocked piping section and leaks out of the plant. In all of these cases, the cause of the problem is the incorrect flow of products, which essentially means that a product is able to flow through a part of the plant where it is not supposed to flow.

Consider the task of filling tank \( T_3 \) with product that enters the filling station through the product input \( J_3 \). For this, the product would need to enter the plant through \( J_3 \), flow through the elements \( P_1, V_5, J_1, V_6, J_2 \) and \( V_3 \), and finally reach tank \( T_3 \). In order to accomplish this, one would need to activate pump \( P_1 \) to draw material from \( J_3 \) and feed it through the plant. Also, in order to allow the product to reach tank \( T_3 \), the valves \( V_5, V_6 \) and \( V_3 \) must be opened. However, it is clear that this is not enough, because the product could additionally reach other parts of the plant which are not meant to be reached. First, the tanks \( T_1, T_2 \) and \( T_4 \) may be reached by diverging product flow, and therefore, the valves \( V_1, V_2 \) and \( V_4 \) must be closed during the filling operation to avoid this. Second, pump \( P_2 \) could be reached, and valve \( V_7 \) must therefore be kept closed as well. Finally, the flow of the product from \( J_3 \) to \( T_3 \) should be protected from the incoming flow of any other products which may reach this product and mix with it. In this example, the closing of the valves \( V_1, V_2, V_4 \) and \( V_7 \) additionally assumes this responsibility. Therefore, the correct settings of the valves is a requirement for enforcing the correct flow of the product.

Going further, we may analyse the possibility of executing multiple filling operations at once, which would permit a more efficient use of the plant. When filling tank \( T_3 \) from the product input \( J_3 \) as described above, valve \( V_7 \) must remain closed, and this implies that pump \( P_2 \) may not be used during this time. That is, the restriction on the setting of the valve \( V_7 \) on the part of the filling operation conflicts with the use of pump \( P_2 \). However, this filling station is capable of performing some filling operations from \( J_3 \) and \( J_4 \) in parallel by activating the pumps \( P_1 \) and \( P_2 \) provided that valve \( V_6 \) is closed. In this case, the restriction on the setting of valve \( V_6 \) is a mutual agreement of the filling operations that perform on both sides of the valve.

### 1.1.3 The Problem

Process control engineers usually perform analyses such as the one that was presented for our sample filling station, and use them to design the control logic of the plant in a task commonly known as the engineering of the control system. When using this method to automate operations in the plant, every single detail which is relevant for an operation must be considered, and special care must be taken to cover all interactions between the different operations. For instance, an operation of pumping a product from \( J_3 \) to \( T_3 \) in our sample plant requires valve \( V_7 \) to be closed, but the engineer that designs a corresponding control sequence may “forget” to add the closing of this valve in the sequence. If the operation is
executed and the valve is open, then a problem will certainly arise. To make matters worse, if the operation is usually executed when the valve happens to be closed, then this error could remain undetected for a long time. So even the control systems of plants that have been in operation for some time could contain errors like this one.

As seen in our sample plant, there are many additional operation details that should be considered when designing a corresponding control logic. For instance, a second operation that requires valve \( V_7 \) to be open cannot execute while the first operation is active because of the conflict between them, and this rule must be enforced by the control system as well. Also, the control logic of the first operation should prohibit any attempt to open valve \( V_7 \) during the execution of the operation, whether this attempt comes from an action of the control system or directly from a human operator. For the simple plant of Figure 1.2, engineering a complete control logic that is also correct might not be a difficult task, but as plants become larger, this difficulty also increases. As mentioned earlier, if the plant is modified, the entire control logic must be revised, which also increases the engineering effort.

With this reasoning, we come to the main problem that this work addresses, which is stated by the following claim:

the method of manually engineering the control of product flow operations in processing plants, without the guide of a theory, increases the cost of the engineering phase and the probability of errors in the control system as the size and complexity of the plants increase.

The engineering cost is increased by this method in several ways. A greater engineering task requires more effort, and thus, more engineer-months. Also, for large and complex plants, more experienced engineers are needed. Furthermore, complex control designs require more rounds of validation to detect errors. In spite of this, the chance of leaving an error somewhere in the design increases as the complexity of the plant increases. The absence of a theory that guides the design of control logic for product flow in processing plants augments this issue, as it makes engineers resort to their experience and their common sense because of the lack of known systematic approaches. Finally, this problem is currently inevitable, because processing plants exhibit the tendency of becoming larger, more complex, and more automated as time goes on.

1.2 An Analogy from Railway Control

When facing a problem for the first time, it usually helps to examine similar problems and their solutions, sometimes in other areas of knowledge, in order to gain an insight into possible approaches that can be taken. The movement of product through a plant is hereby seen as a form of transportation, and therefore, it seems natural to study other forms of transportation together with their problems and solutions. Rail transport [43] is similar in many ways to the transport of products in a plant, as the following points show:

1. Railway networks are transportation systems with a *fixed structure* that defines the ways in which trains may move between locations across the system. The same holds for piping systems in processing plants, which also have fixed structures that define where and how products are able to move through it.
2. Trains can propel themselves along the tracks but are not able to steer their direction. Rather, it is the track layout and the settings of switchable elements called points, turnouts or switches, that determine the direction of a moving train. Likewise, the movement of a product in a plant may be produced by the action of a device such as a pump, but the direction of this flow is not determined by the product or by the pump, but rather by the structure of the plant and the settings of switchable limiting devices such as valves.

3. A railway system is commonly used by multiple trains simultaneously, which brings the need to adopt measures to avoid collisions between trains. This is normally accomplished by a system that administrates the usage of track segments. By requiring all trains to receive an authorisation before moving along a track – usually in the form of a visible signal, the system guarantees that every track segment is occupied by at most one train at a time, and thereby avoids the danger of collisions. Regarding processing plants, a similar goal is attained by a control system that avoids the undesired mixture of two products anywhere in the plant.

4. Trains travel along tracks and over movable points that are used to deviate the train in a desired direction. If a point changes its position as the train moves over it, the train faces the risk of derailing. Therefore, it is commonly required that every point be locked into position while a train moves over it, and cannot be unlocked until the train has moved away. In this manner, this cause of derailment is avoided. Similarly, if an important limiting valve is opened while a product flows along a plant, or if a valve is closed while the product flows through it, then some of the potentially hazardous situations that were discussed earlier could arise. Therefore, it is also important to lock any relevant switchable elements into position before activating a product transport operation.

As there are many similarities between railway transport and product transport in processing plants, it is reasonable to try to adopt some of the techniques that are used by railway control systems in order to guarantee the correct and safe movement of products in a plant. In doing this, we additionally benefit from more than one hundred years of experience in the field of railway operations.

In order to prevent train accidents – which mainly consist of collisions and derailments – a railway control system makes use of a signalling system and an interlocking system. The former indicates to trains their authorisation to enter track segments, and the latter controls the positions of points and the locks on these positions. Whenever a train intends to use a given track segment, the following sequence of steps is followed:

1. identify the intended train path, as well as the neighbouring track segments that connect to this path,

2. clear the path for the train, that is, reserve the path for exclusive use by the train and set all relevant points to the correct position,

3. protect the train path, that is, block any oncoming train traffic from neighbouring tracks by setting points so that approaching trains are deviated, and by issuing stop signals on these tracks,
4. lock the train path, that is, use the interlocking system to lock all relevant points and protecting signals in place,

5. use the train path, that is, signal the train to begin its travel, and finally,

6. unlock the train path by waiting for the train to leave the train path and then releasing all the locks that were set.

In this way, collisions are avoided by disallowing a second train to enter the train path while it is in use – both logically through signalling and physically by means of point settings, and derailments are avoided by locking all points along the path in place, thus keeping the points from changing their positions while the train travels over them.

Altogether, the operation of a railway system must guarantee the correctness and safety of the movement of trains through the railway network, and this is accomplished in an effective manner by adopting a series of measures and a well-defined procedure for performing these transport operations. Given the similarities between railway systems and processing plants, we are given the opportunity to apply some of these same solutions in order to provide a model that guarantees the correct and safe transport of products in a plant.

1.3 The Concept of Product Flow Paths

With the purpose of tackling the problems that are carried by the commonly used method of manually engineering the control of product flow operations in processing plants without the support of a theoretical model, this work introduces the concept of product flow paths.

A product flow path is a functional unit that serves to transport products correctly in a plant, similarly to how the transit of trains along a train path is organised in a railway system. A product flow path is composed of a collection of plant devices that participate in the transport operation, and a control logic that manages the requirements of this operation. In this manner, a product flow path represents a temporal and spatial isolated region for the transport of products in a plant.

Product flow paths are defined by a rigorous formal model that provides a theory on the correct transport of material in a plant, and which is inspired by many of the ideas that are employed in railway systems. For every transport operation, a product flow path is chosen and used by following a well-defined procedure, similarly to how a train path is used in a railway system to allow a train to travel through the railway network.

By encapsulating all of the relevant details of a product transport operation within the unit of a product flow path, the complexity of the corresponding control logic is reduced and the interactions between the different product transport operations in the plant are made clear. Also, by relying on a formal model, it becomes possible to apply a systematic engineering approach, and many of the engineering tasks that regard product flow paths may be therefore accomplished in an automatic manner. In this way, the usage of product flow paths within a process control system has the potential to reduce the cost of engineering the system and the probability of introducing errors in the system.
1.4 Goals of this Work

The main goal of this work is to contribute a model of product flow paths that can be used in processing plants to achieve the correct transport of products, while enabling the automation of many of the corresponding engineering tasks, thereby reducing the associated engineering cost and the probability of errors. This goal comprises several objectives which are outlined as follows.

- To provide a precise definition of the concept of product flow paths in processing plants by means of a formal model, where product flow paths are functional units that perform the correct and safe transport of products in a plant.

- To define the management of product flow paths as a collection of tasks that must be accomplished in order to apply the model of product flow paths within a process control system. Also, to provide algorithmic solutions for these tasks.

- To develop a method of automating the task of engineering a product flow path management system based on the model of product flow paths, thereby providing a cost-effective and high-quality technique for designing this system for a given plant.

- To show the feasibility of this approach based on product flow paths by means of a prototypical implementation and an evaluation of its use in case studies.

1.5 Structure of this Work

The present work is structured as follows: Chapter 2 gives a detailed analysis of the requirements of a product flow path model and a corresponding product flow path management system, and also outlines the different tasks that this work carries out in order to fulfil these requirements; Chapter 3 gives an overview of the current state of the art in the fields of process control engineering and computer science, while focusing on the aspects of these areas which are relevant for this work; Chapter 4 presents a formal framework for defining the constitution and properties of product flow paths in a precise manner, which serves as a guideline for developing technological solutions that are based on the concept of product flow paths; Chapter 5 details the algorithms that are presented in this work as means of performing the required tasks of a product flow path management system, as well as the task of constructing such a system in an automatic way; Chapter 6 summarises all resources and tools which were used to develop the prototypical implementation of the product flow path management technology that is presented in detail in Chapter 7; Chapter 8 presents the results of two case studies where the approach of product flow path management was tested; and finally, Chapter 9 presents the conclusions of this work, closing remarks, and an insight into future research directions.
2 Definition of Requirements and Tasks

The concept of product flow paths offers a new approach for performing product transport operations in processing plants. At the heart of this approach is the notion of a product flow path, which is a functional unit that is responsible for every aspect of a product flow operation. There are several tasks which are necessary for the utilisation of product flow paths in processing plants, and they are collectively handled by a product flow path management system. This chapter presents the fundamental constitution of a product flow path management system, together with its functional requirements. Also, two important design techniques for product flow path management systems – which are followed in this work – are introduced and explained: the decentralisation of the product flow path management tasks, and the automatic synthesis of a product flow path management system.

At present, practically all process control systems are, in one form or another, software-based [17, 45]. Consequently, the technology that is presented in this work is software technology, and consists of data models and algorithms. The intended execution environment is, nevertheless, not an arbitrary computer system, but rather a process control system.

2.1 Product Flow Path Model

Product flow paths are the central concept of the technological approach that is presented in this work, and this section describes them in a precise manner. This is accomplished by presenting a model of product flow paths that describes these entities both structurally and functionally. The analogy of railway systems that was presented in Chapter 1 serves as an inspiration for many of the details of the design of this model.

2.1.1 Composition of Product Flow Paths

Consider again the sample filling station in Figure 1.2, and further consider the task of moving material from the product input $J_3$ to the tank $T_3$. There are several plant elements that are involved in this operation in different ways, and others which are not involved. Therefore, a product flow path denotes a substructure of the structure of the plant. This substructure is defined by the intended path of flow, which contains every plant element that is passed by the product as it flows. In this example, the intended path of flow contains the elements $J_3, P_1, V_5, J_1, V_6, J_2, V_3$ and $T_3$, plus the pipe segments that connect these elements. Such a path of flow through the plant is called a product flow route, and is represented by a sequence of interconnected plant elements that may be used by a product to flow through the plant. A product flow route is essential for the existence of a product flow path, as it defines the actual course of the flow of products. Based on the location of the corresponding product flow route within the plant, a product flow path is able to control all the aspects of the transport operation along the flow route. Chapter 4 will present a formal definition of product flow routes in a plant.
Aside from the plant elements that make up the flow route of a product flow path, there are additional plant elements that are important for the operation of the product flow path, and that are logically contained by it. As discussed in the previous chapter, a product flow path must offer a temporally and spatially isolated area for the transport of products, and this sometimes requires the participation of plant elements outside of the corresponding flow route. Most of the time, these elements are switchable devices that are able to offer the form of isolation that is required, and which inhibit the occurrence of potentially hazardous situations due to incorrect product flow. In the example above, the valves \( V_1, V_2, V_4 \) and \( V_7 \) are responsible for providing this protection, and are thus logically a part of the product flow path that performs the transport of material from \( J_3 \) to \( T_3 \). Chapter 5 will present the criteria for determining which plant elements on the periphery of a product flow route are logically contained in the corresponding product flow path.

2.1.2 Life Cycle of Product Flow Paths

A product flow path operates in a similar manner to a train path in a railway system, as discussed in Chapter 1. This means that a product flow path must be used according to a well-defined procedure, which determines a sequence of steps that have to be taken in order to use the flow path. The life cycle of a product flow path describes the entire history of a product flow path during its existence, and is defined herewith by means of the state diagram shown in Figure 2.1. In a state diagram such as this, states are represented as labelled nodes, and transitions are represented as labelled arrows that connect the states. A state represents a possible situation of the system that is being described, in this case a product flow path, and a transition represents a change of state in the direction of the corresponding arrow due to an action or an event. As is usually the case with state diagrams, in this diagram it is assumed that a product flow path may only be in one state at a time.

Initially, a product flow path does not exist as a concrete entity. When a transport operation is required, a corresponding product flow path must first be created in order to fulfil this
2 Definition of Requirements and Tasks

operation. The flow route that corresponds to this flow path must be clearly identified in the structure of the plant, just like a train path is identified in the railway network prior to being used by a train. This identification of a flow route in a plant is called discovery in this work, and it constitutes one of the tasks that must be performed by a product flow path management system. Apart from the automatic discovery of product flow routes, a manual discovery may also be performed by a human operator by providing the full sequence of plant elements that makes up a product flow route. Once a flow route has been discovered and identified, a product flow path can be created for it. In Figure 2.1 this is shown by the initial transition labelled Discover/Create, which has no source state because the product flow path does not exist prior to its execution. After its creation, the product flow path is in a Created state.

The creation of product flow paths does not guarantee any form of exclusive access to the elements contained in the flow paths. In other words, two flow paths that cross each other or share a part of their corresponding flow routes may exist simultaneously. Before it can be used, a product flow path that is in a Created state must be allocated. The allocation of a product flow path is a logical operation that assigns to the flow path those plant elements that are contained in it, either as part of the corresponding flow route or as protecting elements that surround the flow route. This assignment is registered, so that the allocation of a second flow path that conflicts with an already allocated flow path will not succeed. After a product flow path is allocated, it reaches the Allocated state.

The allocation of a product flow path is a logical operation, which means that it occurs solely at the level of software. Before it can be used, a product flow path that has been allocated must be locked. Locking a product flow path involves emitting control signals to the various switchable plant devices that participate in the flow path, so that they set themselves to the correct position, and afterwards, establishing the necessary interlocks in the control system so that these settings are not modifiable by third parties. When this succeeds, the product flow path finds itself in a Locked state.

Once locked, a product flow path may be activated in order to begin the flow operation. The activation of a product flow path would involve commanding the operation of plant devices such as pumps, and possibly also setting some switchable elements to correct positions along the flow route. Once a product flow path has been successfully activated it reaches the Active state. At this point, the flow path is in use and the corresponding product transport operation is carried out.

The succession of steps that is taken to move a product flow path from a Created state to an Active state may be reversed in order to conclude the use of a product flow path. An active product flow path may be deactivated by issuing proper control signals to the corresponding plant elements, thus reaching a Locked state. A locked flow path may be unlocked by again issuing proper control signals and additionally releasing the corresponding interlocks, thus coming to an Allocated state. An allocated flow path may be logically deallocated in order to reach the Created state again. At this state, if the product flow path is no longer needed, it may be deleted, thus removing it from the control system.

The life cycle of a product flow path is flexible enough to allow a flow path to reach any of its states many times during its life. For instance, a product flow path may be created, then allocated, and then locked, but afterwards unlocked again. From here, the flow path may be either locked again, or deallocated. Any kind of sequence of states that follows the state diagram of Figure 2.1 represents a correct use of a product flow path, and depending on the operation of the plant, some of these reiterative sequences may be necessary in some cases. Most importantly, a product flow path that is in a given state, additionally fulfils the
requirements of each of the inferior states. This means that an allocated flow path is also created, a locked flow path is also allocated – and therefore also created, and an active flow path is locked, allocated and created. This form of hierarchy over the states of a product flow path assists in assuring the correct use of a product flow path, and thus, the correct flow of the products that are transported by the flow path.

2.2 Product Flow Path Management

A product flow path management system is a system that provides the functionality that is needed by a process control system in order to apply the concept of product flow paths to its operation. Therefore, a product flow path management system constitutes a subsystem, or component, of a process control system. This section provides a definition of the role of a product flow path management system, as well as a structural and functional model for implementing this system.

2.2.1 Product Flow Path Management Tasks

A product flow path management system must accomplish a collection of tasks that are required for the usage of product flow paths. Some of these tasks can be compared to the procedure that is involved in the usage of a train path as presented in Chapter 1, while others are inherent to the operation of product flow operations in processing plants. An overview of these tasks is given in the following.

- **Discovery of product flow routes.** A product flow path uses a determined product flow route in the plant, and this flow route must be defined before a corresponding flow path is used. The discovery of product flow routes in the plant is a task which, given certain conditions, identifies valid flow routes through the plant that flow paths can use to perform product transport operations.

- **Creation and deletion of product flow paths.** The life cycle of a product flow path begins when the flow path is created, and ends when the flow path is deleted. These operations, which manage the existence of product flow paths in the plant, must be carried out by a product flow path management system.

- **Assurance of product flow paths.** Before a product flow path is used, it must meet various conditions that assert the correctness of the corresponding product flow operation, such as the avoidance of potentially hazardous situations. The assurance of product flow paths is the general task of providing the proper conditions for the operation of a product flow path in a plant. With respect to the life cycle of a product flow path, this comprises the tasks of allocating and locking product flow paths, as well as their counterparts of deallocating and unlocking flow paths.

- **Activation of product flow paths.** Once a product flow path is assured, it may be activated in order to begin a product flow operation. The activation and deactivation of a product flow path is executed by the product flow path management system. For this, the services of the process control system which are commonly available for interacting with plant devices may be used.
• Monitoring of product flow paths. During its operation, a product flow path must be monitored in order to supervise its correct operation and promptly detect any related anomalies. Whereas the assurance of product flow paths offers a procedure of actively enforcing the correct use of a product flow path, the monitoring of a product flow path serves as an online validation of this correctness. The monitoring task may detect situations that are potentially hazardous, like an incorrect position of a valve. Unlike the previously presented tasks, flow path monitoring must be executed continuously while a product flow path is in operation.

• Issuing of product flow path alerts. During the operation of a product flow path, errors and problems of diverse nature may occur. For instance, a plant device may not respond as expected to a command, or a conflict with a second product flow path may be detected. In all of these cases, the process control system and the plant operators must be properly informed of this situation, and this is accomplished by issuing alerts which are associated to a determined product flow path in the plant.

• Documenting the life cycle of product flow paths. A product flow path goes through several stages of activity during its use, which are caused by the interactions between the product flow path management system, the user of the system, and the devices in the plant. A proper documentation of this behaviour is important for purposes of traceability, and should be accomplished by a product flow path management system for every product flow path that is used in the plant. This documentation consists of recording, for every flow path, every transition between states that occurs during the flow path’s life, as well as the occurrence of every alarm that is issued by the flow path. The information that is generated by this task should be archived in a proper manner, for instance, in a local file or in a database system.

In this work, the ways of realising the specific tasks of locking and activating product flow paths are not covered. The techniques that are required for implementing these tasks within a process control system are, in the general case, system-dependent and plant-dependent. Rather, the model that is presented in this work may be coupled with a subsystem that is responsible for realising these tasks, in a way that follows the ideas of Figure 2.1.

The rest of this section is concerned with outlining a model of a product flow path management system that implements the functional requirements outlined above, and in a way that makes it feasible to integrate these functionalities within a process control system.

### 2.2.2 Product Flow Paths as Objects

Object-oriented software engineering has gained momentum in recent years throughout many different application domains [5], and process control engineering is one of them [61]. The reason for this is that object-orientation allows the software to closely resemble the problem domain, and by understanding this domain, it becomes easier to understand – and develop – the software itself. In the case of product flow paths, it is only natural to represent these entities as objects in a process control system, together with their attributes and operations. Because of this, the present work follows an object-oriented approach that represents product flow paths as software objects.

Figure 2.2 shows a class diagram for the basic model of product flow paths, using the Unified Modelling Language (UML) [5]. We call this model basic because it contains the core
modelling details of a product flow path approach. The section that follows will present an extended version of this model, where additional details are included.

Two classes of objects are included in this model: FlowRoute represents product flow routes in the plant, and FlowPath represents corresponding product flow paths. As the Use relationship shows, a product flow route may be used by a single product flow path object, or not at all; respectively, a product flow path must use a single product flow route.

**Product Flow Route Objects**

A product flow route object represents a product flow route in the plant as described in the previous section, and has the following attributes:

- Origin holds the initial plant element of the flow route, and where the product is “taken” by the flow operation that is performed by a corresponding flow path,

- Target holds the final plant element of the flow route, and where the product is “left” by the flow operation that is performed by a corresponding flow path,

- Length holds the length of the flow route, that is, the number of plant elements that are part of the flow route,

- Elements holds the actual sequence of elements that conforms the flow route, from the initial element until the final element of the flow route.

The representation of the plant elements for their storage as attribute values is not defined by this model, but it must be possible to identify plant elements uniquely. Processing plants usually assign identifiers to all plant elements, and therefore, it is natural to use these same identifiers for representing plant elements within this model.

A product flow route object has no methods or operations. This means that these objects have no direct functionality other than representing a product flow route of the plant in the process control system.

---

**Figure 2.2:** Class diagram of the basic product flow path model.
Product Flow Path Objects

A product flow path object has both attributes and methods associated to it. The attributes of a flow path object are described as follows:

- Identifier holds a unique identifier of this flow path object,
- CreationTime holds the time when the flow path object was created,
- Allocated indicates if the product flow path object is currently allocated,
- Locked indicates if the product flow path object is currently locked,
- Active indicates if the product flow path object is currently active.

Every product flow path must have a unique identifier within the system, and this identifier is stored in the attribute Identifier. The creation time of the product flow path is kept as an attribute in order to help with the identification, and later documentation of the flow path. The attributes Allocated, Locked and Active coincide with the states of the product flow path that are shown in Figure 2.1, and thus, may be used to determine the state of the flow path at a given time.

The operations that are offered by the FlowPath class coincide with the state transitions of the state diagram that was shown in Figure 2.1. Therefore, the semantics of these operations are clear: when a product flow path object is at one of its possible states, the transition to another state may be requested by invoking the corresponding method of the product flow path object. If a method is invoked for a transition that is not valid from the current state of the flow path object, then the invocation has no effect.

Usage of Product Flow Path Objects

Within a process control system, the model of product flow path objects may be used as an interface for operating product flow operations in a plant. In order to execute a given transport operation, a product flow path object must first be obtained, and may then be used by following the procedure described in Figure 2.1. The interaction with these objects may be done directly by a human operator of the plant – in which case an adequate user interface must be provided – or may be carried out by additional software components within the system. In any case, the product flow path objects represent a new interface for executing product transport operations, and this offers an adequate abstraction from the actual device-level control of the plant.

2.2.3 Product Flow Path Management Object Model

The basic model of product flow path objects that has been specified in Figure 2.2 offers the essential functionality of an approach based on product flow paths. There are many tasks regarding product flow paths that are important for the proper use of a product flow path model within a process control system. Some of these tasks are a direct responsibility of a product flow path object, while others are important functionalities that must be provided by an environment that hosts product flow path objects. In this section, we complete the picture.
of a product flow path management system, which accomplishes the various tasks that are necessary for the use of the model of product flow path objects.

Figure 2.3 shows an extended version of the flow path model of Figure 2.2. Three new classes have been added to the model: FlowPathManager, whose instances – flow path manager objects – have the responsibility of handling flow path and flow route objects; FlowPathAlert, which represents alerts that may be emitted by a product flow path object; and FlowPathLogger, which represents flow path logger objects that must keep historical records of the life cycle of flow path objects in the system. A flow path manager object may handle any number of product flow path and product flow route objects, and this handling is done by the flow path manager object in an exclusive manner. Furthermore, the Emit relationship defines that a single product flow path may issue any number of flow path alerts, and that every alert is issued by exactly one product flow path. Finally, a flow path object may use any number of flow path logger objects for recording its own life cycle.

In the model of Figure 2.3, Alert is included as an attribute of the class FlowPath, and it indicates if any alerts have been issued by the product flow path object. This is done in order to have a simple way of determining if any problems have occurred with respect to the flow path. Flow path alerts may be issued by a product flow path because of the failure of an operation, or after the detection of a potential hazard. The state of alert that the Alert attribute indicates is independent of the states shown in Figure 2.1, which means that a flow path object may exhibit an alert status in any of these states.

The model of Figure 2.3 also adds the functionality of monitoring to the FlowPath class. The methods StartMonitoring and StopMonitoring respectively activate and deactivate the monitoring task within a product flow path object, which performs an online validation of the correctness of the flow operation that is carried out by the flow path object. In turn, the attribute Monitoring is an indicator of the activation of the monitoring task of a flow path object.
2 Definition of Requirements and Tasks

Just as in the case of the alert status, a product flow path object exhibits a monitoring status independently of the state or the flow path object with regard to the object’s life cycle that was shown in Figure 2.1.

Product Flow Path Manager Objects

Within a product flow path management system, a flow path manager object has the responsibility of handling flow path and flow route objects, which consists of accomplishing the following tasks:

- discovering product flow routes in the plant, and creating new product flow route objects,
- creating new product flow path objects based on existing product flow route objects, and
- storing the flow route and flow path objects that are created, and making them available to the process control system.

The task of discovering product flow routes is implemented by the operation DiscoverFlowRoutes, which additionally creates flow route objects for every flow route that is discovered in this manner. In turn, the task of creating these objects is handled by the operation CreateFlowRoute. Likewise, creating flow path objects is the task of the operation CreateFlowPath. All objects that are created by a flow path manager object are kept within the domain of this object, and are made available by the flow path manager to the other components of the process control system. In this manner, a flow path manager acts both as a factory and as a container of product flow route and product flow path objects in a product flow path management system.

Product Flow Path Alert Objects

When a product flow path issues an alert, it activates its Alert indicator and additionally creates an object of the class FlowPathAlert. A flow path alert object has no operations, and features the following attributes:

- Type holds the type of the alert, which indicates the nature and cause of the alert,
- Timestamp holds the time when the alert occurred,
- Elements holds the list of plant elements that were involved in the cause of the alert,
- FlowPaths holds the list of product flow paths (i.e. their identifiers) that were involved in the cause of the alert.

A flow path alert object may be used to report operation alerts in a very general manner. For instance, if a flow path cannot reach a locked state because a given plant element cannot be set to a desired position or cannot receive an interlock, then an alert object may be used to report this by setting the type of the alert to a value that indicates an error that occurred while locking the flow path, and by including the problematic plant element in the attribute Elements. Also, if an error occurs during the allocation of the product flow path because of a conflict with another flow path, an alert object may be used to report this error by setting
the type of the alert to an adequate value again, and by storing the identifier of the conflicting flow path in value of the attribute FlowPaths. Thus, a single class of objects may be used to report many different kinds of flow path alerts, and supports the addition of new alert types at later development stages of the system.

Product Flow Path Logger Objects

The task of documenting the life cycle of product flow path objects in a product flow path management system is accomplished by flow path logger objects. A logger object is responsible for saving a record of every state transition of the flow path object, as well as of every alert that the flow path emits. For this, the flow path logger uses a given data store for saving this information. A product flow path may use any number of flow path logger objects for recording its life cycle, and in this manner, any number of distinct data stores. For instance, a product flow path may use two flow path logger objects simultaneously: one to record information locally within the process control system, and another one to store information remotely at a database system. In this way, a publish/subscribe method or observer design pattern [19] is followed, where flow path loggers subscribe themselves with product flow path objects, which in turn communicate life cycle events to every flow path logger that is subscribed to them. This offers a flexible way to document the life cycle of product flow paths in the plant.

The methods of the class FlowPathLogger are described in the following:

- LogFlowPathCreation registers the creation of a product flow path object in the system,
- LogFlowPathState registers the successful execution of an operation that causes a product flow path object to reach a new state in its life cycle,
- LogFlowPathDeletion registers the deletion of a product flow path object from the system, and
- LogFlowPathAlert registers the creation of a new flow path alert by the product flow path object.

2.2.4 Assignment of Product Flow Path Management Tasks

The model of a product flow path management system that has been presented in this section fulfils the functional requirements that were outlined at the beginning of this section. That is, the different product flow path management tasks have been considered within the design of this system, and have been properly assigned to the classes of the system. This assignment of tasks is summarised in Table 2.1.

Flow path manager objects are responsible for discovering product flow routes in the plant, and for creating product flow path objects based on existing flow routes. In turn, product flow path objects have most of the responsibilities of a product flow path management system, namely, the assurance (allocation and locking), activation and monitoring of the flow path, as well as issuing flow path alerts. Also, flow path objects must implement the task of deleting themselves from the system. The task of documenting the life cycle of a product flow path is handled, as already mentioned, by instances of the FlowPathLogger class.
Table 2.1: Assignment of product flow path management tasks.

<table>
<thead>
<tr>
<th>Product Flow Path Management Task</th>
<th>Implementing Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discovery of product flow routes</td>
<td>FlowPathManager</td>
</tr>
<tr>
<td>Creation of product flow paths</td>
<td>FlowPathManager</td>
</tr>
<tr>
<td>Deletion of product flow paths</td>
<td>FlowPath</td>
</tr>
<tr>
<td>Assurance of product flow paths (allocation, locking)</td>
<td>FlowPath</td>
</tr>
<tr>
<td>Activation of product flow paths</td>
<td>FlowPath</td>
</tr>
<tr>
<td>Monitoring of product flow paths</td>
<td>FlowPath</td>
</tr>
<tr>
<td>Issuing of product flow path alerts</td>
<td>FlowPath</td>
</tr>
<tr>
<td>Documenting the life cycle of product flow paths</td>
<td>FlowPathLogger</td>
</tr>
</tbody>
</table>

2.3 Decentralised Product Flow Path Management

The management of product flow paths in processing plants, as described in this work, may be implemented by following a traditional *centralised* approach. This means that the flow path management system resides within a single computational unit, like a programmable logic controller or an automation system. Under this scheme, all objects of the flow path management system are placed in the same local memory of the machine, and are able to access each other’s attributes and call each other’s methods directly. Therefore, traditional programming techniques may be applied in order to implement the interaction between these objects. Although this implementation approach is straightforward, a product flow path management system that is developed in this manner has some drawbacks when operating in an actual processing plant. In order to fulfil the requirements of this operation environment in a better way, this work presents a *decentralised* approach for implementing a product flow path management system.

2.3.1 Advantages of Decentralisation

The first process control systems that were based on *process control computers* concentrated all process monitoring and control functions in a single computing unit following a *totally centralised* structure [45]. The main problem of this organisation scheme is that it offers a low availability of the process control functionality, because the failure of the process control computer affects the entire process. Therefore, the most common architecture used by process control systems at present is *decentralised*, where the functionality of process monitoring and control is spread across multiple process control computers. In this manner, a higher availability of the system is achieved. Chapter 3 presents this topic in greater detail.

In general, the decentralisation of process control systems offers many advantages with respect to centralised schemes. We discuss the most notorious of these advantages in the following.

- **Higher availability.** As already discussed, a service interruption at some decentralised controller affects only a part of the process, which means that the availability of the plant is higher than if a single central automation system is used. As a result of this, the system is more robust against errors and service interruptions, and these situations may be handled in a more efficient manner.
2.3 Decentralised Product Flow Path Management

- **Greater flexibility.** The fact that every decentralised automation system works independently from its peers allows a more flexible manipulation of the system. For instance, an automation station may be re-parametrised, reconfigured, reprogrammed, upgraded or even replaced without affecting the rest of the decentralised control system, thus offering a simpler maintenance of the system.

- **Greater scalability.** Decentralised process control system architectures are better prepared to handle large and complex plants. Decentralised control systems become larger by adding more decentralised controllers, whereas centralised control systems must grow by providing a larger and more capable controller. Therefore, the capability of a centralised process control computer limits the scale of the plant that his controller can handle. In contrast, a decentralised control system is able to grow with the plant, enabling the control of plants of any size in an efficient manner.

- **More efficiency.** The decentralisation of the process monitoring and control tasks leads to a parallelisation of these tasks, and therefore, to a higher computational throughput. This is possible because many of these tasks refer to a limited area of the plant, and because of this locality, they are able to work completely independent of other tasks.

- **Greater adaptability.** The greater flexibility of a decentralised process control system also allows the system to adapt to new contexts dynamically. Flexible communication networks allow decentralised controllers to leave and join the network dynamically, and this permits changing the physical and topological structure of the process control system in order to adapt to new plant layouts and new process control needs.

The multiple advantages of following a decentralised design approach, coupled with the fact that most process control systems currently in use operate in a decentralised manner, strongly motivates the adoption of decentralised techniques in the design of a product flow path management system. In doing this, many of the advantages of a decentralised approach can be foreseen as benefiting the management of product flow paths: the failure of a decentralised controller and its successive servicing only affect the operation of a limited number of flow paths, larger flow paths may be handled in a scalable manner, unrelated flow paths are handled independently and therefore more efficiently, and finally, the product flow path management system can be adapted to the plant in a more flexible manner.

2.3.2 Decentralised Railway Control

The analogy of railway paths and product flow paths that was presented in Chapter 1 has been used as an inspiration for developing a model of product flow paths in processing plants. Just as the topic of decentralisation is important in the field of process control, it has also influenced the design of modern railway control systems. Therefore, when adopting a decentralised approach for a product flow path management system, we may also obtain some inspiration from the decentralised techniques that are used in the field of railway control.

Railway control systems exist in both centralised and decentralised forms [43]. In [39], a system for the locking of railway paths is presented which works entirely in a decentralised manner, as opposed to the more common interlocking stations which are centralised and
custom-built for the corresponding area of railway tracks that they control. The system consists of distributed control objects which are hardware and software components that communicate with each other. Each control object corresponds to a single track element, which in turn is either a signal, a switch or a track block, and the behaviour of each control object depends on the type of the corresponding track element. The control objects are laid out next to their corresponding elements and are also interconnected with each other in an analogous way. This allows the system to be easily assembled by following the structure of the railway network. It also makes it easy to adapt and extend the system whenever the railway network changes. A more complete description of this approach is given in Chapter 3.

### 2.3.3 Decentralised Product Flow Path Management Systems

A decentralisation scheme for a product flow path management system may be adequately developed by considering the composition of decentralised process control systems, and the decentralisation principles used in railway control. The former offers decentralised controllers which are capable of communicating with each other and of interacting with plant devices. In turn, the latter seek to distribute the operation of the control system across the railway network, such that unrelated operations are able to execute independently, and such that control systems may be easily assembled by considering the structure of the railway network. Therefore, a decentralised product flow path management system distributes its tasks – and therefore its objects – among several decentralised controllers by following the structure of the plant.

With help of Figure 2.4, the decentralisation principle that is used in the design of product flow path management systems for the sample filling station.
2.3 Decentralised Product Flow Path Management

flow path management systems is explained by considering its application to the sample filling station that was presented in Figure 1.2. This figure shows a division, or partition, of the sample plant by means of dashed lines, such that every partition contains an individual plant element. The basic idea of the decentralisation scheme of a product flow path management system is that every such partition, and therefore every plant element, is assigned exclusively to a single decentralised controller. In this manner, the structure of the plant is mimicked by the structure of the decentralised product flow path management system.

As opposed to decentralised control in railway systems, a one-to-one mapping between plant elements and decentralised controllers is inadequate, because plant elements will usually outnumber decentralised controllers in a plant. Processing plants are commonly divided into logical plant sections, and this division is also used to separate the tasks of the control system among the distinct controllers. Therefore, it is natural to assign multiple plant elements to a single decentralised controller, and this is supported by a decentralised product flow path management system. In Figure 2.4, the sample filling station has been divided into seven sections that are indicated by contiguous shadings of the same colour. By this example, each of these sections may be completely assigned to a decentralised controller, thus requiring seven of these controllers to deploy the decentralised product flow path management system for this plant. The assignment of plant elements to controllers may occur in any way that is desired, as long as the rule of assigning every plant element to a single decentralised controller is obeyed. Chapters 4 and 5 will present plant models and algorithms which together accomplish the tasks of a product flow path management system under this decentralisation scheme.

Since a decentralised product flow path management system is distributed across several decentralised controllers, the instances of the classes from Figure 2.3 must also be assigned to the different controllers of the system. By this approach, a decentralised controller hosts the following objects:

- a single flow path manager object that manages local flow route and flow path objects,
- flow route and flow path objects whose initial plant element must be assigned to the controller,
- flow path alert objects that are issued by local flow path objects and,
- flow path logger objects that record the life cycle of local flow path objects.

These policies enforce the distribution of objects across the decentralised product flow path management system to occur in an unambiguous manner. Every controller hosts a single flow path manager object, which means that there are equally many decentralised controllers as there are flow path manager instances. Furthermore, a flow route object, and every corresponding flow path object, is created in the controller which has been assigned to the initial plant element of the flow route. Therefore, even though a product flow path may involve plant elements from many different controllers, the object resides in only one of these controllers, namely, the one that has been assigned the initial element of the corresponding flow route. Finally, all flow path alarm and flow path logger objects that are related to a flow path object, coexist with this object in the same decentralised controller.

In this manner, a simple and effective way of distributing the functions of a product flow path management system in a decentralised process control system is achieved. The algo-
rithms which are able to implement the tasks of a product flow path management system in a decentralised manner are presented in Chapter 5.

2.4 Synthesis of Product Flow Path Management Systems

In Chapter 1 it was argued that the engineering of the control functionality for product flow operations in a plant, without the support of a model for the correctness of these operations, increases the engineering cost and the probability of errors, and that one of the goals of the adoption of a product flow path model is to tackle these problems by providing a method of automating the task of engineering a product flow path management system. The application of the decentralised approach that has been presented in the previous section paves the way for a synthesis technique for the automatic construction of decentralised product flow path management systems.

As seen in Figure 2.4, the deployment structure of a decentralised product flow path management system follows the structure of the plant. Therefore, if a machine-readable representation of the structure of the plant is available, an algorithm may be devised which reads this representation and, based on the information contained therein, creates and configures the objects of a decentralised product flow path management system that corresponds to the plant. This procedure is similar to the construction of a decentralised railway control system that is composed of control objects that are laid out along the structure of the railway tracks as described in [39]. Concretely, a synthesis algorithm such as the one envisioned needs the following information:

1. a machine-readable representation of the structure of the plant, where the plant’s elements are clearly represented together with their relevant attributes and the interconnections among the elements, and

2. information about configuration of the deployment, which offers a mapping of every element in the plant to a decentralised controller of the process control system.

The first of these requisites may be fulfilled by the use of plant engineering data files in formats such as CAEX [15], which are readily available for many plants today. The second requisite may be accomplished by following a default mapping scheme, say by assigning the plant elements of each plant section defined in the representation of the structure of the plant to a given controller, or more generally, by crafting this deployment information in accordance with a desired control structure.

After the execution of the synthesis algorithm, the decentralised product flow path management system is ready to begin its operation. Each of the various decentralised controllers of the system has an active instance of a flow path manager object, and knows which plant elements have been assigned to it. Additionally, proper communication links have been established among the decentralised controllers, such that the execution of every product flow path management operation is possible, even for flow paths whose elements span multiple decentralised controllers. The details of this decentralised algorithmic approach, as well as of the algorithm for performing the synthesis of the decentralised product flow path management system, will be explained in Chapter 5.
3 The State of the Art

The present work belongs to the field of process control engineering, and builds on a large body of theory and practise which has been developed in this area throughout many years. However, it is essentially focused on developing a software technology, and therefore follows methodologies from the fields of computer science and software engineering. Additionally, it borrows ideas from other fields such as railway engineering and the engineering of communication systems. In order to bring all of the relevant areas of knowledge into perspective, this chapter presents an overview of the relevant aspects of the state of the art in each of these areas.

3.1 Process Control Engineering

Process control engineering [17, 45] is the field that studies the systems that control the operation of processing plants. In this context, the term systems covers physical, mechanical, electrical, electronic and software systems, and the term control covers a very broad range of functions that a process control system must accomplish. Furthermore, process control systems are not completely autonomous, but rather perform many of their tasks in cooperation with human operators of the plant.

The most important tasks that process control system performs in a plant are classified and outlined in the following.

- **Automation technology.** The realisation of tasks within direct contact of the processes in the plant, and with a high degree of automation.
  - **Measurement.** Obtaining readings of the magnitude of physical properties of the processes in the plant during operation, and transmitting these values for their remote use. This task is carried out by devices known as sensors or transmitters.
  - **Automatic control.** Applying techniques from control engineering, such as open-loop and closed-loop control [18], in order to regulate the processes in the plant.
  - **Actuation.** Driving active plant devices known as actuators, such as valves, pumps, compressors and stirrers.

- **Process control technology.** The collection of technologies and tools that enable the operation of processes in a plant, under the direction and supervision of human operators.
  - **Operation.** Performing plant-level automation in the form of control sequences. Also, offering a user interface for operators, so that they are able to initiate and direct the execution of processes in the plant.
  - **Monitoring.** Gathering of process information, commonly in the form of sensor readings, and making this information available to plant operators and to other systems.
– **Reporting and alarming.** Reporting the occurrence of events in the plant to the operators, in the form of notifications and alarms. Also, tracking the acknowledgement of notifications and alarms from the part of the plant operators.

– **Engineering.** Providing a user environment that enables engineers to create, edit and maintain the control logic of the plant, including control loops, control sequences and interlocks. This control logic is transferred to the various plant automation systems for their execution in the form of stored programs.

– **Archiving.** Recording the relevant information about the operation of the plant in data stores, in order to support the traceability of the plant’s operations.

In addition to these main tasks, the information technology at the enterprise level is becoming more and more tightly coupled with the systems at the level of process control. These technologies may span the operation of several plants, and are responsible for supporting the handling production orders, supply chain management, engineering of plant layouts, plant asset management, the tracking, quality assurance, inventorying and documentation of products and the planning, acquisition, installation and monitoring of plant devices.

### 3.1.1 Engineering of Plants and Control Systems

The design of a plant begins with the design of the actual process or processes that the plant will execute. A process, in this context a chemical or physical process, is specified by process engineers, and is usually represented conceptually by means of process block diagrams [14] or formalised process descriptions [60] that depict the processes as transformations of raw products, yielding main products and byproducts, and consuming and producing energy flows at the same time. Once a process is properly specified, and normally tested in a laboratory, process flow diagrams [14] are used to represent the realisation of the process by means of common plant equipment. These diagrams relate the process to this equipment, but do not give enough detail for actually building a corresponding plant. For this, piping and instrumentation diagrams (P&ID) [14] are created, which depict the structure of a plant in a detailed manner, together with the location of the necessary measurement and control functions in the plant. These diagrams are used as a reference for the construction of the plant, and at the same time, for the design of the corresponding process control system. Figure 3.1 shows an example of a P&ID. For the design of large and complex plants, software packages known as CAE systems (Computer-Aided Engineering) are commonly employed, which offer an environment that supports the design phase of a plant, together with the creation of P&ID.

The complete implementation of a process control system for a plant consists of two main phases: the installation of the control hardware, and the engineering of the control logic of the plant. These two phases are explained in the following sections.

### Installation of the Control Hardware

The first phase in the implementation of a process control system is the acquisition, installation and configuration of the process control hardware. This includes many devices of a wide range of kinds. Automation stations, or process control computers, are usually digital programmable logic controllers (PLC) that are based on microprocessors, host a real-time operating environment, and have communication interfaces for exchanging data with other
components in the plant. Operator stations are human-machine interfaces used by operators to supervise the activity of the plant, for which interactive computer systems with common components like screens, keyboards, and mice are usually employed. Most modern plants use digital communication networks called fieldbuses [25] which enable real-time communication between controllers and sensors and actuators in the plant. Thus, the necessary hardware for the use of the fieldbus is also required. In addition to fieldbuses, analog and digital input/output interface modules are commonly employed to interact with sensors and actuators in the plant. Also, the required power supplies for the various active devices must be provided.

The process control hardware must be physically installed and configured so that it may inter-operate correctly, both physically and logically. The physical interconnection of the various plant devices must be carried out and configured so that the communication modes at the electrical level agree. The participants in a fieldbus communication network must be properly configured in order to communicate effectively with one another. And on the side of an automation station, every relevant sensor and actuator connected to the fieldbus or to an input/output port of a communication module must be properly identified. The interoperability of the devices in the plant is commonly tested after the installation phase.

Engineering of the Control Logic

Modern process control systems are integrated hardware and software platforms that offer complete engineering environments. This means that once the system is correctly installed and configured, process control engineers may implement, test and deploy the desired control logic of the plant. The engineering environment is usually a graphical user interface (GUI) that
runs on one or more computers that are called *engineering stations*.

The control logic of the plant is usually designed using one or more engineering languages that may be either text-based or graphical. The IEC 61131-3 standard [23] defines five engineering languages, three graphical and two text-based, that may be used to program programmable logic controllers. *Instruction List* (IL) is textual language which represents programs as sequences of simple instructions, similarly to the assembly language of a microprocessor. *Structured Text* (ST) is a textual high-level structured programming language similar to the language Pascal. *Ladder Diagram* (LD) is a graphical language that represents control logic as diagrams of electrical circuits that use *relays*, which are essentially circuits of on/off switches. *Function Block Diagram* (FBD) is a graphical language that represents control logic by means of function blocks that are connected with each other through data (or signal) connections. A function block has input and output variables, and contains an algorithm that assigns the values of the output variables based on the values of the input variables. The output variables of one function block may be connected to — and thereby equated with — the input variables of other function blocks in order to build up the intended control logic. Function block diagrams are ubiquitous in process control engineering environments, and are a very common way to represent the control logic of processing plants. The last language of the IEC 61131-3 standard is *Sequential Function Chart* (SFC), which is used for organising sequential and parallel control functions in a controller, where the individual control functions in the sequence must be themselves specified in one of the previous four languages. The language features action steps, conditional transitions, and directed links between the steps and the transitions, following a similar structure to that of Petri nets [44]. Most process control engineering environments offer languages that are similar to those of the IEC 61131-3 standard to a greater or lesser degree, with some environments being compliant with the standard. Therefore, the IEC 61131-3 standard describes the programming techniques that are used at present to design the control logic for processing plants.

**Goals of the Control Logic**

The individual goals that the control logic of a process control system aims to reach can be very diverse in practise, ranging from flexible switching of control strategies, sensors and actuators, to complex calculation of derived quantities based on physical models [6]. In general, these goals depend on the type of plant and the kind of processes and products that are found on the plant. However, the most common goals of the control logic used in process control systems may be outlined as follows.

- **Continuous control of process values.** Using techniques from control engineering [18], the values of some process quantities, such as temperature, pressure, flow velocity, level in a tank, etc., may be controlled in order to enforce desired values by directly manipulating other process parameters such as intensity of a heater, speed of a pump, aperture of a control valve, etc. These control functions are inherently continuous and are commonly expressed using function block diagrams.

- **Sequential control of procedures.** Many aspects of the operation of a plant are best described as sequences of steps, such as the procedures to begin and end a certain chemical reaction in a reactor, or the operation of filling a tanker vehicle. These sequential procedures are commonly expressed using the sequential function chart language,
• Safety assurance through interlocks. Interlocks in process control systems may be conceptually described as condition/action pairs which are used to prevent undesired situations in the plant. When the condition of the interlock is satisfied by the current state of the plant, the corresponding action is triggered in order to inhibit the cause of a potential problem. The action of an interlock is sometimes reactive, for instance shutting off a mixer when a foreign object is detected in a container, and sometimes preventive, for instance by disabling the activation of a pump if a certain valve is closed. Often, special purpose controllers are destined to exclusively execute this kind of safety-related control logic, in order to offer an independent and sometimes redundant implementation of the safety functionalities. Because of their conditional nature, interlocks are often designed using ladder diagrams or function block diagrams. Figure 3.2 shows an example of a function block diagram which represents the safety interlock of a pump under the consideration of the statuses of relevant valves and tanks in the plant.

3.1.2 Continuous Processes and Batch Processes

The design of a plant and its corresponding process control system greatly depends on which processes are to be carried out in the plant. Though a great variety of processes are currently executed in processing plants, all processes may be generally classified into two main groups, namely, continuous processes and batch processes [24]. The nature of a process
Continuous processes are those processes where a continuous flow of products is yielded by processing a continuous flow of raw materials for an arbitrarily long period of time. To an observer, an operating plant seems to be still, with occasional small fluctuations in the different process values. The main goals of a process control system for continuous processes is maintaining the operation of the plant in a desired steady state, and monitoring this operation. Therefore, the greater part of the control logic of these systems is devoted to continuous control or process values.

Batch processes are those processes where given quantities of products are produced by processing given quantities of raw materials at determined time intervals. The operation of a batch processing plant switches among processes and products, and is described by a schedule that indicates what processes are to be executed at which time. A process control system for batch processes must provide the functions for beginning, executing and ending the individual processes, and commonly, for performing plant-cleaning operations in between processes. Also, the information about the kinds of products that are produced in the plant and the processes that are used to produce them must be managed in the form of recipes, which permit the specification of products and the execution of their corresponding processes. Whereas continuous control of process values is needed during the operation of an individual process, sequential control conforms a great part of the control logic of a process control system for a batch processing plant.

### 3.1.3 Decentralised Control Systems

Chapter 2 introduced the concept of decentralised process control systems, which are the most common form of process control systems at present [45]. Decentralised (or distributed) control systems (DCS) are composed of a collection of control computers, automation systems, or simply controllers, that are connected to each other and to other computer systems in the plant – such as operator and engineering stations – by means of a communication network commonly called a system bus. The controllers are distributed across the plant, sometimes just logically and other times also physically, and each controller is normally assigned the control and monitoring tasks of a given plant section. The controllers are able to communicate directly with the sensors and actuators in the plant via a fieldbus, through dedicated analog or digital lines, or by means of remote I/O modules that are directly connected to the sensors and actuators and which are themselves accessible via the fieldbus. Using this setup, the operations of the plant may be executed and supervised by plant operators sitting in a control room by interacting directly with the operator stations, while the actual control logic is executed on the individual decentralised controllers. The many advantages of a decentralised control system over a centralised one have been outlined in Chapter 2, being the most important one the high availability of a decentralised control system, which may continue operating, at least partially, when one of its controllers fails.

### 3.1.4 Control of Material Transport

Up to now, this chapter has given an overview of the current state of the art in the field of process control engineering. As the main topic of this work is the correct transport of material
3.1 Process Control Engineering

in a plant, we will now concentrate on this particular aspect of process control by covering two corresponding approaches that are used at present: traditional manually engineered control, and an automatic technique offered by the SIMATIC PCS 7 system from Siemens.

Manually Engineered Control

Traditional manually engineered control is the most common way to implement the control of product flow operations in today’s plants. By this approach, the control logic of the different aspects of the flow operations in the plant, such as control sequences and safety interlocks, are manually designed by engineers, using process control languages such as those standardised in the IEC 61131-3 norm, and in most cases, producing diagrams such as the one shown in Figure 3.2. In order to accomplish this, the engineers must study the layout of the plant carefully, and consider every detail of the relationships and interactions among the different plant elements. For instance, and as depicted in Figure 3.2, an interlock may condition the ability to activate a pump for a product transport operation based on the level of a source tank, the levels of two different destination tanks, and the positions of three neighbouring valves. This is done in order to inhibit the pump from operating with an empty source of material, full destinations of material, or incorrect valve settings. By this example, it becomes clear that the design of this control logic must be carefully developed: should the engineer forget to consider a tank or valve in this analysis, then the control logic would be incomplete, and thus incorrect. Manual engineering techniques are also employed for developing control sequences for startup, execution and shutdown of product transport operations.

For large plants, and for long product routes, the control logic is usually too complex to be handled all at once, and modularisation techniques are employed to break down the control logic into manageable units that interact with each other within the control system. Nevertheless, the entire plant must be carefully studied by engineers in order to design a control logic that considers every relevant interaction of the elements of the plant.

SIMATIC Route Control

SIMATIC PCS 7 [55] is a process control system developed by Siemens AG. Apart from the functionalities offered by practically all modern process control systems, SIMATIC PCS 7 features an add-on software package specifically designed for supporting the configuration, control and monitoring of material transport in piping systems called SIMATIC Route Control [53]. By using this package, it is possible to automate the execution of flow operations in flexible plants with complex piping structures in a simplified manner.

SIMATIC Route Control consists of three software modules: an engineering environment called Route Control Engineering, an execution environment called Route Control Server, and an operator client application called Route Control Center. All three modules run as programs on the Microsoft Windows operating system, and it is recommended that each module be run in a different machine.

The engineering environment of SIMATIC Route Control is integrated with the general engineering environment of the PCS 7 system. After having configured the latter environment with the relevant information about the plant, this information is imported into the former environment in order to begin with the design of the control of transport operations. This engineering process follows a series of steps that may be roughly described as follows.
1. **Define locations in the plant.** The information about the plant that is available to the engineering environment comprises the collection of elements in the plant, such as tanks, pumps and valves. However, there is no information about the structure of the plant that may be used by the system to determine the product flow routes of the plant. For this, the engineer must begin by manually defining *locations* in the plant, which are reference points for the later construction of flow routes. Locations are virtual plant elements in the engineering environment, and must be “placed” in the structure of the plant in a convenient way, for instance on tanks and on pipe joins. Each location may be configured as a *source*, *destination*, or *via*, which respectively denotes the intended position of a location within a flow route: initial, final, or intermediate.

2. **Configure mode tables.** In order to enable the definition of the control sequences that are needed to execute product flow operations, the engineer creates *mode tables*, which are tables that contain the steps that a given control sequence may take. These steps are in turn called *mode levels*, and have names that clearly indicate their intentions, for instance “open valves” or “start pump”.

3. **Configure partial routes.** In order to be able to determine the valid flow routes in the plant, the system needs to know the connection structure of the elements of the plant. The engineer specifies this information by adding *partial routes* to the system. A partial route is the minimal section of a flow route, and it consists of the following information:
   - an identifier that is unique within the system,
   - a mode table,
   - a priority (between 1 and 9999),
   - an indicator of bidirectionality (activated/deactivated),
   - a source location,
   - a destination location, and
   - a collection of assigned elements.

   The types of elements that can be assigned to a partial route are, among others, control elements that are able to control plant devices like pumps and valves, and sensor elements that obtain readings from sensors in the plant. The actual control behaviour of a partial route is specified in a corresponding control table that assigns an action for every element assigned to the partial route, and in every mode level of the corresponding mode table. Actions are, for instance, the activation or deactivation of an active element such as a pump, or the monitoring of the level in a tank. Thus, when a partial route is commanded to execute a given mode level, then every element that is assigned to the partial route executes the corresponding action that has been defined for it for this model level.

4. **Define materials.** The kinds of materials that are to be transported by the piping system must be defined in the system, and the engineer accomplishes this by enumerating the names of these materials in a list. Materials may also be collected in material groups for simplifying the management of large numbers of materials.

5. **Define material successor relationships.** During operation, a given piping section may be needed for transporting one type of material, and afterwards for transporting another
3.1 Process Control Engineering

Depending on the types of these materials, this situation may pose a problem. For instance, the materials could react and cause a dangerous situation, or the second material could be damaged as a result of the combination with the first material. In order to tackle this problem, SIMATIC Route Control requires the definition of valid sequences of materials by means of a material successor relationship. This relationship associates each material with every other material that may follow it in a product transport operation, and any pair of materials that is not specified in this relationship is assumed to be an invalid material sequence. In this manner, incorrect sequences of materials are avoided, and at the same time, required sequences are enforced. For example, if a given material must always be followed by a cleaning operation with a given cleanser, then the cleanser may be configured as the only successor of the material in the engineering environment.

6. Download configuration to server. Once the task of engineering the control of the product transport operations has been completed, the control logic may be transferred to the route control server by the engineering environment for its use as part of the process control system.

During the operation of the plant, an operator may request the execution of a transport operation by specifying the parameters of the desired route. These parameters are the source location of the material, the destination location of the material, the via locations that must be visited by the material (optional parameter, up to 10 locations may be given), and the material that is to be transported. Based on this information, the route control server executes a search for flow routes in the plant. It does this by finding the shortest sequence of partial routes that fulfils the following conditions:

- the source location of the first partial route is the source location of the specified route,
- the destination location of the last partial route is the destination location of the specified route,
- the destination location of a partial route is the source location of the partial route that follows it,
- no partial route in the sequence is currently in use by another operation, and
- the material to be transported is a valid successor of the last material that was transported by every partial route.

In this search, partial routes that have been configured as bidirectional may be reversed, meaning that their source and destination locations are interchangeable. Apart from giving preference to the shortest route that fulfils the above conditions, the partial routes with the highest priority are preferred by the search algorithm. When the route is found, the operator may request the initiation of the transport operation. Generally, the mode table of the partial routes is configured so that the following steps are taken during this initiation:

1. the active elements are set to their basic positions,
2. the valves along the path, and surrounding it, are set to appropriate positions,
3. the source is tested for emptiness,
4. the destination is tested for fullness, and

5. the corresponding pump is started.

Likewise, the finalisation of the operation follows a similar sequence of steps. In this manner, the execution of product transport operations in plants with flexible and complex piping systems is supported by means of a route-oriented engineering approach, and an online search algorithm that finds flow routes in the plant under constraints that regard the priorities of plant sections and the correct sequences of materials.

3.2 Computer Science

Computer science is a broad field that studies the subjects of information and algorithms. The study of information is mainly concerned with its structuring and interpretation. In turn, algorithms are formal mechanisms that operate on information, and their study involves analysing their capabilities (what can algorithms accomplish) and the amount of resources that they require for their operation (usually execution time and storage space). Since algorithms both operate on information and are a form of information, the study of these two concepts is strongly coupled.

While theoretical computer science is strictly a branch of the field of mathematics and logic, the practical application of computer science driven by the emergence of software engineering has caused the domain of computer science to expand and combine with other fields from the areas of natural sciences, engineering, economics, social sciences, health, politics and art.

The following sections present an overview of some topics from the field of computer science that are relevant for this work.

3.2.1 Fundamental Theories

The development of theoretical computer science was based on well-known theories from mathematics and logic, and has contributed to these fields with some theories of its own. This section gives an overview of some theoretical notions from computer science that will be referred to in Chapters 4 and 5.

Set Theory

Set theory [11] studies the notion of sets, which are collections of objects called elements. Sets may contain an infinite number of elements, although the sets that are used in this work are all finite. A set $A$ that contains the elements $x_1, \ldots, x_n$ is defined by $A = \{x_1, \ldots, x_n\}$, where all these elements are distinct from one another, and where the ordering of the elements is irrelevant. If a set $A$ contains an element $x$, this is written as $x \in A$; likewise, $x \notin A$ expresses that the element $x$ is not contained in the set $A$. The empty set, which is the set that contains no elements at all, is denoted by $\emptyset$. The cardinality $|A|$ of a set $A$ is the number of elements that are contained in $A$.

A set $B$ is a subset of a set $A$, written as $B \subseteq A$, if every element contained in $B$ is also contained in $A$. If $A$ additionally contains an element that is not in $B$, then $B$ is a proper subset of $A$, written as $B \subset A$. 
There are a number of operations that can be applied to sets. The union of two sets $A$ and $B$, $A \cup B$, is the set that contains every element that is contained in $A$, or in $B$, or in both. The intersection of two sets $A$ and $B$, $A \cap B$, is the set that contains every element that is contained in both $A$ and $B$. The subtraction of a set $B$ from a set $A$, $A \setminus B$, is the set that contains every element in $A$ that is not in $B$. Finally, the power set of a set $A$, $2^A$, is the set that contains all subsets of $A$, including the empty set $\emptyset$. This notation is used to represent the power set because its cardinality is given by $|2^A| = 2^{|A|}$.

An $n$-tuple is an ordered sequence of $n$ elements written as $(x_1, \ldots, x_n)$. Given the sets $A_1, \ldots, A_n$, the $n$-ary Cartesian product $A_1 \times \ldots \times A_n$ of these sets is the set that contains all $n$-tuples $(x_1, \ldots, x_n)$ where $x_1 \in A_1, \ldots, x_n \in A_n$. Because this product contains every possible combination of the elements in the sets $A_1, \ldots, A_n$, the cardinality $|A_1 \times \ldots \times A_n|$ of the Cartesian product is given by the multiplication $|A_1| \cdot \ldots \cdot |A_n|$. A set of $n$-tuples $R \subseteq A_1 \times \ldots \times A_n$ is called an $n$-ary relation.

Given two sets $A$ and $B$, a function $f$ from $A$ to $B$, written as $f : A \rightarrow B$, is a mapping that associates every element of $A$ with a single element from $B$. A function $f$ is a binary relation $f \subset A \times B$ such that for every $(a, b_1) \in f$ and every $(a, b_2) \in f$, it must be that $b_1 = b_2$. If $(a, b) \in f$, the notation $b = f(a)$ is commonly employed. Also, the notation $|A \rightarrow B|$ is used to represent the set of all functions that associate elements of the set $A$ to elements of the set $B$.

### Formal Languages

The theory of formal languages [21] is of central importance to the field of theoretical computer science, helping to define the property of computability. It has also supported the development of programming languages, communication protocols and the use of textual databases, among others.

Formally, an alphabet is a set of symbols. A word, or string, over an alphabet $\Sigma$ is a finite sequence of symbols from $\Sigma$. Thus, for an alphabet $\Sigma = \{a, b, c\}$, the sequences $a$, $bc$ and $cbac$ are all words over $\Sigma$. The empty word, which contains no symbols at all, is denoted by $\varepsilon$. The alphabet of a word $w$, which only contains those symbols that are found in $w$, is denoted by $a(w)$. If $w$ is a word, then $w^R$ denotes its reverse word, which denotes the inverted sequence of symbols of $w$. If $w$ and $x$ are words, then the word $wx$ is their concatenation, which is composed of the sequence of symbols of $w$ followed by the sequence of symbols of $x$.

A language over an alphabet $\Sigma$ is a set of words over $\Sigma$. Because languages are sets, all the operations on sets that were specified in the previous section equally apply to languages. Additionally, the following operations on languages are commonly used, where $L$ and $M$ are languages over $\Sigma$ and $w$ and $x$ are words over $\Sigma$.

- **Concatenation:** The language $L \cdot M$ is the language that contains every word of the form $wx$, where $w \in L$ and $x \in M$.
- **Kleene closure:** The language $L^*$ is given by $\{\varepsilon\} \cup (L \cdot L^*)$, that is, the language that contains all those words that can be obtained by concatenating the words in $L$ zero or more times.
- **Reversal:** The language $L^R$ is the language that contains the reverse word $w^R$ of every word $w \in L$.  

---

3.2 Computer Science
The notation $L^+$ is sometimes used to denote the language $L \cdot L^*$, that is, $L^*$ without the empty string $\varepsilon$. Because an alphabet $\Sigma$ additionally denotes a language that contains its symbols as single-length words, the language $\Sigma^*$ contains all those words that can be obtained by concatenating symbols from the alphabet $\Sigma$ zero or more times.

Finally, the projection function $\pi_\Sigma : U^* \rightarrow \Sigma^*$, where $U$ and $\Sigma$ are alphabets and $\Sigma \subseteq U$, removes all symbols which are not contained in $\Sigma$ from a word $w$ and is defined as

$$\pi_\Sigma(w) = \begin{cases} 
\varepsilon & \text{if } w = \varepsilon \\
\pi_\Sigma(x) & \text{if } w = ax \text{ and } a \notin \Sigma \\
a\pi_\Sigma(x) & \text{if } w = ax \text{ and } a \in \Sigma 
\end{cases}$$

where $w$ and $x$ are words over $U$ and $a \in U$.

Formal Logic

Formal logic [22] studies formal languages called logics that are used to express sentences or formulae, which are affirmations that may be true (valid) or false (invalid) in a given context. Additionally, formal logic studies systematic techniques called deductive systems that may be used to determine the validity of sentences. In computer science, formal logic is used extensively, from the design of computer processors through programming languages, database systems, artificial intelligence and the automatic verification of systems.

Propositional logic is a logic where sentences may be formed by combining atomic propositions, such as $p$ and $q$, with logical connectives. Atomic propositions may be either true or false in a given context, and the validity of a sentence is thereby dependent on the validity of the atomic propositions that it contains, as well as on the structure of the sentence. The following rules may be used to construct sentences in propositional logic, where $p$ is an atomic proposition and $\alpha$ and $\beta$ are themselves sentences.

- Atomic proposition: The sentence $p$ is valid if and only if the atomic proposition $p$ is valid.
- Negation: The sentence $\neg \alpha$ is valid if and only if $\alpha$ is not valid.
- Conjunction: The sentence $\alpha \land \beta$ is valid if and only if $\alpha$ is valid and $\beta$ is valid.
- Disjunction: The sentence $\alpha \lor \beta$ is valid if and only if $\alpha$ is valid, $\beta$ is valid, or both $\alpha$ and $\beta$ are valid.
- Implication: The sentence $\alpha \Rightarrow \beta$ is valid if and only if $\beta$ is valid whenever $\alpha$ is valid. The sentence $\alpha \Rightarrow \beta$ is equivalent to the sentence $(\neg \alpha) \lor \beta$.
- Equivalence: The sentence $\alpha \Leftrightarrow \beta$ is valid if and only if $\beta$ is valid whenever $\alpha$ is valid and $\alpha$ is valid whenever $\beta$ is valid. The sentence $\alpha \Leftrightarrow \beta$ is equivalent to the sentence $(\alpha \Rightarrow \beta) \land (\beta \Rightarrow \alpha)$.

Occasionally, a conjunction $\alpha \land \beta$ is written as $\alpha \beta$.

First-order logic is an extension of propositional logic that introduces predicates and quantification. An $n$-ary predicate is a syntactic construction of the form $P(x_1, \ldots, x_n)$, where $P$ is a predicate symbol and $x_1, \ldots, x_n$ are variables that denote objects in a given context.
A predicate, just like an atomic proposition, may be valid or invalid in a given context, and 0-ary predicates are actually atomic propositions. The objects that variables denote must be defined in the context of the predicate that contains them, and this is done through the use of quantifiers that associate elements of a set, known as the domain of discourse, with variables. This set depends on the context, and contains all objects that may be referred to within this context. As in the case of propositional logic, the validity of a sentence depends on the validity of the predicates that it contains, and on the structure of the sentence. The following rules may be used to construct sentences in first-order logic, where $P$ is a predicate symbol, $x, x_1, \ldots, x_n$ are variables, $A$ is a set, and $\alpha$ is itself a sentence. In addition to these rules, the rules for negation, conjunction, disjunction, implication and equivalence from propositional logic are equally applicable to first-order logic.

- **Predicate:** The sentence $P(x_1, \ldots, x_n)$ is valid if and only if the predicate $P(x_1, \ldots, x_n)$ is valid for the objects associated with the variables $x_1, \ldots, x_n$ in the given context.

- **Existential quantification:** The sentence $\exists x \alpha$ is valid if and only if there exists an object in the domain of discourse such that $\alpha$ is valid when the object is associated with the variable $x$. Also, the sentence $\exists x \in A : \alpha$ is equivalent to $\exists x (x \in A \land \alpha)$ and is valid if and only if there exists an object in $A$ such that $\alpha$ is valid when the object is associated with the variable $x$.

- **Universal quantification:** The sentence $\forall x \alpha$ is valid if and only if for every object in the domain of discourse, $\alpha$ is valid when the object is associated with the variable $x$. Also, the sentence $\forall x \in A : \alpha$ is equivalent to $\forall x (x \in A \Rightarrow \alpha)$ and is valid if and only if for every object in $A$, $\alpha$ is valid when the object is associated with the variable $x$.

In this manner, quantifiers are used to give meaning to variables, and predicates are used to express properties of objects. For instance, the sentence $\forall x \in A : P(x)$ is valid if and only if the predicate $P(a)$ is valid for every object $a$ that is contained in the set $A$. Furthermore, the syntax of predicates is sometimes relaxed to allow more natural forms. For instance, the sentence $\exists x (x^2 = 0)$ is true if there exists an object, in this case a number, such that its square has the value 0. Here, this property may be rewritten as a predicate $P(x)$ that is valid whenever $x^2 = 0$.

Finally, first-order logic is sometimes used to define sets by means of the notation $A = \{ x \mid \alpha \}$, which is equivalent to $\forall x (\alpha \iff x \in A)$ and defines $A$ as the set of all objects such that the sentence $\alpha$ is valid when the variable $x$ is associated to the object.

### Graph Theory

Graph theory [12] studies the concept of graphs, which are one of the most widely used forms of structuring information in computer science. They are used, for instance, to represent file systems, computer programs, communication networks, geographical maps, organisational structures and the behaviour of systems.

A **directed graph** is a pair (or 2-tuple) $(V, A)$, where $V$ is a set of vertices and $A \subseteq V \times V$ is a set of arcs. A graph of this form is said to be directed because the set of arcs $A$ contains pairs of the form $(v, w)$, and they are interpreted as associations, or links, that go in one direction from $v$ to $w$. Directed graphs are often represented graphically by drawing the vertices of $V$ as graphical symbols in a 2-dimensional space and drawing an arrow from $v$
to \( (v, w) \in A \). In turn, an **undirected graph** is a pair \((V, E)\) where \( V \) is a set of vertices and \( E \subseteq \{ S \mid S \in 2^V \land |S| = 2 \} \) is a set of edges. An undirected graph associates its vertices by means of the set of edges \( E \) without specifying a direction for this association: if \( \{v, w\} \in E \), then \( v \) and \( w \) are linked in the graph without a direction because \( \{v, w\} = \{w, v\} \). The graphical representation of undirected graphs is similar to that of directed graphs, but instead of drawing arrows, simple lines are drawn for the edges.

A path in a directed graph \((V, A)\) is a sequence of vertices \( v_1 \ldots v_n \) such that \( (v_i, v_{i+1}) \in A \) for every \( i \in [1, n-1] \). In turn, a path in an undirected graph \((V, E)\) is a sequence of vertices \( v_1 \ldots v_n \) such that \( \{v_i, v_{i+1}\} \in E \) for every \( i \in [1, n-1] \). A **simple path** in a graph is a path where every vertex \( v \in V \) appears at most once. A vertex \( w \) is reachable from a vertex \( v \) in a graph if there exists a path \( v \ldots m \) in the graph. A cycle or loop in a graph is a path \( v \ldots v \) that begins and ends at the same vertex \( v \), and whose length is at least 2. Finally, the notion of paths may be extended to include **infinite paths** of the form \( v \ldots \) that have an initial vertex but no final vertex. If \( V \) is a finite set, an infinite path visits some of the vertices of the graph again and again.

An undirected graph is **connected** if every pair of vertices \( v, w \in V \) are reachable from each other. A **tree**, which is a very common form of graph, is a connected graph that contains no cycles. Trees are also commonly represented by directed graphs, where a single vertex is designated as the root of the tree in such a way that there exists exactly one path from the root vertex to any other vertex in the tree.

**Search Algorithms**

A very common class of algorithms used in computer science is the class of **search algorithms** [1, 9], which represent a generalised way of problem solving [50]. A search algorithm tries to solve a problem by finding a solution for this problem within a set of possible solutions. By this approach, the original problem is restated as a **search problem**, and is then solved by applying a search algorithm to the search problem. The set of possible solutions of a search problem is called the **search space** of the problem, and a search algorithm is always able to recognise a valid solution to the search problem when it sees one. This recognition step is usually simple, and the main challenge of a search algorithm is to explore the search space in an effective and efficient manner.

Many state spaces of common problems can be represented by graphs, where the set of possible solutions corresponds to the set of vertices of the graph. Therefore, the structure of the graph may be used in order to guide the examination of the state space. **Depth-first search** does this by examining an initial vertex — which must be defined — and continuing with another vertex that is directly reachable from it if the solution was not found at this vertex. This procedure is repeated until no more unexplored vertices can be examined, at which point the algorithm “backtracks” to the last vertex that was previously examined and tries to continue from there. The search ends when a solution is found, or when all vertices of the graph have been examined. In turn, **breadth-first search** examines the initial vertex, and then examines each of the vertices that are directly reachable from it if the solution was not found at this vertex. If no solution was found at any of these vertices, then the same procedure is repeated at each of these vertices, where those unexplored vertices that are directly reachable from them are examined. This continues until a solution is found, or until the entire graph is explored. These two algorithms are perhaps the best-known methods to traverse a graph, and both require the same amount of time to complete their operation. The
difference between them lies in the way they operate: throughout the execution, depth-first search needs to remember the path from the initial vertex to the vertex that is currently being explored, whereas breadth-first search needs to keep track of every vertex whose directly reachable vertices are to be explored next. This causes breadth-first search to require a greater amount of storage space. However, breadth-first search has the advantage of always finding the solution vertex that is closest to the initial vertex, that is, where the path between these two vertices is minimal.

3.2.2 Decentralised Systems

Traditionally, computer systems were all centralised, which means that they would operate as individual units that had a certain amount of resources, such as memory and input/output devices, exclusively assigned to them. With the advent of modern communication systems, computer networks were built, and thus, computer systems were now required to interact with each other. At present, practically every computer system is connected to other systems, and is capable of interacting with these systems in a variety of ways. A computer network is essentially a decentralised computer system, where multiple intelligent devices are able to operate independently, and at the same time, interact with each other. The participants of a decentralised system go by many different names: processes, components, agents, nodes, clients, servers and peers.

A distributed system is a decentralised computer system whose participants work cooperatively to reach a common goal. An analogous definition can be given for a distributed program and a distributed algorithm. The concept of a parallel system is very similar to that of a distributed system, and these terms are sometimes treated as synonyms. Both refer to concurrent systems, which are systems whose components are active at the same time. However, parallel systems are usually considered to be those concurrent systems that run in a multiprocessor machine that is capable of executing multiple programs simultaneously, whereas distributed systems are usually considered to be those concurrent systems that run on multiple separate machines simultaneously. Therefore, the components of a parallel system usually communicate with each other by using shared memory, while those of distributed systems usually communicate by a technique known as message passing, which consists of sending and receiving messages. Nevertheless, exceptions to these rules exist, because some parallel systems communicate through messages, and some distributed systems have components that execute on the same physical machine.

The advantages of a decentralised computer system are the same as those of a decentralised process control system, as mentioned in Chapter 2: availability, flexibility, scalability, efficiency and adaptability. Additionally, the goal of providing the communication of information, even across great physical distances, has been a motivation for the use of decentralised computer systems.

3.2.3 Object-oriented Software Engineering

Software engineering is the discipline that studies methodologies for designing and implementing software systems. These systems are implemented by means of programming languages, from which there are an immense variety, both special-purpose and general-purpose. The first programming languages were low-level assembly languages that consist
of sets of simple instructions which are specific to the machine that is being programmed. High-level programming languages were developed in order to allow programs to be written in a style that is closer to natural language, and at the same time, to be independent from the target machine. These languages were implemented either by interpreters, which are programs that run directly in a machine and read and execute other programs, or by compilers, which are programs that read other programs and translate them to a language that can run directly on a machine – usually and assembly language. These techniques were developed in the early days of software engineering, and both are still used in the same manner at present.

Object-oriented programming languages are currently among the most widely-used programming languages. Examples of these languages are Smalltalk, Java, C++, Objective-C, Python, and JavaScript. Programs that are written in these languages consist of definitions of objects, which are units that have a state, also known as attributes, and a behaviour, also known as methods, associated with them. Objects follow the principles of information hiding and encapsulation by hiding their actual state from other objects, and allowing access to their behaviour by exposing their interface, which permits an object to receive messages and execute its corresponding methods. The advantage of this approach is flexibility: objects with different attributes and methods, but which share the same interface, are usable in the same manner in any context that accesses their interface, an abstraction technique known as runtime polymorphism.

In most object-oriented programming languages, objects are defined by classes, which specify the constitution of the state of an object and its behaviour. During execution, a class can be instantiated in order to create corresponding objects that are used as parts of the program. Furthermore, classes may be used to define new classes that extend these by adding attributes or methods, or by redefining existent methods, in a modelling technique known as inheritance.

The methodology of designing and implementing software by following the ideas of object-orientation is called object-oriented software engineering. This methodology consists of several phases: object-oriented analysis, where the different entities and concepts of the application domain are identified and described, together with their responsibilities and the relationships between them; object-oriented design, where an object-oriented model is created to represent the entities, concepts, responsibilities and relationships of the application domain, and finally, the implementation of the software system itself by following the object-oriented model, and ideally using an object-oriented programming language. By bridging the conceptual gap between the application domain and the software domain, object-oriented software development has obtained great popularity in recent years and its use is ubiquitous among software development projects across many different fields of application.

3.3 Railway Control

Railway systems are one of the main forms of transportation for persons and cargo at present. The first railway systems were simple enough to operate without any form of control, but as railway networks became more complex and the traffic of trains increased, it became necessary to adopt control measures in order to avoid accidents, of which the most common are collisions and derailments. Thus, railway control is an important aspect of the engineering of railway systems.

In order to avoid collisions, the principle of train separation is employed, whose goal is to
3.3 Railway Control

maintain an adequate distance between any two trains that travel along the railway network, allowing a train to brake to a full stop before colliding with another train. For this, a method is commonly employed that divides the railway tracks into blocks of an adequate length, and enforces that every block be occupied by at most one train at a time. There are several techniques for implementing this method of train separation, based either on a predefined schedule that the trains must obey or on signalling, where a train must wait for an authorisation signal before entering a given block. Signals are usually implemented by visible lights, similar to common traffic lights, and are given to the train by a human operator, or automatically by a control system, after having verified that the block to be entered by the train is free.

As explained in Chapter 1, a railway network is built from fixed train tracks and switchable elements called points, turnouts or switches, that allow to control the direction of a train that moves over them. Also, it is commonly required that every point must be locked into position while a train moves over it, and cannot be unlocked until the train has moved away, in order to avoid derailments. This locking is accomplished by a safety system known as an interlocking, which enforces not only the correct settings of the points, but also the correct signals given to the trains. Interlockings commonly consist of a table that only permits safe combinations of point settings and signals for the a given region of the railway network under control. Throughout history, interlockings have been implemented first by mechanical, then by electrical, and now by computer-based means, although the general principle of the interlocking table has remained the same.

3.3.1 Geographical Railway Control

The traditional way of implementing interlocking systems by means of tables has the drawback that the table depends on the layout of the track, and any change to this layout forces a redesign of the table. Also, table-based interlockings are inherently centralised systems that operate from an interlocking station, and suffer from the same problems that affect centralised process control systems and centralised computer systems. Therefore, techniques for implementing decentralised interlocking and signalling systems for railway networks were developed. These systems go by the name of geographical interlocking systems or geographical control systems.

In [39], a geographical control system for railway interlocking and signalling is presented which works entirely in a decentralised manner, as opposed to the more common interlocking stations which are centralised and custom-built for the corresponding area of railway tracks that they control. The system consists of distributed control objects which are hardware and software components that communicate with each other via messages. Each control object corresponds to a single track element, which in turn is either a signal, a switch or a track block. The behaviour of each control object depends on the type of the corresponding track element, and the control objects are interconnected analogously to their corresponding elements and may only exchange messages with their direct neighbours through these connections. This allows the system to be easily assembled by following the structure of the railway network. It also makes it easy to adapt and extend the system whenever the railway network changes.

When a train is about to enter a track segment which is guarded by a red signal – which requests the train to stop, the control object of the signal element initiates the process outlined before by sending a clear request to its neighbour in the direction of the intended train
movement. This request is forwarded by each control object in the same direction. When an object which controls a diverging switch receives the request, it forwards it again in the direction of intended movement and at the same time issues a protect request in the other direction. Clear requests are finally answered with clear grants by the control object at the end of the track segment, and protect requests are answered with protect grants either by objects controlling merging switches (which are set in such a way that incoming traffic is deviated), or by objects controlling signals (which are set to red). When each control object receives a grant for every request issued, it then sends a corresponding grant back towards the initial control object. When this object finally receives a clear grant, the corresponding signal is set to green and the train may safely advance. This process locks all switches and signals in place until the train leaves the track segment, in which case the process is reversed and the locks are released.
4 Formal Models

As explained in Chapter 1, the present work aims to develop a technology for the automatic management of product flow paths in processing plants which fulfils the functional requirements outlined in Chapter 2. The current chapter presents a formalisation of the corresponding problem setting, in the form of models which represent a plant and the flow paths it contains in an abstract and precise manner. This formalisation helps in understanding the concrete problem at hand together with its details, and serves as a guideline when developing a corresponding implementation. Furthermore, as the required information about the plant is made explicitly clear, this formalisation serves as a criterion for evaluating the feasibility of applying this technological approach to a given plant.

4.1 Modelling Approach

After taking a close look at the current engineering practices used in the design and construction of processing plants and their control systems (Chapter 3), one finds a gap between the current state of the art in the field of process control engineering and what is needed by an automatic product flow path management system: the access to precise and unambiguous knowledge about the location and direction of product flow in a plant. Further, this knowledge may be seen in two different ways:

1. Offline or static knowledge describes the possible location and direction of product flow in the plant at any given time. An example of this is knowing that a given product at a tank T1 may flow to another tank T2 through a certain piping section whenever an intermediate valve V1 is open.

2. Online or dynamic knowledge comprises the location and direction of product flow in the plant at the current time. Analogously to the previous example, this is knowing whether a given product at the tank T1 is currently flowing to the tank T2 through the intermediate valve V1.

The ways in which a product flow path management system may use this knowledge can also be classified as follows:

1. Passive usage of knowledge about product flow consists of observing and analysing product flow, and is used in product flow path analysis and monitoring, for instance. Both offline and online information about product flow are required for these tasks.

2. Active usage of knowledge about product flow consists of controlling plant devices in order to enforce a given and desired flow of products, or to prevent undesired product flow. This requires knowing where and how a product can flow in a plant, and how this may be controlled by controlling the adequate plant devices. Thus, offline information about product flow plays a major role in deciding how to control this flow, while online
information is useful for confirming the effects of the control. For instance, based on offline information about the plant, the allocation of product flow paths offers information that can be used to control the flow of products along a product flow path in a safe manner.

So how may we, or a product flow path management system, have access to precise and unambiguous knowledge about product flow? At present, plants are usually described by means of diagrams like P&ID which are intended to be interpreted by engineers and are therefore ambiguous in many ways [51]. Most notably, the intended and possible direction of product flow through the various plant elements is usually not specified in these graphical models, as this knowledge is meant to be inferred by the engineers and plant operators that commonly use them [17]. In an actual plant however, the flow of products has determined directions which may be taken. This means that current plant diagrams alone do not offer the aforementioned offline knowledge about product flow in a plant. In order to accomplish the formal representation of this knowledge, an augmented plant meta-model is needed which, apart from the usual information about plant elements and their interconnections, additionally provides precise information about product flow, that is, exactly that information which is commonly inferred by engineers. The abstract plant model and flow allowance model presented in the following sections together constitute such a plant meta-model.

As for the case of online knowledge about product flow, it must be said that as much as we would like to have access to precise and unambiguous knowledge about the current location and direction of product flow in a plant at a given time, this may be difficult to accomplish in today’s processing plants. Flow sensors or product sensors (such as the level sensor in a tank) are not built into every location of the plant, which means that for the most part current information about the position and flow of products would be incomplete. As the goal of this work is to be useful and applicable to current plants, we need to accept this limitation and try to work around it. This has motivated the modelling approach to online information about product flow which is followed in this work: rather than considering real and current information about product flow, this model considers information about the currently possible flow of products in a plant at a given time, which in turn is a necessary condition for actual product flow. In effect, this corresponds to a combination of offline and online knowledge which is more specific than offline information (it considers the current situation in the plant), but less specific than complete online information (it lacks sensor readings for product location and product flow). This knowledge is nevertheless useful and sufficient for accomplishing many tasks regarding product flow paths, as the following sections will show. Most importantly, this knowledge may be computed from information which is commonly available in current processing plants.

Throughout this chapter, the necessary concepts and techniques are defined using a bottom-up approach based on the principle of decomposition, which is commonly used in the field of process control engineering for designing and describing processing plants and process control systems [45]. We start by defining elemental concepts, and gradually build more complex concepts based on them. For instance, the structure and behaviour of a plant is defined in terms of the structure and behaviour of its elements, and likewise, the flow of products through a plant is described in terms of the flow of products through the plant’s individual elements. This allows us to cope with the complexity of modern day plants while regarding the flow of products in sufficient detail.
4.2 Abstract Plant Model

The first thing that we need to determine in order to reason about product flow paths in a plant is where this flow occurs, that is, the location of product flow. This suggests that a map of the plant is needed, that is, some representation of the possible locations in the plant where products may be found as they flow, as well as the directions this flow may take. Such a representation is almost completely given already for any plant by its corresponding plant diagrams, be it P&ID or a similar CAE model. Nevertheless, these representations often carry ambiguities as discussed earlier. Because of this, an abstract plant model is presented here which explicitly defines the possible locations of products in a plant, and the possible movement of products through the plant.

The plant model presented here is abstract because it represents all elements of the plant in a uniform way, considering those aspects which are important for modelling product flow and leaving out any additional details about the nature of the elements. Additionally, it is generic, meaning that any plant element may be directly modelled in it, even novel, unknown or uncommon elements. This model is similar to the RIVA model which will be discussed in Chapter 6, and as presented in Chapter 7, instances of the abstract plant model may in many cases be created for a given plant in a semi-automatic or fully-automatic manner using readily available plant data.

**Definition 1 (Plant structure)** A plant is a tuple \((T, E, C, \tau, \epsilon, \circ)\) where

- \(T\) is a finite set of element types,
- \(E\) is a finite set of plant elements,
- \(C\) is a finite set of connectors,
- \(\tau : E \rightarrow T\) is a function such that \(\tau(e)\) is the type of element \(e\) for every \(e \in E\),
- \(\epsilon : C \rightarrow E\) is a function such that \(\epsilon(c)\) is the element of connector \(c\) for every \(c \in C\),
- \(\circ \subseteq C \times C\) is the connection relation, which is
  - irreflexive: \(c_1 \circ c_2 \Rightarrow c_1 \neq c_2\)
  - symmetric: \(c_1 \circ c_2 \leftrightarrow c_2 \circ c_1\)
  - functional: \((c_1 \circ c_2 \land c_1 \circ c_3) \Rightarrow c_2 = c_3\)

for any \(c_1, c_2, c_3 \in C\). Additionally, for every \(e_1, e_2 \in E\) and \(c_1, c_2, c_3, c_4 \in C\) where \(e_1 = \epsilon(c_1), e_1 = \epsilon(c_2), e_2 = \epsilon(c_3)\) and \(e_2 = \epsilon(c_4)\), it holds that

\[
c_1 \circ c_3 \land c_2 \circ c_4 \Rightarrow c_1 = c_2 \land c_3 = c_4.
\]

For every \(t \in T\), the set \(E_t\) of elements of type \(t\) is defined such that

\[
e \in E_t \Leftrightarrow \tau(e) = t.
\]

For every \(e \in E\), the set \(C_e\) of connectors of element \(e\) is defined such that

\[
c \in C_e \Leftrightarrow \epsilon(c) = e.
\]
A similar definition of the structure of a plant was presented in [46–48]. A plant is formally represented by a tuple \((T, E, C, \tau, \epsilon, \circ)\) with sets of plant element types \(T\), plant elements \(E\) and product connectors \(C\). The set \(T\) contains an enumeration of the distinct types of elements found in the plant, and may be defined according to the application scenario. Every plant element is of a given type, and the function \(\tau\) maps each element to its corresponding type. Elements have one or more connectors which may be used to link them to other plant elements, and the function \(\epsilon\) maps every connector to the plant element which owns it. Finally, the binary relation \(\circ\) represents the interconnection of element connectors as is found in the physical plant: in an exclusive one-to-one manner. The properties of irreflexivity, symmetry and functionality of the relation enforce this interconnection pattern. Figure 4.1 shows the diagram of the sample filling station presented in Chapter 1 and a graphical representation of its corresponding abstract plant model, and Figure 4.2 shows a part of the formulation of this model in mathematical notation.

An additional condition on the connection relation \(\circ\) is included which disallows the multiple interconnection of two elements \(e_1\) and \(e_2\), that is, the use of two distinct product connections between \(c_1\) and \(c_2\). This is done in order to simplify the modelling of flow through the plant, which appears later on in this chapter. In practice, this kind of multiple interconnection of elements should rarely be needed, as the interconnection of plant devices usually occurs in the form of single connections, and the author is unaware of any single case where multiple interconnections of physical plant devices occurs in a plant. If this were the case in a given plant which is to be modelled, an adequate modelling which introduces duplicated elements may be employed in order to work around this limitation.

In accordance to this model, a product may find itself “inside” a given plant element at a given time, which means that the set of elements \(E\) corresponds to the set of possible product locations in the plant. Product connectors are considered interfaces that an element has, and are therefore not seen as product locations proper.

As the reader may have already noticed, a plant \((T, E, C, \tau, \epsilon, \circ)\) represents a special kind of undirected graph where the nodes are the elements and the connectors of the plant. This graph is defined in the following.

\begin{definition}[Plant graph] Given a plant \((T, E, C, \tau, \epsilon, \circ)\), the graph of the plant is an undirected graph \((N, D)\) with a set of nodes \(N = E \cup C\) and a set of edges \(D\) defined for every \(e \in E\) and \(c, c_1, c_2 \in C\) as follows:

\begin{itemize}
  \item \(c_1 \circ c_2 \Rightarrow \{c_1, c_2\} \in D\),
  \item \(e = \epsilon(c) \Rightarrow \{e, c\} \in D\).
\end{itemize}
\end{definition}

As the graph of the plant \((N, D)\) represents the way the plant is structured, we consider that products are able to move from node to node through the edges of this graph: a product may move between an element \(e\) and one of its connectors \(c\), and it may move between a connector \(c_1\) and a connector \(c_2\) if \(c_1 \circ c_2\). However, in a real plant the movement of material is bound to many conditions, and the way in which we model these conditions in this work is the topic of the next section.
Figure 4.1: Diagram of a sample filling station and graphical representation of the corresponding abstract plant model. All elements, including the pipes which transfer material between plant devices, are represented uniformly. Connectors are contained in their corresponding elements; connections occur in an exclusive one-to-one manner.
\[ T = \{ R, T, V, P, J \} \]
\[ E = \{ R_1, \ldots, R_{16}, T_1, \ldots, T_4, V_1, \ldots, V_7, P_1, P_2, J_1, J_2 \} \]
\[ C = \{ R_{1C1}, \ldots, R_{16C1}, R_{1C2}, \ldots, R_{16C2}, T_{1C1}, \ldots, T_{4C1}, \ldots, V_{1C2}, \ldots, V_{7C2}, P_{1C1}, P_{2C1}, P_{1C2}, P_{2C2}, J_{1C1}, \ldots, J_{1C4}, J_{2C1}, \ldots, J_{2C4} \} \]
\[ \tau = \{ (R_1, R), (T_1, T), (V_1, V), \ldots \} \]
\[ \varepsilon = \{ (R_{1C1}, R_1), (R_{1C2}, R_1), (T_{1C1}, T_1), \ldots \} \]
\[ \circ = \{ (T_{1C1}, R_{1C1}), (R_{1C2}, V_{1C1}), (V_{1C2}, R_{5C1}), \ldots \} \]

**Figure 4.2:** Part of the mathematical formulation of the abstract plant model \((T, E, C, \tau, \varepsilon, \circ)\) corresponding to the sample filling station.

## 4.3 Flow Allowance

Having a formal representation of the structure of a plant, we still need to model the flow of products through this plant structure, that is, we need to model how this flow occurs. Before going into detail with this question, we should consider another one which is more fundamental: why does flow occur? Physically, there are several causes for product flow in processing plants: the effect of gravity, the effect of devices like pumps or compressors, differences in pressure caused by heating or by physical or chemical changes, etc. [59]. In all of these cases, a cause of flow makes the product move through the plant; where and how this flow occurs does not depend so much on this cause, but rather on characteristics of the plant itself, such as the shape of the containing elements, the interconnection of the elements, the settings of limiting elements like valves, etc. For instance, when a pump is activated in a plant, material may start flowing through a given piping system. Nevertheless, the pump has no influence on the location and direction of this flow; it is rather determined by the structure of the piping system and the settings of any valves it may contain. This property of determining the location and direction of flow in a passive manner is denoted as **flow allowance** in this work, which contrasts and complements the active role played by an element such as a pump in the movement of products. This is stated in the following conceptual definition.

**Definition 3 (Flow Allowance)** The property of a physical system of permitting material to flow through it is called **flow allowance**.

The model of product flow used throughout this work is based on the concept of flow allowance. It considers the possibility of flow through the elements of the plant, and is collectively called the **flow allowance model** of the plant. This modelling approach to product flow may also be seen as **plant-oriented**, as it considers characteristics of the plant itself instead of considering the actual physical properties of the material which flows. The latter would be the aim of a **material-oriented** model, which is typically used in the simulation of processes [52]. Effectively, the kind of material which flows through the plant does not form a part of the modelling approach presented in this work, which is intended to be applicable to any product which may flow through the plant.

As in the case of the abstract plant model presented earlier, the flow allowance model outlined here is **generic**, which allows it to model the possibility of flow through any element
of a plant, even new or uncommon elements. This modelling power is required in order to guarantee the applicability of this approach to the widest possible range of plants and plant devices.

4.3 Boolean Flow Allowance

In this work, the property of flow allowance is expressed using Boolean values or bits, where 0 represents the inhibition of flow and 1 represents the allowance of flow. Here we only consider this form of binary or qualitative flow allowance, rather than a real or quantitative flow allowance where a richer set of values, such as the real numbers, could be used to express the possibility of flow. The main reasons for this design decision are the following:

- **Simple interpretation.** Any valuation given to a physical characteristic of an object must offer an unambiguous interpretation of its values [20, 59]. As an abstraction of a physical characteristic, the Boolean valuation of flow allowance has a simple and useful interpretation, where 0 stands for “absolutely no flow allowed” and 1 stands for “a positive but unknown amount of flow allowed”. Whereas product flow is usually quantitatively measured in units of volume per time or mass per time [59], Boolean flow allowance only expresses if a nonzero amount of flow is possible, that is, 0 is interpreted as an allowance of a maximum of 0 units of flow, and 1 is interpreted as an allowance of a maximum of $x$ units of flow where $x > 0$. This simple interpretation allows one to easily assess and determine the flow allowance values of a given plant device. For instance, a valve might offer two positions, namely “closed” and “open”, while another valve might offer an aperture setting between 0% and 100%. In any case, a completely closed valve would have a flow allowance value of 0, and any other position would indicate a flow allowance value of 1. This offers an abstraction over the state of the valve which equates all states where the valve is not completely closed, and differentiates them from those states where the valve is completely closed.

- **Simple combination of values.** The combination of Boolean flow allowance values is straightforward: a connection of two flow allowance values in series corresponds to the conjunction (or the Boolean and operation $\land$) of the values, as the resulting flow allowance is 1 only if both connected values are 1; likewise, a connection of two flow allowance values in parallel corresponds to the disjunction (or Boolean or operation $\lor$) of the values, because the resulting flow allowance is 0 only if both connected values are 0. An analogy to the parallel and serial connection of electrical switches may be drawn here.

- **Application goals.** The previous points clarify the simplicity of the Boolean valuation of flow allowance in terms of semantics, yet the question remains: Is this abstraction useful? The product flow path management technology presented in this work must execute the tasks of discovery, safety monitoring and safety locking of product flow paths. For all these functions, it is always important for the system to determine if flow is possible or not, in given locations and directions, at given situations in the plant. For these applications, Boolean flow allowance suffices. Even more so, the Boolean valuation of flow allowance provides an effective abstraction of the plant situation which benefits the simplicity of the algorithms used by a product flow path management system.
A quantitative form of flow allowance can be imagined, where the quantitative flow allowance values would describe the maximal amount of flow which may occur in each element in the plant. Furthermore, the combination of such quantitative flow allowance values could be accomplished using the \textit{min} operator for combining values connected in series, and the + operator for combining values connected in parallel. However, quantitative flow allowance is not required for solving the problems which regard product flow paths that are outlined in Chapter 2, and is therefore considered to be beyond the scope of this work.

4.3.2 Flow Allowance Settings

If we consider the property of flow allowance for the case of processing plants, we quickly become aware the following facts, some of which may already be perceived by the reader:

1. \textit{The flow allowance of a plant is compositional}. In other words, the flow allowance of a plant is defined by the flow allowance of its individual components. For instance, if a given piping section in a plant contains one or more valves, the flow allowance of the piping section as a whole depends on the settings of these valves, that is, on the flow allowance of each of the individual valves. In a strictly serial configuration, the closing of a single valve is enough to block the flow of material through the entire piping section.

2. \textit{The flow allowance of a plant is dynamic}. Plants usually contain elements like valves which may be opened or closed in order to control the flow of material. This leads to the conclusion that the flow allowance of a plant is in general dynamic, meaning that it may change during the operation of the plant.

3. \textit{The flow allowance of a plant is directed}. Some plant elements allow flow to occur in certain directions, while disallowing flow in other directions. For instance, a holding valve is a device with two product connection points which allow flow through it in one direction only. A tank with a product input at the top and a product output at the bottom is another example of this. This means that when regarding flow allowance, we must also consider the direction of the flow which is allowed.

These observations motivate the flow allowance model which is presented in this work. Because of the compositional property, it explicitly models the flow allowance of each individual plant element, which implicitly models the flow allowance of the entire plant. Because of the dynamic property, it models the flow allowance of a plant in terms of flow allowance settings which may be changed during the operation of the plant. Because of the directed property, it models the allowance of flow in each of the different directions through an element. The latter is accomplished by considering the element as a “product room” where all contained products may mix freely, and by defining the possibility of entering and exiting this element through each of its product connectors, which may be seen as the “product doors” of this room. Figure 4.3 shows this idea in graphical form. A product door may be used for entering or exiting the room exclusively, or may be used simultaneously as an entrance and as an exit. Furthermore, some doors may be opened and closed in accordance with the dynamic property of flow allowance, and this may be done for the entrance and exit aspects independently. These are the ideas behind the following definition.

\textbf{Definition 4 (Flow allowance setting)} \textit{Given a plant} $(T, E, C, \tau, e, o)$ \textit{and an element} $e \in E$, \textit{a function} $\phi_e : C_e \to \{0, 1\} \times \{0, 1\}$ \textit{is a flow allowance setting of} $e$. \textit{For a given} $\phi_e$ \textit{and} $c \in C_e$...
4.3 Flow Allowance

A flow allowance setting \( \phi_e(c) = (i, o) \), we denote \( i \) as the input flow allowance setting of \( c \) at \( \phi_e \), and \( o \) as the output flow allowance setting of \( c \) at \( \phi_e \). For a given \( \phi_e \), the function \( \phi^I_e : C_e \rightarrow \{0, 1\} \) denotes the input flow allowance setting from \( \phi_e \) as follows:

\[
\phi_e(c) = (i, o) \Rightarrow \phi^I_e(c) = i.
\]

For a given \( \phi_e \), the function \( \phi^O_e : C_e \rightarrow \{0, 1\} \) denotes the output flow allowance setting from \( \phi_e \) as follows:

\[
\phi_e(c) = (i, o) \Rightarrow \phi^O_e(c) = o.
\]

A flow allowance setting \( \phi_e \) of an element \( e \) corresponds to an assignment of a pair of Boolean values to every connector \( c \) of \( e \). The first of these values represents the allowance of input flow through this connector and into the element; consequently, the second value represents the allowance of output flow through this connector and out of the element. These values are individually given by the functions \( \phi^I_e \) and \( \phi^O_e \) respectively, as shown in Figure 4.4. By specifying the input and output flow allowance at every connector of the element \( e \), a flow allowance setting \( \phi_e \) describes the flow allowance of \( e \) at a given time, where a given control state of the element is in effect. Figure 4.5 shows some examples of flow allowance settings for several plant elements.

Most common plant devices like pipes, valves, pumps, tanks, reactors, etc., have a single interior room where all products that enter the device are able to mix. Therefore, all these devices may be directly modelled by a plant element \( e \in E \) as shown in Figure 4.3. However, some plant devices, most notably heat exchangers [59], have two or more product cavities that are isolated from each other, and therefore permit the flow of two or more products through them without causing them to mix. For these devices, a different modelling technique is needed, which models every internal product cavity as a unique element \( e \in E \), effectively requiring more than one element to model a single plant device. In the RIVA model (see Chapter 6), plant devices are modelled by units, and their product rooms by channels. The

Figure 4.3: A plant element \( e \) is modelled as a “product room” that may be entered and exited through its connectors \( c_1 \ldots c_n \), which are seen as “product doors”. Products may mix freely inside the element.
**4 Formal Models**

**Figure 4.4:** Flow allowance setting of a connector \( c \) of an element \( e \): \( \phi^I_e(c) \) refers to the allowance of flow which enters \( e \) through \( c \), and \( \phi^O_e(c) \) refers to the allowance of flow which leaves \( e \) through \( c \).

**Figure 4.5:** Flow allowance settings for various types of plant elements. A pipe is an element with two connectors which allow flow in both directions. A valve is a 2-connector element whose connectors may allow or inhibit flow in both directions; pictured is the flow allowance setting of a closed valve. A holding valve is a 2-connector element where one connector allows incoming flow only while the other allows outgoing flow only. Finally, a 3-way pipe join is a 3-connector element whose connectors allow flow in both directions.

<table>
<thead>
<tr>
<th>Element</th>
<th>Connectors</th>
<th>( \phi_e(c_1) )</th>
<th>( \phi_e(c_2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipe</td>
<td>( c_1 )</td>
<td>( 1, 1 )</td>
<td>( 1, 1 )</td>
</tr>
<tr>
<td>Valve (closed)</td>
<td>( c_1 )</td>
<td>( 0, 0 )</td>
<td>( 0, 0 )</td>
</tr>
<tr>
<td>Holding Valve</td>
<td>( c_1 )</td>
<td>( 1, 0 )</td>
<td>( 0, 1 )</td>
</tr>
<tr>
<td>Join</td>
<td>( c_1 )</td>
<td>( 1, 1 )</td>
<td>( 1, 1 )</td>
</tr>
</tbody>
</table>
4.3 Flow Allowance

latter are the semantic equivalent of the plant elements of the abstract plant model presented in this work.

4.3.3 Flow Allowance Model

In practise, many elements such as pipes and tanks are static in nature and only have one possible flow allowance setting associated with them. However, other plant elements may have multiple flow allowance settings, as is usually the case with valves and pumps. A flow allowance model of an element $e$ defines the possible flow allowance settings that $e$ may use. In turn, the flow allowance model of a plant combines the flow allowance models of all of the plant’s constituent elements in order to describe the flow allowance of the entire plant. This is formalised in the following definition.

**Definition 5 (Flow allowance model)** Given a plant $(T, E, C, \tau, \epsilon, \circ)$ and an element $e \in E$, a flow allowance model of $e$ is a nonempty set of functions $F_e \subseteq [C_e \rightarrow \{0, 1\} \times \{0, 1\}]$ where every $\phi_e \in F_e$ is a flow allowance setting of $e$. A flow allowance model of the plant is a function $\Phi : E \rightarrow 2^{[C \rightarrow \{0, 1\} \times \{0, 1\}]}$ where $\Phi(e)$ is a flow allowance model of $e$ for every $e \in E$.

A flow allowance model $F_e$ of an element $e$ provides a complete description of the way in which the plant device that corresponds to $e$ permits the flow of material through it. Devices that are strictly static have exactly one flow allowance setting associated with them, which means that the corresponding flow allowance model $F_e$ is a singleton; devices with multiple flow allowance settings have corresponding flow allowance models which contain all of these settings. In any case, a flow allowance model of an element may never be empty, because otherwise the flow allowance of the element would be undefined. The flow allowance model $\Phi$ of a plant is basically a mapping which provides a flow allowance model $F_e$ for every element $e$ contained in the plant.

The plant structure model $(T, E, C, \tau, \epsilon, \circ)$ and the flow allowance model $\Phi$ of the plant together constitute the complete augmented plant meta-model which is required in order to apply the product flow path management approach presented in this work. That is, properly instantiated $(T, E, C, \tau, \epsilon, \circ)$ and $\Phi$ must be provided beforehand as a description of the plant. Chapter 7 details how this information is obtained and used by a prototypical implementation of a product flow path management system.

4.3.4 Flow Allowance States

When we define a flow allowance model $\Phi$ for a given plant, we determine the possible flow allowance settings that each of the elements of the plant may use. Moreover, the possible situations that the plant may find itself in with respect to flow allowance are also implicitly defined, where such a situation corresponds to the assignment of a valid flow allowance setting $\phi_e$ to every element $e$. These situations are called flow allowance states of the plant, and are defined in the following.

**Definition 6 (Flow allowance state)** Given a plant $(T, E, C, \tau, \epsilon, \circ)$ and a corresponding flow allowance model $\Phi$ of the plant, a flow allowance state of the plant is a function $\sigma : E \rightarrow [C \rightarrow \{0, 1\} \times \{0, 1\}]$ where $\sigma(e) \in \Phi(e)$ for every $e \in E$. 
At any time during the operation of the plant, a product flow path management system may determine the flow allowance state $\sigma$ of the plant by determining the flow allowance setting $\phi_e$ of each plant element $e$ which is currently in use. This is trivial in the case of static devices, and for dynamic devices, the process control system must provide some way of knowing the current working state of the corresponding plant device, and the flow allowance setting $\phi_e$ may be then determined based on this information. For instance, control valves usually have associated check-back signals which indicate if the valve is closed or open. A mapping which associates signal values to flow allowance settings is all that is needed by the system in order to determine the current flow allowance setting $\phi_e$ of the element $e$ during operation. Chapter 7 presents some techniques for achieving this in practice.

4.4 Flow Routes

The main intention of the modelling approach presented in this chapter is to ultimately derive a formal model for product flow paths in processing plants. The abstract plant model and the flow allowance model which have been outlined enable us to reason about product flow in a way which suits the definition of such a flow path model. From the requirements of this work, presented in Chapter 2, we know that flow paths are to be represented as objects in a process control system, and managed as units. Intuitively, a flow path must refer to a substructure of the plant, which contains exactly those plant elements that are considered to be a part of the path. In other words, a flow path corresponds to a section of the plant which is a route that may be used by a product to flow through the plant. Consequently, we denote this plant section as a flow route, and distinguish it from the actual flow path object which is related to it and which manages the product flow path and executes its corresponding operations.

4.4.1 The Structure of Flow Routes

The flow of products in a plant can be very complex in practice. Depending on the structure of the plant, products may be sent along diverging routes, rejoined and mixed at certain points in the plant, recirculated, etc. Thus, modelling the actual flow of products in a plant could be achieved by representing this flow as traversals of the graph $(N, D)$ if the plant. Using this approach, the flow of a certain product would be represented by a sub-graph of this graph which contains the entire area of the plant where the product flows. However, in this work the routes of product flow are modelled in a simpler way: as sequential paths through the plant. There are several reasons for this design decision, and they are explained in the following:

1. Reduced complexity. Sequential paths are simpler than arbitrarily complex flow graphs. This means that the data structures which are needed for representing these paths and the algorithms which operate on them are simpler as well. This reduction in complexity is beneficial and should be considered as an advantage of this modelling approach.

2. Comprehensibility. Engineers, plant operators and other persons that need to understand product flow in a plant and discuss about it, find it easier to speak about simple paths than about more complex flow structures. Even if such structures are present in a plant, they are usually broken down mentally into sequential routes, and each route is regarded as an entity. Therefore, sequential paths of flow are structures which are easy to understand and to work with.
3. **Expressiveness.** As suggested by the previous point, complex flow structures with branches, joins and cycles may always be broken down into a collection of sequential flow routes by splitting the graph that represents the complex flow structure at every point where the structure branches or joins. Therefore, sequential flow routes may also be used collectively in this manner to represent arbitrarily complex flow through a plant. This approach, along with its algorithms, are not a part of this work and considered to be a future research direction. Nevertheless, it is important to realise that the treatment of complex flow structures is not a limitation, but rather an extension of this work.

4. **Applicability.** Although complex flow structures are sometimes found in today’s processing plants, it is usually the case that flow operations occur along simple, sequential routes which connect an initial plant element with a final plant element. Therefore, the product flow path model presented in this work should be directly applicable to most scenarios where products flow in a plant.

5. **Intention.** From the analogy of train routing along a railway system brought forward in Chapter 1, the concept of a flow path is a new way to specify a flow operation in a plant, just like a train path specifies the way in which the train has to travel. This specification, which expresses the intention of the transport operation, additionally determines the conditions for safe travel along the track system, respectively for safe flow through the plant. In both train control and in this work, this specification takes the form of a simple, sequential path through the transportation network, because it is the most effective way to move between two points of this network. In this manner, our model of product flow paths aims to introduce a new way of specifying flow operations that is simple because of the reasons provided above, and which is supported by a model that helps to assert the correctness and safety of the flow of material.

Because of these reasons, by modelling flow routes as simple paths through the plant we achieve a useful model that is accurate and comfortable to work with while at the same time expressive enough to handle both simple and complex plant designs.

At first look, flow routes correspond to paths through the graph $(N, D)$ of the plant. However, the flow allowance model $\Phi$ of the plant determines the ways in which products may flow through the various plant elements. Therefore, the flow allowance model imposes restrictions on the flow routes of the plant. Flow routes are thus given by paths through the graph of the plant which are permitted by the flow allowance model of the plant. We will now look at this idea in more detail.

### 4.4.2 Flow Steps

A flow allowance setting $\phi_e$ of an element $e$ tells us the ways in which material may flow within $e$ whenever $\phi_e$ is in effect. As a product flows through the plant, it moves through several plant elements according to the respective flow allowance setting of each of the elements. Because of this, we first define the possible ways in which a product may flow within each plant element by defining the set of *flow steps* of the plant as a set of sequences of elements and connectors. Later on, we may use this definition to build up flow routes by joining several flow steps, one after the other.
4 Formal Models

![Diagram](image)

\[ \exists \phi_e \in \Phi(e) : \phi_e^O(c) = 1 \quad \exists \phi_e \in \Phi(e) : [\phi_e^I(c_1) = 1, \phi_e^O(c_2) = 1] \quad \exists \phi_e \in \Phi(e) : \phi_e^I(c) = 1 \]

**Figure 4.6:** Scenarios of an initial flow step (left), middle flow step (centre) and final flow step (right).

**Definition 7 (Flow steps)** Given a plant \((T, E, C, \tau, e, \circ)\) and a corresponding flow allowance model \(\Phi\), the sets of initial flow steps \(S^I \subset E \cdot C\), middle flow steps \(S^M \subset C \cdot E \cdot C\) and final flow steps \(S^F \subset C \cdot E\) of the plant are defined for every \(e \in E\) and \(c, c_1, c_2 \in C\) as follows:

\[
S^I = \{ ec \mid e(c) = e, \exists \phi_e \in \Phi(e) : \phi_e^O(c) = 1 \},
\]

\[
S^M = \{ c_1c_2 \mid e(c_1) = e, e(c_2) = e, c_1 \neq c_2, \exists \phi_e \in \Phi(e) : [\phi_e^I(c_1) = 1, \phi_e^O(c_2) = 1] \},
\]

\[
S^F = \{ ce \mid e(c) = e, \exists \phi_e \in \Phi(e) : \phi_e^I(c) = 1 \}.
\]

The set of flow steps of the plant is defined as \(S = (S^I \cup S^M \cup S^F)\).

Flow steps are classified as initial, middle and final flow steps. These attributes refer to the position of each flow step within a complete flow route, and also correspond to the three possible scenarios of sequential product flow within an element which are shown in Figure 4.6. Initial flow steps represent the flow of a product from the interior of an element \(e\) to one of its connectors \(c\). For this to occur, the output flow allowance setting \(\phi_e^O(c)\) of the connector \(c\) must have the value 1, which means that a product may leave \(e\) through \(c\). By this definition, the sequence \(ec\) is considered a flow step if there exists an flow setting \(\phi_e\) in the flow allowance model \(\Phi(e)\) of the element \(e\) where this is the case. This tells us that flow steps represent possible flow allowance, and that this possibility depends on the flow allowance state \(\sigma\) of the plant. Clearly, sequences for whom this condition does not hold are not considered flow steps because product flow will never be possible in this way. Middle flow steps represent the flow of a product that enters an element \(e\) through one of its connectors \(c_1\) and leaves \(e\) through another one of its connectors \(c_2\), and for this to occur, there must exist a flow allowance setting \(\phi_e\) in the flow allowance model \(\Phi(e)\) of the element \(e\) where both the input flow allowance setting \(\phi_e^I(c_1)\) of the connector \(c_1\) and the output flow allowance setting \(\phi_e^O(c_2)\) of the connector \(c_2\) have the value 1. Lastly, final flow steps represent the flow of a product that enters an element \(e\) through one of its connectors \(c\) and reaches the interior of \(e\). Consequently, for this to occur there must exist a flow allowance setting \(\phi_e\) in the flow allowance model \(\Phi(e)\) of the element \(e\) where the input flow allowance setting \(\phi_e^I(c)\) of the connector \(c\) has the value 1. In this way, the set \(S\) contains all possible flow steps which may be used to construct flow routes in the plant.

4.4.3 Flow Routes

Building upon our definition of flow steps, we can now formulate the concept of a flow route of the plant. When we defined the abstract plant model \((T, E, C, \tau, e, \circ)\), it was determined that
the possible locations of products in the plant correspond to the set $E$ of elements of the plant. Also, we have stated that a flow route is given by a sequential path through the graph $(N,D)$ of the plant which is permitted by the flow allowance model $\Phi$ of the plant, which means that a flow route must correspond to a succession of flow steps. Putting it all together, this implies that a flow route must always begin at an element $e_1$, the initial element of the route, and end at a final element $e_n$, which additionally tells us that a flow route must begin with an initial flow step, and end with a final flow step. Thus, the meaning of a flow route is the ability to move material from $e_1$, represented by the initial flow step, to $e_n$, represented by the final flow step, along a sequence of intermediate elements, represented by middle flow steps.

We define the concept of a flow route in the plant with the help of another concept, a flow trace. A flow trace is essentially a flow route at level of greater detail: flow traces are sequences of flow steps that include both the plant elements which are traversed and the corresponding connectors that are used to transfer material among the elements. Flow routes are abstract or summarised flow traces. From a given flow trace, a flow route which only contains the elements of the corresponding flow trace may be obtained by leaving out the connectors. This is defined in the following.

**Definition 8 (Flow traces)** Given a plant $(T,E,C,\tau,\epsilon,\circ)$ and a corresponding flow allowance model $\Phi$, the set of flow traces $A \subset S^I \cdot S^{M^*} \cdot S^F$ of the plant is defined with the help of the set $\tilde{A} \subset S^I \cdot S^{M^*}$ as follows:

\[
\tilde{A} = \{ s \mid s \in S^I \} \cup \{ tc_1e_2 \mid t \in \tilde{A}, t = \ldots c, c_1e_2 \in S^M, c \circ c_1, a(t) \cap a(c_1e_2) = \emptyset \},
\]

\[
A = \{ tc_1e \mid t \in \tilde{A}, t = \ldots c, c_1e \in S^F, c \circ c_1, a(t) \cap a(c_1e) = \emptyset \}.
\]

The set of flow traces $A$ is defined with the help of the set $\tilde{A}$ of incomplete flow traces, which is recursively defined according to the following ideas: all initial flow steps are incomplete flow traces; also, a new incomplete flow trace may be formed by appending a middle flow step to an incomplete flow trace, if the incomplete flow trace ends with a connector $c$ that is connected to the first connector $c_1$ of the middle flow step. A flow trace may then be formed by appending a final flow step to an incomplete flow trace, if the incomplete flow step again ends with a connector $c$ that is connected to the first connector $c_1$ of the final flow step. All these append operations are done only if the operands to be joined have disjunct alphabets, which means that every incomplete and complete flow trace is free from any repeated items. Because of this, and because the plant has a finite size, it is guaranteed that every flow trace is finite. Furthermore, this also guarantees that the sets $\tilde{A}$ and $A$ are themselves finite. There are additional reasons for why it is important that flow traces have no repeated items, and these reasons will be discussed later on in this chapter. In this manner, the complete set $A$ of flow traces of the plant is obtained.

**Definition 9 (Flow routes)** Given a plant $(T,E,C,\tau,\epsilon,\circ)$ and a corresponding flow allowance model $\Phi$, the set of flow routes $R \subset E^+$ of the plant is defined as the projection of the set of flow traces $A$ of the plant with respect to $E$ as follows:

\[
R = \{ \pi_E(t) \mid t \in A \}.
\]
Flow routes are thus determined by their corresponding flow traces, which means that in order to tell if an arbitrary sequence of plant elements $e_1 \ldots e_n$ is in fact a flow route, we must first find a corresponding flow trace of the form $e_1c_1e_2c_2 \ldots c_ne_n$ as shown in Figure 4.7. Flow traces are important in our model because they represent the possibility of flow through the plant in a precise manner. In turn, flow routes are of importance for practical reasons. In a real plant, actual flow traces are usually too long and too complex to be used effectively in user interfaces. Engineers and plant operators will usually refer to the elements in the plant, and not to the corresponding connection points between the elements because they are obviated. Product flow is thereby described as a series of “hops” from element to element, and this is exactly what a flow route describes. Therefore, we use flow routes when interacting with the user and as a description of the section of the plant that a product flow path uses for the transport of material, and use the corresponding flow traces internally in the related algorithms.

A flow trace $t$ fully determines a flow route $r$, but does the opposite also hold? That is, can a flow route $r$ have more than one flow trace associated to it? By looking at the definition of flow traces, we see that this can occur whenever products may flow from an element $e_1$ to an element $e_2$ through more than one possible connection, that is, that the elements $e_1$ and $e_2$ are connected to each other more than once. As discussed earlier when defining the abstract plant model $(T, E, C, \tau, \epsilon, \circ)$, this should rarely happen in practise, and the abstract plant model prohibits this form of duplicate interconnection of the elements. Therefore, under these conditions it is correct to assume that there is a one-to-one relation between flow routes and flow traces.

Because flow traces are free from any repeated items, this property also holds for flow routes. The restriction of disallowing repeated items in flow routes is important and has been included in the definition of flow traces and flow routes for the following reasons:

1. **Absence of cycles.** Disallowing repeated elements in a flow route causes flow routes to be free of cycles, and therefore, guarantees that flow routes have a finite length (given that the plant is itself finite). This simplifies the representation of flow routes as data structures and the algorithms which work with these flow routes.

2. **Avoidance of simultaneous bidirectional flow.** Allowing repeated elements in a flow route would permit routes of the form $\ldots e_1e_2 \ldots e_2e_1 \ldots$, which would model simultaneous bidirectional flow between the elements $e_1$ and $e_2$ when the flow route is in use. Physically, this type of flow could occur if some type of circulation of material would take
4.4 Flow Routes

place between the elements \( e_1 \) and \( e_2 \) through a single product connection [20]. However, this phenomenon is not considered in this work, because in process engineering it is commonly assumed that material flows always in a single direction at a time [59], effectively neglecting small turbulences and circulations in the products that flow. Thus, we rule out any form of simultaneous bidirectional flow from our model. A flow route where no element appears twice represents a form of "forward only" flow, which best describes the intended and assumed flow of products in processing plants.

With the definition of the set \( R \) of flow routes of the plant, we obtain a formal framework for describing the regions in a plant where products flow in a simple, precise and practical manner. This will allow us to formulate many useful tasks which are important for a product flow path management system.

4.4.4 Open Flow Routes

Flow routes are defined in terms of flow traces, which are themselves defined in terms of flow steps. Because flow steps may or may not allow flow to occur along them at a given time during the operation of the plant (depending on the flow allowance setting which is currently in use by the corresponding plant element), we may generalise this observation and state that a flow route may or may not allow flow to occur along itself at a given time. The relationship between the flow allowance of flow steps and flow routes is simple: a flow route allows flow if and only if every single flow step which is contained in its corresponding flow trace allows flow. An illustrative example is a straight piping section that contains many valves. It is clear that the piping section allows flow, or is said to be open, only if every single valve it contains is also open. We therefore call a flow route that allows flow an open flow route. In general, a flow route may or may not be open depending on the flow allowance state \( \sigma \) of the plant, and because of this, we define the set of open flow routes of the plant with respect to a flow allowance state \( \sigma \).

In analogy to the definitions of flow steps and flow traces which enabled the formulation of flow routes, the concepts of open flow steps and open flow traces allow us to give a definition of open flow routes as follows.

**Definition 10 (Open flow steps)** Given a plant \((T, E, C, \tau, e, o)\), a corresponding flow allowance model \(\Phi\) and a flow allowance state \(\sigma\) of the plant, the sets of open initial flow steps \(S^I_\sigma \subseteq S^I\), open middle flow steps \(S^M_\sigma \subseteq S^M\) and open final flow steps \(S^F_\sigma \subseteq S^F\) of the plant at \(\sigma\) are defined for every \(e \in E\) and \(c, c_1, c_2 \in C\) as

\[
S^I_\sigma = \{ ec \mid ec \in S^I, \phi_e^O(c) = 1 \},
\]

\[
S^M_\sigma = \{ c_1ec_2 \mid c_1ec_2 \in S^M, \phi_e^I(c_1) = 1, \phi_e^O(c_2) = 1 \},
\]

\[
S^F_\sigma = \{ ce \mid ce \in S^F, \phi_e^I(c) = 1 \}
\]

where \(\sigma(e) = \phi_e\). The set of open flow steps of the plant at a flow allowance state \(\sigma\) is defined as \(S_\sigma = (S^I_\sigma \cup S^M_\sigma \cup S^F_\sigma)\).

The open flow steps at a flow allowance state \(\sigma\) are those flow steps of the plant whose elements have flow allowance settings at \(\sigma\) with the appropriate values for the input and output allowance of flow in accordance with the type and direction of the flow step. Flow steps not in \(S_\sigma\) are considered to be closed at \(\sigma\).
Definition 11 (Open flow traces) Given a plant \((T, E, C, \tau, e, \circ)\), a corresponding flow allowance model \(\Phi\) and a flow allowance state \(\sigma\), the set of open flow traces \(A_\sigma \subseteq A\) of the plant at \(\sigma\) is defined as

\[
A_\sigma = \{ t \mid t \in A, t \in S^L_\sigma \cdot S^M_\sigma \cdot S^F_\sigma \}.
\]

The concept of open flow steps allows the definition of open flow traces to be very simple: the open flow traces at the flow allowance state \(\sigma\) are those flow traces of the plant which are composed exclusively of open flow steps at \(\sigma\). Again, flow traces not in \(A_\sigma\) are considered to be closed at \(\sigma\).

Definition 12 (Open flow routes) Given a plant \((T, E, C, \tau, e, \circ)\), a corresponding flow allowance model \(\Phi\) and a flow allowance state \(\sigma\), the set of open flow routes \(R_\sigma \subseteq R\) of the plant at \(\sigma\) is defined as the projection of the set of open flow traces \(A_\sigma\) of the plant with respect to \(E\) as follows:

\[
R_\sigma = \{ \pi_E(t) \mid t \in A_\sigma \}.
\]

An open flow route at the flow allowance state \(\sigma\) is then the corresponding flow route of an open flow trace at \(\sigma\). Also, a flow route is closed at \(\sigma\) if it is not in \(R_\sigma\). These definitions suggest a simple technique for determining if a flow route in the plant is open at a given time: by identifying every flow step in the corresponding flow trace of the route, and testing the required values in the current flow allowance setting of each element involved. The usage of this idea for the monitoring of product flow paths will be shown later on in this chapter.

4.5 Enclosed Flow Routes

So far we have considered the flow of products along flow routes in a plant, and have assumed that products “stay on track” as they flow. Clearly, in real plants the flow of products may reach structures such as pipe joins which can cause this movement of material to branch out in more than one direction. Furthermore, the flow of products outside of a given flow route could reach an element in the flow route if the structure of the plant and its current flow allowance state permit it. One of the desired characteristics of a product flow path in a plant, as discussed in Chapter 2, is that it represents a safe and exclusive area for the transport of products. Thus, situations like these where products may branch out or mix along a flow route should be avoided.

From the analogy of railway control presented in Chapter 1, a product moving along a flow path should be protected from deviations and from the flow of other products just like a train moving along a route is protected from deviations and approaching trains. In the following, a model of enclosed flow routes is presented, which supports the definition of this kind of protection for flow routes. Based on this model, a product flow path management system can implement algorithms which monitor the protection of flow paths, and which offer a locking mechanism that aims to guarantee this protection.

4.5.1 Product Leaks and Product Mixtures

We previously defined open flow routes as those flow routes which permit flow along them at the current flow allowance state of the plant. The concept of enclosed flow routes is an
additional criterion for describing the state of flow routes in a plant, and is defined orthogonally to the concept of open flow routes. In an enclosed flow route, products that are able to flow along the route are not able to deviate from the route and leave it. At the same time, other products are not able to flow into the flow route and mix with the product flowing along the route. Thanks to these two properties, an enclosed flow route effectively isolates the products within the route from other products which are found outside of the route. These two properties are, respectively, the avoidance of product leaks and product mixtures along the flow route.

The notion of safety in processing plants is a very broad and important part of the field of process control engineering [45]. Based on the abstract plant model and the flow allowance model, the definition of enclosed flow routes at a given flow allowance state has the intention of identifying general scenarios which correspond to undesired and potentially hazardous situations that may arise during the usage of a product flow path. This model-based approach defines an aspect of safety which regards the avoidance of these particular situations.

The analogy of railway control from Chapter 1 helps to identify the occurrence of product leaks and product mixtures in a plant, and to develop ways to prevent them. In a railway network, deviations occur when a train moves over a switch which is set to the wrong position, which causes it to enter a wrong track segment and to deviate from its intended course; collisions occur when an approaching train enters a track segment already in use by another train. We may characterise product leaks and mixtures in a plant with the help of these ideas:

- **Product leaks.** A product leak is similar to a train deviation, and occurs when the flow of a product along a route diverges from the intended route \( r_1 \). In our model, this occurs when there exists a diverging open flow route \( r_2 \) which begins at an element in \( r_1 \).

- **Product mixtures.** A product mixture is similar to a train collision, and occurs when another material is able to flow into, and mix with, the product flowing through a flow route \( r_1 \). In our model, this occurs when there exists a joining open flow route \( r_2 \) which ends at an element in \( r_1 \).

In both of these cases, the additional flow route \( r_2 \) which leaves or joins the flow route \( r_1 \) is called a branch flow route. Figure 4.8 shows a flow route and a corresponding branch flow route which stems out from it. In order for the flow route to be enclosed, it must be free from product leaks and mixtures, which means that both incoming and outgoing flow to and from the route must be blocked by the elements in every neighbouring branch flow route. However, several questions may be asked at this point: How long is a branch flow route? That is, where does a branch flow route end when it represents a leak, and where does a branch flow route start when it represents a mixture? How may we ensure that this blocking is effective? In order to shed some light into this matter, consider a tank which appears at some point in the branch flow route. If flow between the flow route and the tank is not blocked by an element in between both of them, then we should not consider the route as enclosed, because the product flowing in the route may flow into the tank, and any product in the tank could flow into the route. However, if the branch flow route consists only of pipes and a valve at the end, then we could consider the route as enclosed whenever this valve is closed, and no matter how long the branch flow route is. This suggests that the concept of enclosed flow routes depends on the semantics of the neighbouring elements. Concretely, we need to determine which plant elements are considered “dangerous” and should not be found in an open branch flow route of an enclosed flow route.
In order to complete our notion of product leaks and mixtures, we first identify two subclasses of plant elements: *product sources* and *product sinks*. The former are elements which can yield material that may flow to other points in the plant, such as tanks, reactors or input nozzles; the latter are elements which can consume material which flows from other points in the plant, such as tanks, reactors or output nozzles.

**Definition 13 (Product sources and product sinks)** Given a plant \((T, E, C, \tau, e, o)\), the set \(E^\uparrow \subseteq E\) is the set of *product sources* of the plant, and the set \(E^\downarrow \subseteq E\) is the set of *product sinks* of the plant.

A similar classification which describes the general function of an element with respect to product flow may be found in [33]. We assume that the classification of elements as product sources or product sinks is given. In practise, creating this classification usually represents an easy task, whereby a few elements such as tanks and input and output nozzles may be “marked” by engineers as either product sources and/or product sinks. It may also be possible to determine this classification based on the types of the elements as given by the function \(\tau\) of the abstract plant model. By means of this classification of the elements in the plant, we characterise a relevant aspect of the semantics of the elements in an abstract and flexible manner.

We may now restate our definition of product leaks and product mixtures using the sets of product sources and sinks in the plant and with the help of Figure 4.8:

- A product leak occurs in a flow route \(r_1 = e_1 \ldots e_i \ldots e_n\), and a corresponding branch flow route \(r_2 = \ldots e_j \ldots\). Product flow is intended along the flow route from \(e_1\) to \(e_n\). Additional outgoing flow from \(e_i\) to the branch flow route represents a product leak. Likewise, additional incoming flow to \(e_i\) from the branch flow route represents a product mixture.

- A product mixture occurs in a flow route \(r_1 = e_1 \ldots e_i \ldots e_n\) when there exists a diverging open flow route \(r_2\) which begins at an element \(e_i\) in \(r_1\) and ends at a product sink element in \(E^\downarrow\).

- A product mixture occurs in a flow route \(r_1\) when there exists a joining open flow route \(r_2\) which begins at a product source element in \(E^\uparrow\) and ends at an element \(e_i\) in \(r_1\).
We now have a precise characterisation of product leaks and mixtures using the abstract plant model and the flow allowance model. These modelled situations represent a necessary condition for actual leaks and mixtures of products along a flow route in the plant, and may be used by a product flow path management system to both detect and prevent potentially hazardous situations in the plant, as will be shown in Chapter 7.

In order to formalise the absence of leaks and mixtures at an element \( e \) with respect to a flow route \( r \) which contains it, we present the following definition of flow protection functions at a flow allowance state \( \sigma \) of the plant. As these and some other subsequent functions presented here yield Boolean values, they are defined in terms of first-order logic formulae.

**Definition 14 (Flow protection)** Given a plant \( (T, E, C, \tau, e, o) \), a flow allowance model \( \Phi \) and a flow allowance state \( \sigma \), the input flow protection function \( \rho^I_\sigma : E \times R \to \{0, 1\} \) at \( \sigma \) is defined as

\[
\rho^I_\sigma(e, r) = \neg \left[ \exists e_1 \ldots e_n \in R_\sigma : \{e_1, \ldots, e_n\} \cap \alpha(r) = \emptyset, \ e_1 \in E^\uparrow \right].
\]

Analogously, the output flow protection function \( \rho^O_\sigma : E \times R \to \{0, 1\} \) at \( \sigma \) is defined as

\[
\rho^O_\sigma(e, r) = \neg \left[ \exists e_1 \ldots e_n \in R_\sigma : \{e_1, \ldots, e_n\} \cap \alpha(r) = \emptyset, \ e_n \in E^\downarrow \right].
\]

Because of the use of negations in this definition, the functions \( \rho^I_\sigma \) and \( \rho^O_\sigma \) yield the value 0 when the potentially hazardous situations of mixtures and leaks are present, and 1 when they are absent. Furthermore, for a diverging branch flow route to be considered a leak, and for a joining branch flow route to be considered a mixture, they must not have any element in common with the route \( r \) whose enclosure is being determined, excluding the element \( e \). This is done in order to exclude branch routes which leave and rejoin \( r \) from this condition, as well as routes which have a common prefix of suffix section with \( r \). This limits the detection of leaks and mixtures to strictly diverging and strictly joining branch routes.

The goal now is to determine the absence of leaks and mixtures along a flow route in order to formulate its enclosure, and this may be done by decomposing a flow route into a series of flow hops. A flow hop is a sequence \( e_1 e_2 \) of two elements which is a sub-sequence of a flow route \( r = \ldots e_1 e_2 \ldots \) in the plant, as shown in Figure 4.9. The flow hop represented by \( e_1 e_2 \) is said to be enclosed if flow may occur from \( e_1 \) to \( e_2 \) exclusively. Additional outgoing flow in the form of product leaks from \( e_1 \) and additional incoming flow in the form of mixtures to \( e_2 \) are violations of this principle. Thus, we may determine if a flow hop is enclosed by verifying the impossibility of these additional flows through the application of the functions \( \rho^I_\sigma \) and \( \rho^O_\sigma \). The enclosure of an entire flow route may now be determined by the application of this rule to each flow hop in the route, as shown in Figure 4.10. A given intermediate element \( e_1 \) in a flow route \( r \) is both start and end of a flow hop, and should therefore be free from both leaks and mixtures for \( r \) to be enclosed. On the other hand, the initial element of a flow route \( r \) need only be free from leaks, and the final element of a flow route \( r \) need only be free from mixtures for \( r \) to be enclosed. This follows from the determination of enclosure based on the point of view of a flow hop as shown in Figure 4.9, and additionally has the nice property of allowing a product to flow into a flow route through its initial element without considering this a product mixture, and of allowing a product to flow out of a flow route through its final element without considering this a product leak. This suggests that the concatenation of flow routes in a plant, and therefore of flow paths, may be done without violating the enclosure property of each individual route. It also means that long flow routes may be broken down into shorter
**Figure 4.9:** Determination of the enclosure of a flow hop at a flow allowance state $\sigma$. In a flow route $r = \ldots e_1 e_2 \ldots$, a flow hop from $e_1$ to $e_2$ is enclosed if there exists no product leak at $e_1$, and if there exists no product mixture at $e_2$. This is determined by the value of the function $\rho^O_{e_1}$ at $e_1$ and $\rho^I_{e_2}$ at $e_2$.

\[
\rho^O_{e_1}(e_1, r) = 1 \quad \rho^I_{e_2}(e_2, r) = 1
\]

**Figure 4.10:** Determination of the enclosure of a flow route at a flow allowance state $\sigma$. A flow route $r = e_1 e_2 \ldots e_n$ is enclosed if every flow hop it contains is also enclosed: every element except the final element $e_n$ must be free of leaks, and every element except the initial element $e_1$ must be free of mixtures.

\[
\rho^O_{e_1}(e_1, r) = 1 \quad \rho^I_{e_2}(e_2, r) = 1 \quad \rho^O_{e_n}(e_n, r) = 1
\]
sub-routes and handled individually while still adhering to this property. This benefits the
users of this model, as it supports the flexible use of flow routes and flow paths in a plant.

Following the idea shown in Figure 4.10, the set of enclosed flow routes of a plant at a flow
allowance state is defined in the following.

**Definition 15 (Enclosed flow routes)** Given a plant \((T, E, C, \tau, e, \circ)\), a corresponding flow
allowance model \(\Phi\) and a flow allowance state \(\sigma\), the set of enclosed flow routes \([R]_{\sigma} \subseteq R\)
of the plant at \(\sigma\) is defined as

\[
[R]_{\sigma} = \{ r \mid r \in R, f_{\sigma}(r, r) = 1 \}
\]

with the help of the auxiliary function \(f_{\sigma} : R \times R \to \{0, 1\}\) defined as

\[
\begin{align*}
  f_{\sigma}(e, r) &= 1 \\
  f_{\sigma}(e_1 e_2 \ldots e_n, r) &= [\rho_{\sigma}^O(e_1, r) \land \rho_{\sigma}^I(e_2, r) \land f_{\sigma}(e_2 \ldots e_n, r)].
\end{align*}
\]

An enclosed flow route at a flow allowance state \(\sigma\) is a flow route of the plant where every
flow hop it contains is also enclosed at \(\sigma\); in this definition, the function \(f_{\sigma}\) determines this
property for a given flow route \(r\) inductively over the length of \(r\). Its definition may literally be
read as follows: “a partial flow route which contains a single element \(e\) is always enclosed
with respect to its container flow route \(r\); also, a partial flow route which contains at least two
elements \(e_1\) and \(e_2\) at its beginning and in that order, is enclosed with respect to its container
flow route \(r\) whenever \(r\) is free of product leaks at \(e_1\), \(r\) is free of product mixtures at \(e_2\), and
the partial flow route which results by removing \(e_1\) from the given partial flow route is itself
enclosed with respect to \(r\).” The reader may assert that this reasoning coincides with the idea
of enclosed flow route shown in Figure 4.10, as it verifies that every flow hop \(e_1 e_2\) in the flow
route \(r\) is enclosed with respect to \(r\).

With the definition of the set \([R]_{\sigma}\) of enclosed flow routes, we complete a modelling frame-
work of product flow in processing plants which supports the treatment of product flow paths
as presented in this work.
5 Algorithms

The formal model of product flow presented in Chapter 4 supports the development of algorithms that fulfil the functional requirements of a product flow path management system as outlined in Chapter 2. These algorithms are the topic of the present chapter. It begins by presenting the decentralised component model that is followed in this work, and continues with the presentation of decentralised algorithms which are based on this model and perform the tasks of discovering, monitoring and allocating product flow routes in a plant. Afterwards, a simple algorithmic approach to the automatic synthesis of decentralised component-based systems is presented, which may be used to automatically construct a decentralised product flow path management system for a given plant based on an abstract model of the plant. The computational complexity of these algorithms is analysed at the end of this chapter.

5.1 Decentralised Component Model

As explained in Chapter 3, decentralised systems are designed and built in many different ways. The decentralisation model that is followed in this work, as stated in Chapter 2, can be regarded as plant-oriented: the decentralised system consists of a collection of active participants called components, and every component of the system corresponds to an element \( e \in E \) of the abstract plant model. This element is called the local element of the component. This one-to-one mapping between plant elements and system components allows a clear and localised formulation of the algorithms that the system performs, because every component has complete access to the information regarding its local element, and no direct access to the information regarding the other elements in the plant. In this manner, the notions of information hiding and encapsulation from the methodology of object-oriented software design [19] are followed.

In order for the algorithms to function in a decentralised manner, the components of the system must exchange information with one another, and in our system the components achieve this by sending and receiving messages, which are represented by data structures with named data fields, and which are formally denoted as elements from the set of messages \( M \). Thus, the technique of message-passing [35] is employed. The exchange of messages is restricted in a plant-oriented manner as well: every component has one communication port for every connector \( c \in C_e \) of its local element \( e \), and these ports are interconnected through bidirectional communication links in accordance with the connection relation \( \circ \) of the plant connectors. Thus, the communication structure of the decentralised system is an analogy of the plant layout, which means that components may only exchange messages with other components whose local plant elements are directly connected to their own local plant elements in the abstract plant. This causes the flow of messages to be restricted to the same paths as the flow of products in the plant, which further means that there is a correspondence between flow routes and message routes. Rather than being a limitation, this communication scheme actually simplifies the design of the algorithms used by a product
5.2 Decentralised Algorithms

Figure 5.1: Graphical representation of the corresponding abstract plant model from Figure 4.1 (left), and a depiction of the decentralised component-based system which corresponds to it (right). There exists one component for every plant element in the abstract plant, and the connection structure of the plant defines the communication structure of the system.

flow path management system, because within them every component of the system only executes the necessary operations which regard its own plant element, and delegates the rest of the required operations to the rest of the components in the route. Figure 5.1 shows the structure of a decentralised component-based system which corresponds to the sample filling station that was shown in Figure 4.1.

The decentralised component-based approach presented here is inspired by geographical railway control, as presented in Chapter 3. We hereby make an analogy between railway structure and plant structure: whereas geographical control systems consist of logical objects which are laid out along the structure of the train track network, in the present approach, components are laid out in analogy to the structure of the plant. In this way, we benefit from the many advantages that geographical railway control has over the more traditional centralised techniques, such as the simplified construction, extension and maintenance of the system. As we will see, this also offers the opportunity of achieving the automatic construction of a decentralised system.

5.2 Decentralised Algorithms

Product flow path management, as presented in this work, offers a core functionality which consists of several operations on product flow paths as presented in Chapter 2. The purpose of this section is to explain the algorithms which implement these operations in a product flow path management system. As stated in Chapter 2, the design of the product flow path management system and its algorithms follows a decentralised approach, because it offers several advantages over a fully centralised solution, such as increased flexibility, robustness and maintainability.
5.2.1 Syntax of Algorithms

The algorithms that are presented in this work are written using pseudo-code, which is an abstraction of the syntax of common structured programming languages such as Pascal or C, combined with mathematical expressions. This formulation is not intended to be directly executed on any computer system, but rather to present the algorithms in a way which is clearly understandable by human readers. Implementing these algorithms on a real programming language based on this formulation should mostly be a straightforward task.

The algorithms are broken down into a collection of procedures, which helps the reader concentrate on each the different tasks which are carried out by each algorithm. Every procedure is written as

\[
\text{procedure } \text{ProcedureName}(a_1, \ldots, a_n) \text{ returns } S \\
\text{ProcedureBody} \\
\text{end procedure}
\]

where \(a_1, \ldots, a_n\) are the \(n\) formal arguments that the procedure expects. The body of the procedure corresponds to common pseudo-code that may contain variable assignments, conditional and iterative control structures, procedure calls, etc. The construct \text{returns } S is optional, and its usage means that the procedure will return a value from the set \(S\) to its caller as it exits.

Variables are assumed to be implicitly declared, as well as the data-types of variables and formal arguments. In addition to scalar data-types, arrays are also used in order to store sequences of data items. The expression \(\langle x_1, \ldots, x_n \rangle\) refers to an array of length \(n\) which holds the items \(x_1, \ldots, x_n\). The syntax \(a[i]\) refers to the item located at index \(i\) in the array referred to by \(a\). Hereby, the convention of array-indexing from the language C is followed, where \(a[0]\) corresponds to the first item in the array \(a\). Furthermore, we will conceive data structures with named data fields. The syntax \(v.f\) refers to the data field \(f\) of the data structure referred to by \(v\). The fields contained in data structures are also assumed to be implicitly declared.

The following predefined “library procedures” are assumed, which consist of basic operations that are commonly offered by modern programming environments, or that may be easily implemented in them:

- **LengthOf(a)** Returns the length of the array \(a\), that is, the number of elements that \(a\) can hold, as a positive integer number.

- **Contains(a,x)** Returns 1 (true) if the array \(a\) contains the item \(x\), and 0 (false) otherwise.

- **IndexOf(a,x)** Returns the index \(i\) of the first occurrence of the object \(x\) in the array \(a\), such that \(a[i] = x\), or the value \(-1\) if \(x\) is not contained in \(a\).

- **Append(a,x)** Returns a new array which holds all the items in the array \(a\), and additionally the item \(x\) as its last element. The ordering of the items with respect to the array \(a\) is preserved.
5.2.2 Component Environment

Within a component, one or more algorithms may operate at a given time. These algorithms have access to the information about the local plant element $e$ and its connectors $c_i \in C_e$, such as its flow allowance model $\Phi(e)$ and its current flow allowance setting $\sigma(e)$. Additionally, the component offers the algorithms the following basic services in the form of procedures:

- **CreateMessage($t$)** Creates a new message data item with a message type $t$. The type of the message may be accessed later via the data field `MessageType` of the message.

- **SendMessage($c, m$)** Sends out a message $m$ over the local connector $c$. This means that the message will be sent to the component whose local element is connected to the local element of the current component through a connection of $c$.

- **GetConnectedElement($c$)** Obtains the neighbouring plant element that is connected to the local element by a connection of the connector $c$. It is assumed that this information is available to the component, and may be achieved in an implementation, for instance, by facilitating a "connected component identifier" on every communication link. As these links are point-to-point in the presented model, this should not be difficult to accomplish in practise.

It is assumed that the communication between components is **asynchronous** and **non-blocking**. This means that when a message is sent, the algorithm which sends the message continues executing immediately after issuing a call to `sendMessage`, and that the message is placed on a message queue in order to be sent. The communication channels are considered to be **reliable**, which means that no message gets lost before reaching its destination, that no message is delivered twice (i.e. free of duplications), and that every message that is delivered was originally sent (i.e. free of insertions), and **fifo** (first in, first out), which means that messages reach their destination in the same order as they were sent. Furthermore, we make no assumptions about the scheduling of the algorithms among the different components, other than to assume a **fair scheduling** scheme where no algorithm suffers from **starvation**. This means that every algorithm at every component is able to execute again and again during the operation of the system, and is never blocked indefinitely. A more precise description of all these properties may be consulted in [35].

The algorithms that are presented in this work are **reactive** in nature, meaning that they first wait for input, and then execute when an input is obtained. In our component-based model, whenever a component receives a message $m$, it issues a call to a procedure `HandleMessage($e, c, m$)` of every algorithm, where $e$ is the local plant element of the component and $c$ is the connector that corresponds to the communication link where the message was received. In this way, the definition of the procedure `HandleMessage` determines how the algorithm reacts to incoming messages from neighbouring components. It is assumed that each algorithm is defined in a proper **namespace** which avoids collisions of procedure names among different algorithms executing in the same component.

In addition to the reactive behaviour, algorithms may also execute actions of their own which are not triggered by incoming messages. These may occur periodically at given time intervals, or may be caused by local component events such as user requests. This additional behaviour is algorithm-specific and will be properly explained in each case.
5.2.3 Flow Path Analysis

The first algorithm that is presented in this work is called flow path analysis, and deals with the task of discovering flow routes in a plant as explained in Chapter 2. A plant \((T, E, C, \tau, \epsilon, \circ)\) and a corresponding flow allowance model \(\Phi\) determine the set of possible flow routes of the plant, and this is given by the definition of the set \(R\). Thus, flow path analysis has the objective of finding flow routes \(r_i \in R\) in the plant.

In accordance with the flow path model and the general description of a flow path management system presented in Chapter 2, the main goal of this algorithm is not to find all flow routes in \(R\), but rather to offer a mechanism for discovering interesting and useful flow routes, according to criteria that are plant-specific and user-defined. For this, the user gives the flow path analysis algorithm a simple specification of a set of flow routes, which consists of a pair of plant elements \((e, e')\). In this specification, \(e\) is the initial element of the flow route, also called the origin of the route, and \(e'\) is the final element of the flow route, or the target of the route. By giving the flow path analysis algorithm a flow route specification of this form, the user expresses his or her intention of finding a flow route that may be used as a flow path which transports material from \(e\) to \(e'\) in the plant.

Within a component, the procedure \texttt{ReportFlowRoute}(r) may be called by the flow path analysis algorithm in order to submit a flow route \(r\) that has been found. The component may then process this newly flow route in a system-dependent way.

### Decentralised Search

In essence, product flow path analysis poses a search problem, which may be solved by several well-known search algorithms such as depth-first or breadth-first search, as outlined in Chapter 3. For finding flow routes in a plant, one can use the flow allowance graph of the plant itself as a search graph: for a flow route specification \((e, e')\), a search begins at \(e\) and seeks to find \(e'\); the search graph is traversed and whenever \(e'\) is reached, the path in the graph that has been traversed by the search is added to the set of solutions. Any correct search strategy yields the same result, differing only in the order in which the search graph is traversed, and the order in which the solutions are given. The basis of the product flow path analysis algorithm is a form of decentralised breadth-first search \([35]\). This search technique has been chosen because it is more natural to implement within our model of decentralisation: breadth-first search looks “in all directions” at the same time, and this may be done by components which issue search requests in all possible directions when a route is searched. Additionally, this technique finds shorter flow routes before longer ones, and this may be useful if shorter flow routes are preferred.

### Handling Incoming Messages

The procedure \texttt{HandleMessage} of the flow path analysis algorithm, which is called whenever a message that corresponds to this algorithm is received by a component, is defined as follows:

```plaintext
procedure HandleMessage(e, c, m)
    if m.MessageType = Search then
        HandleSearchMessage(e, c, m)
    else if m.MessageType = SearchResult then
```
5.2 Decentralised Algorithms

HandleSearchResultMessage(e, c, m)
end if
end procedure

Messages with the type Search are search requests, and those with type SearchResult correspond to one result of a given search. This procedure basically dispatches the message to the procedures HandleSearchMessage or HandleSearchResultMessage, depending on the type of the message received.

Searching for Flow Routes

When a flow route specification \((e, e')\) is given to a product flow path management system, an event occurs at the component whose local element is \(e\), which initiates the execution of the flow path analysis algorithm by invoking the procedure InitiateSearch with the arguments \((e, e')\). This procedure is defined as follows:

\[
\text{procedure InitiateSearch}(e, e')
\]
\[
\text{if } e = e' \text{ then}
\]
\[
\text{ReportFlowRoute}(e)
\]
\[
\text{else}
\]
\[
\text{for } c \in C_e \text{ do}
\]
\[
\text{if } \exists \phi_e \in \Phi(e) : \phi_e^I(c) = 1 \text{ then}
\]
\[
m \leftarrow \text{CreateMessage(Search)}
m.\text{EndLocation} \leftarrow e'
m.\text{FlowRoute} \leftarrow \langle e, e' \rangle
\]
\[
\text{SendMessage}(c, m)
\]
\[
\text{end if}
\]
\[
\text{end for}
\]
\[
\text{end if}
\]
end procedure

The first thing this procedure does is check whether \(e\) and \(e'\) are the same, which indicates that a trivial flow route of length 1 is the result of the search. Otherwise, a search message is sent out along every connector \(c\) of \(e\) where \(ec\) is an initial flow step in \(S^I\). The end location \(e'\) of the search and the partial flow route \(e\) found up to now are stored in the message.

The procedure HandleSearchMessage is called whenever a search message is received by a component, and it is defined as:

\[
\text{procedure HandleSearchMessage}(e, c, m)
\]
\[
\text{if } \exists \phi_e \in \Phi(e) : \phi_e^I(c) = 1 \text{ then}
\]
\[
\text{if } m.\text{EndLocation} = e \text{ then}
\]
\[
\text{SendBackResultMessage}(e, c, m)
\]
\[
\text{else if not Contains}(m.\text{FlowRoute}, e) \text{ then}
\]
\[
\text{ForwardSearchMessage}(e, c, m)
\]
\[
\text{end if}
\]
\[
\text{end if}
\]
end procedure
This procedure first checks whether $ce$ is a valid final step in $S^F$, that is, if it is possible for flow to enter $e$ through $c$ at some flow allowance setting $\phi_e$ of $e$. If this is not the case, then nothing else is done, effectively disregarding the message $m$, which means that the search process is aborted at this component. Otherwise, it is checked if the end location stored at $m$ is the same as the local element $e$, which means that a specified flow route has been found. In this case, the procedure $SendBackResultMessage$ is invoked. If a flow route has not yet been found, then the search is continued by invoking the procedure $ForwardSearchMessage$. However, this is done only if the local element is not contained in the partial flow route stored in $m$. This condition is needed in order to avoid cycles: a cyclic search path visits an element $e$ more than once, and this is detected by an element when it receives a search message whose partial flow route already contains it. Therefore, by discarding search messages when this occurs, the flow path analysis algorithm avoids cycles when it searches for flow routes in the plant.

The procedure $ForwardSearchMessage$ continues with the search process and is defined as follows:

```
procedure ForwardSearchMessage(c, c', m)
for $c' \in C_e$ do
    if $c' \neq c$ and $\exists \phi_e \in \Phi(e) : (\phi^I_e(c) = 1 \text{ and } \phi^O_e(c') = 1)$ then
        $m' \leftarrow CreateMessage(Search)$
        $m'.EndLocation \leftarrow m.EndLocation$
        $m'.FlowRoute \leftarrow Append(m.FlowRoute, e)$
        $SendMessage(c', m')$
    end if
end for
end procedure
```

This procedure identifies every connector $c'$ of the local element $e$ which is not the connector $c$ where the search message was received, and such that $ce c'$ conforms a valid middle flow step in $S^M$. In other words, it identifies every connector $c'$ through which product may flow outwards from the local element $e$ after having flown inwards through connector $c$. For each of these connectors found, a search message is created and sent along it. The end location of the new message is the same end location of the original message, as we are still looking for the same target element. Furthermore, the flow route stored in the new message is the partial flow route found up to now, and with the local element appended to it. This means that, for every middle flow step visited during the decentralised search, the local element of the corresponding component is appended to the partial flow route, which in turn, grows with every component that it visits.

### Reporting Search Results

The task of initiating the report of a resulting flow route is done by the procedure $SendBackResultMessage$ defined as:

```
procedure SendBackResultMessage(c, c, m)
    $m' \leftarrow CreateMessage(SearchResult)$
    $m'.FlowRoute \leftarrow Append(m.FlowRoute, e)$
    $SendMessage(c, m')$
```

5 Algorithms
This procedure basically creates a new result message where the complete flow route is stored, and sends it back over the same connector $c$ where the original search message was received. This is done in order to communicate this result to all components which were involved in the search and ultimately to the component which initiated the search, that is, the component that corresponds to the initial element of the flow route. The flow route stored in the message corresponds to the partial flow route found up to now, with the local element appended to it, as this element is the end location or target element of the route.

The procedure `HandleSearchResultMessage` processes result messages received by a component as follows:

```plaintext
procedure HandleSearchResultMessage(e, c, m)
    i ← IndexOf(m.FlowRoute, e)
    if i = 0 then
        ReportFlowRoute(m.FlowRoute)
    else
        ReturnSearchResultMessage(e, c, m, i)
    end if
end procedure
```

Within this procedure, if the local element is the first element of the flow route which is being reported, then the result has reached the origin of the flow route and the component which initiated the search process. In this case, the procedure `ReportFlowRoute` is called with the flow route stored in the received message, which as was mentioned earlier, processes the flow route in a system-dependent way, thus concluding the task of the flow path analysis algorithm. In the case that the local element is not the origin of the flow route, the procedure `ReturnSearchResultMessage` is invoked, which is itself defined as follows:

```plaintext
procedure ReturnSearchResultMessage(e, c, m, i)
    for $c' \in C_e$ do
        if $c' \neq c$ and $m.FlowRoute[i - 1] = GetConnectedElement(c')$ then
            $m' ← CreateMessage(SearchResult)$
            $m'.FlowRoute ← m.FlowRoute$
            SendMessage($c', m'$)
        end if
    end for
end procedure
```

This procedure first identifies the connector $c'$ of the local element which connects this element with its predecessor element in the flow route found: given that $m.FlowRoute[i]$ corresponds to the local element, its predecessor is stored in $m.FlowRoute[i - 1]$. Then, a new result message, which is a copy of the received message, is created and sent along this connector. In this manner, identical result messages are sent back towards the component which initiated the flow route search. When this component finally receives the result message, it reports it locally as defined in the procedure `HandleSearchResultMessage`.
Similar Algorithms

There are many algorithms, used in other application domains, which are similar to the decentralised product flow path analysis algorithm that has been presented in this section. Some of them have been used as a source of inspiration when formulating the decentralised technique for searching flow routes in plants. Because of this, they are mentioned in the following.

- **Geographical railway interlocking** [43]. As presented in Chapter 3, geographical control systems consist of logical objects which are laid out along the structure of the train track network. When a train route is to be set up for use by a train, an electrical system sends a signal from the start of the route that spans in all possible directions, forming a tree structure. When the end of the route is reached by this signal, a response signal is sent back towards the start of the route, which causes all intermediate switches to move into the required position for the route to be used. These electrical signals advance along the structure of the railway network in the same manner as how the search and result messages corresponding to product flow path analysis move through the decentralised component-based system. This is because geographical railway interlocking was the starting point for the formulation of the flow path analysis algorithm that has been presented in this section.

- **Distance-vector routing in computer networks** [57]. The goal of distance-vector routing may be stated as follows: given a network of interconnected communication nodes, provide each node with routing information to each of the other nodes in the form of a table which contains the target nodes, the distance to each target node and the immediate neighbour which leads to each target node. The algorithm operates in a distributed fashion, where each node periodically sends its own information to each of its neighbours. When a node receives new routing information, its updates its own tables accordingly. In this way, eventually all nodes hold routing information about every other reachable node, without communicating directly with any node other than their direct neighbouring nodes. This work adopts the idea of sharing routing information used in distance-vector routing algorithms as a model of route information sharing among components. This may be seen in the use of the data field EndLocation, which corresponds to the target of a route. This approach has been extended in this work by transmitting the entire flow route in a message.

- **Monotone data-flow analysis in program analysis and optimisation** [2, 30]. Monotone data-flow analysis is a common technique used for the static analysis and optimisation of programs performed by optimising compilers. It is based on the mathematical theory of **lattices**, **monotone functions** and **fixed points**. The analysis works on a directed **control-flow graph** which represents the execution of the program: each node corresponds to a program location, and each edge links a location with another location which may be executed next. The goal is to compute dynamic program properties in a static way, and is achieved by associating each location with a value which represents the information relevant to that location. A set of **data-flow equations** define each location value as a function of the values of its neighbouring locations. Because of the possibility of cyclic inter-dependencies among values, these equations may not always be solved in one pass. However, if the values form a finite lattice and the functions which relate values to each other are monotone, then the equation system may be solved by
applying a series of fixed point iterations, which transfer values between neighbouring locations until these values stabilise and no more changes may occur. The algorithmic approach used by decentralised product flow path analysis is similar to monotone data-flow analysis, given the similarity between control-flow graphs of programs, where program control flows, and flow allowance graphs of plants, where product flows. If flow route searches are initiated periodically, then the information which is contained in the messages which circulate through the decentralised system increases as the search progresses; however, when all flow routes have been found, then effectively a fixed point is reached, where the same messages are sent over and over again, and no additional results are obtained. This suggests that the traffic due to messages in the system is always bounded.

• Automotive navigation systems [32]. Automotive navigation systems are small computers that are used in automobiles and similar vehicles for guiding their drivers when travelling. These systems contain a road database that models the actual road network where the vehicle may travel, and a positioning device which enables the system to determine its current position in the road network. When given a destination as input, the system may compute a route to be followed and then assist the driver by giving step by step instructions in order to follow the route and reach the destination. The task of computing a route that connects two locations in the road database is very similar to the task of finding a flow route in an abstract plant model which satisfies a flow route specification \((e_1, e_2)\).

Flow Path Analysis of Open Flow Routes

It is sometimes useful to be able to restrict the search that flow path analysis performs to open flow routes only. For instance, if a plant is large and it is required to determine which flow routes are currently open for monitoring or diagnostic purposes, a combination of flow path analysis and flow path monitoring may be employed. However, the overhead with respect to the number of messages sent along the decentralised system may be reduced if the flow route search is restricted to open flow routes, which effectively performs a kind of monitoring of the currently open flow routes which satisfy the given flow route specification.

The flow path analysis algorithm that was presented in the previous section must be slightly modified in order to restrict the search to currently open flow routes. The modifications only affect those procedures which are involved in the searching phase. The modified versions of these procedures are presented in the following.

```java
procedure InitiateSearch(e, e')
  if e = e' then
    ReportFlowRoute(e)
  else
    \( \phi_e \leftarrow \sigma(e) \)
    for \( c \in C_e \) do
      if \( \phi^O_e(c) \) then
        m ← CreateMessage(Search)
        m.EndLocation ← e'
        m.FlowRoute ← \{ e' \}
        SendMessage(c, m)
```
5 Algorithms

end if
end for
end if
end procedure

procedure HandleSearchMessage(e, c, m)
\( \phi_e \leftarrow \sigma(e) \)
if \( \phi^I(e) \) then
if \( m.\text{EndLocation} = e \) then
SendBackResultMessage(e, c, m)
else if not Contains(m.\text{FlowRoute}, e) then
ForwardSearchMessage(e, c, m)
end if
end if
end if
end procedure

procedure ForwardSearchMessage(e, c, m)
\( \phi_e \leftarrow \sigma(e) \)
for \( c' \in C_e \) do
if \( c' \neq c \) and \( \phi^I(e) \) and \( \phi^O(c') \) then
\( m' \leftarrow \text{CreateMessage}(\text{Search}) \)
\( m'.\text{EndLocation} \leftarrow m.\text{EndLocation} \)
\( m'.\text{FlowRoute} \leftarrow \text{Append}(m.\text{FlowRoute}, e) \)
SendMessage(c', m')
end if
end for
end procedure

In all of these procedures, \( \phi_e \) obtains the value of the current flow allowance setting of the local element \( e \). Rather than checking if there exists a flow allowance setting in the flow allowance model of the local element which permits the flow of products in the direction of the flow route that is being searched, these procedures test if this flow is permitted at the current flow allowance setting of the element. In this way, open flow steps from \( S_\sigma \) are identified and required for the continuation of the search, instead of accepting flow steps from \( S \), which guarantees that any flow route found by this version of the algorithm is open at the current flow allowance state \( \sigma \) of the plant.

5.2.4 Flow Path Monitoring

The second decentralised algorithm that is presented in this work is called flow path monitoring, and it has a dual goal: first, to determine if a given flow route \( r \) is open at the current flow allowance state \( \sigma \) of the plant, and second, to determine if the given flow route \( r \) is enclosed at the current flow allowance state \( \sigma \). Even though these two conditions are independent from one another as explained in Chapter 4, the decentralised flow path monitoring algorithm evaluates both of them at the same time for a given flow route because it is natural to do so: both are flow route properties which depend on the flow allowance state of the plant, and which may be evaluated by an algorithm that traverses the flow route and all its relevant
branch flow routes in a similar way. Most importantly, having an integrated algorithm which performs an online evaluation of the open and enclosed properties of a flow route simplifies the integration of the functionality of product flow path monitoring within a product flow path management system.

The mechanism which is used by the decentralised flow path monitoring algorithm is similar to the one used by the decentralised flow path analysis algorithm presented in the previous section. Components first send messages along a flow route in a forward direction, and then send response messages backwards along the flow route. This time, branch flow routes are also involved, and this messaging technique equally applies to them. As the flow route which is to be monitored is previously known, messages are sent along this route without performing any search. However, in the case of branch flow routes, these routes are not known in advance and must therefore be searched by this algorithm in a similar way to how flow routes are searched by the flow path analysis algorithm.

The flow path monitoring algorithm reports whether the given flow route is currently open or not by calling the predefined procedure `ReportMonitorResult(r, o)` at the component whose local element is the initial element of the route. Hereby, `r` is the flow route that is being monitored and `o` is a Boolean value which indicates if `r` is open. Furthermore, the algorithm determines the enclosure of the flow route by detecting product leaks and product mixtures, as defined earlier in this chapter. Every time one of these situations is detected, the algorithm calls the predefined procedure `ReportAlert(t, r, b)` at the component whose local element is the initial element of the route. Hereby, `t` is the type of alert that is being issued (Leak or Mixture), `r` is the flow route that is being monitored and `b` is the branch flow route that is causing the situation. In this manner, the flow path management system may process these procedure calls in order to complete the flow path monitoring functionality, for instance by updating the state of a corresponding flow path object, issuing user alarms, updating a process visualisation, etc.

**Handling Incoming Messages**

Whenever a message is received by a flow path monitoring algorithm which is active at a component of the decentralised system, it is distributed to one of several local procedures depending on the type of the message, as defined by the procedure `HandleMessage`:

```plaintext
procedure HandleMessage(e, c, m)
    if m.MessageType = Monitor then
        HandleMonitorMessage(e, c, m)
    else if m.MessageType = MonitorResult then
        HandleMonitorResultMessage(e, c, m)
    else if m.MessageType = MonitorBranch then
        HandleMonitorBranchMessage(e, c, m)
    else if m.MessageType ∈ {Leak, Mixture} then
        HandleAlertMessage(e, c, m)
    end if
end procedure
```

The flow path monitoring algorithm uses many different types of messages: `Monitor` is used by monitor request messages, `MonitorResult` is used for reporting the result of the monitoring operation, `MonitorBranch` is used to request the monitoring of branch flow
routes, and Leak and Mixture are used respectively to report possible product leaks and product mixtures that have been detected by the algorithm.

Monitoring Flow Routes

The component which corresponds to the initial element $e$ of a flow route $r$ may commence the process of monitoring $r$ by issuing a call to the procedure `InitiateMonitoring` of the flow path monitoring algorithm, which is defined as follows:

```
procedure InitiateMonitoring(e, r)
    if LengthOf(r) > 1 then
        for $c \in C_e$ do
            if $r[1] = \text{GetConnectedElement}(c)$ then
                SendInitialMonitorMessage($e, c, r$)
                SendInitialBranchMonitorMessages($e, 0, c, r$)
            end if
        end for
    else
        ReportMonitorResult(r, 1)
    end if
end procedure
```

This procedure initiates the monitoring of $r$ whenever $r$ contains more than one element; otherwise, the trivial route is considered to be open, and this result is reported for completeness. For non-trivial routes, the connector $c$ of the local element which has a connection to the second element of $r$ is identified, and the procedures `SendInitialMonitorMessage` and `SendInitialBranchMonitorMessages` are called. The latter deals with the sending of messages which monitor branch flow routes and will be presented later in this section, and the former, which sends out a monitor request along the route $r$, is defined as follows:

```
procedure SendInitialMonitorMessage($e, c, r$)
    $\phi_e \leftarrow \sigma(e)$
    $m \leftarrow \text{CreateMessage(Monitor)}$
    $m.\text{Open} \leftarrow \phi^O_c(c)$
    $m.\text{FlowRoute} \leftarrow r$
    SendMessage($c, m$)
end procedure
```

The message created by this procedure carries the complete flow route $r$ and an initial value for its open property, which corresponds to the output flow allowance setting of the connector $c$. This means that the initial value of this property is 1 whenever $ec$ is an open initial flow step. Effectively, the tests that are done throughout this algorithm are designed to determine if the flow steps from the flow route $r$ and its branch flow routes correspond to open flow steps or not. The message $m$ is sent along the connector $c$, that is, in a forward direction along the flow route $r$.

We will now examine the handling of monitoring request messages which have the message type `Monitor`. When one of these messages is received by a component, it is processed by the `HandleMonitorMessage` procedure which is defined as follows:
5.2 Decentralised Algorithms

procedure HandleMonitorMessage(e, c, m)
    i ← IndexOf(m.FlowRoute, e)
    if i = LengthOf(m.FlowRoute) - 1 then
        SendBackResultMessage(e, c, m)
        SendInitialBranchMonitorMessages(e, c, 0, m.FlowRoute)
    else
        HandleMonitorMessageAtMiddlePosition(e, c, m, i)
    end if
end procedure

This procedure first determines whether the local element e is the final element in the flow route that is being monitored. If this is the case, then the procedures SendBackResultMessage and SendInitialBranchMonitorMessages are called. The former initiates the sending of a result message that reports the open property of the corresponding flow route, and the latter initiates the monitoring of the branch flow routes which stem from the local element e. Otherwise, the procedure HandleMonitorMessageAtMiddlePosition, which is defined in the following, is called.

procedure HandleMonitorMessageAtMiddlePosition(e, c, m, i)
    for c' ∈ C_e do
        if c' ≠ c and m.FlowRoute[i + 1] = GetConnectedElement(c') then
            φ_e ← σ(e)
            m' ← CreateMessage(Monitor)
            m'.Open ← [m.Open and φ_e(c) and φ_e(c')]
            m'.FlowRoute ← m.FlowRoute
            SendMessage(c', m')
            SendInitialBranchMonitorMessages(e, c, c', m.FlowRoute)
        end if
    end for
end procedure

What this procedure does is first look for a flow step cec' where the connector c' has a connection which leads to the element e' that comes immediately after e in the flow route r that is being monitored. If this flow step is found, then a new monitor request message is created and sent along c'. The flow route stored in the new message is preserved from the original message, and the open property of the flow route corresponds to the conjunction of two conditions: the corresponding value from the original message, that is, the open property of the partial flow route monitored up to this point, and a condition which says if cec' is an open middle flow step. As stated earlier in this chapter, a flow route r is open only if every one of the flow steps it contains is open as well, and this idea is expressed by means of this conjunction. Finally, the procedure SendInitialBranchMonitorMessages is called, which is responsible for initiating the monitoring of the branch flow routes which stem from the local element e.

Reporting Monitoring Results

When a monitoring request message reaches the component whose local element is the final element of the flow route r that is being monitored, a chain of messages is initiated
with the goal of reporting the result of the monitoring. This is done by calling the procedure `SendBackResultMessage` defined as follows:

```
procedure SendBackResultMessage(e, c, m)
    φ_e ← σ(e)
    m′ ← CreateMessage(MonitorResult)
    m′.Open ← [m.Open and φ_I(e)]
    m′.FlowRoute ← m.FlowRoute
    SendMessage(c, m′)
end procedure
```

This procedure creates a new result message which preserves the flow route from the original monitoring request message, and which holds the definitive value of the open property for the corresponding flow route. This value corresponds to the conjunction of the value of the open property which was carried by the monitor request message that was received, and the input flow allowance setting of the connector \(c\). Just like the procedure `HandleMonitorMessageAtMiddlePosition` calculates the value of this property by testing if the corresponding middle flow step \(ccc'\) is open at the current flow allowance state \(σ\) of the plant, the procedure `SendBackResultMessage` tests if the final flow step \(c\) is open at \(σ\), and uses this for deciding whether the flow route is open at \(σ\) by evaluating this conjunction. The new message is sent backwards along the corresponding flow route, that is, over connector \(c\).

The procedures `HandleMonitorResultMessage` and `ReturnMonitorResultMessage` of the flow path monitoring algorithm are very similar to their counterparts from the flow path analysis algorithm. Their task is to forward a message with the result of the algorithm, backwards along the route until it reaches the component whose local element is the initial element of the flow route. When this occurs, the procedure `ReportMonitorResult` is called at this component in order to report this result to the flow path management system. Its arguments – the flow route that was monitored and the value of its open property – are taken directly from the result message that was received. The definition of the procedure `HandleMonitorResultMessage` follows.

```
procedure HandleMonitorResultMessage(e, c, m)
    i ← IndexOf(m.FlowRoute, e)
    if i = 0 then
        ReportMonitorResult(m.FlowRoute, m.Open)
    else
        ReturnMonitorResultMessage(e, c, m, i)
    end if
end procedure
```

The procedure `ReturnMonitorResultMessage` is called whenever the result message is processed by a component whose local element is not the initial element of the flow route, and basically sends a copy of this message to the element \(e'\) that precedes the component's local element in the flow route. This procedure is defined in the following.

```
procedure ReturnMonitorResultMessage(e, c, m, i)
```

80
Monitoring Branch Flow Routes

The part of the flow path monitoring algorithm that has been presented so far accomplishes the monitoring of the given flow route, determining if this flow route is open at the current flow allowance state $\sigma$ of the plant. A second part of this algorithm deals with the monitoring of the branch flow routes that stem from the main flow route that is being monitored, with the purpose of determining the enclosed property of this flow route by searching for product leaks and product mixtures along these branch flow routes.

In order to determine the presence of leaks and mixtures along branch flow routes in a fully decentralised manner, the possibility of flow at each element along a branch flow route is analysed as shown in Figure 5.2. Two situations of an element $e$ are considered: when the element is the initial element in the branch flow route $b$ and is therefore an element in the flow route $r$ that is being monitored, and when the element is in the branch flow route $b$ but not in the flow route $r$. In both cases, the Boolean values $\lambda_e$ and $\mu_e$ respectively represent the possibility of leak flow and mixture flow which occurs inside the element, from and to $r$. In the first case, the branch flow route $b$ starts at the element $e$ and continues along one of its connectors $c'' \in C_e$ which is not part of the flow trace of $r$. For this connector, the Boolean values $\lambda_{c''}$ and $\mu_{c''}$ respectively represent the possibility of leak and mixture flows through the connector. Also, these values describe the leak and mixture flows “after” the connector $c''$, that is, on the immediate exterior of $c''$. In the second case, the branch flow route $b$ contains $e$ in a non-initial position, and the values $\lambda$ and $\mu$ respectively represent the possibility of leak and mixture flow “before” the connector $c$ which leads to the flow route $r$. In turn, the values $\lambda_{c'}$ and $\mu_{c'}$ respectively represent the possibility of leak and mixture flow through and “after” the connector $c'$ which leads away from the flow route $r$, and possibly to a product source or sink. By computing the values of these conditions for every element in a branch flow route, the flow path monitoring algorithm is able to detect both leaks and mixtures at the same time, and in a fully decentralised manner.

The monitoring of the collection of branch flow routes which stem from the flow route $r$ at the element $e$ is initiated by the procedure $\text{SendInitialBranchMonitorMessages}$ which is defined as follows:

```
procedure SendInitialBranchMonitorMessages(e, c, c', r)
for $c'' \in C_e$ do
  if $c'' \neq c$ and $c'' \neq c'$ then
    $\phi_e \leftarrow \sigma(e)$
    $\lambda_{c''} \leftarrow [c' \in C_e \text{ and } \phi^O_e(c'')]$
    $\mu_{c''} \leftarrow [c \in C_e \text{ and } \phi^I_e(c'')]$
    if $\lambda_{c''}$ or $\mu_{c''}$ then
```
Figure 5.2: Possible leak flow and mixture flow of products within an element that occurs in a branch flow route. On the left, an element $e$ that is the initial element in a branch flow route $b$ which stems from a flow route $r$. On the right, a non-initial element $e$ within a branch flow route $b$.

\[
\begin{align*}
\lambda_{e''} & \quad \mu_{e''} \\
\lambda_{e'} & \quad \mu_{e'} \\
r & = \ldots e \ldots \\
b & = e \ldots
\end{align*}
\]

This procedure takes as arguments, apart from the local element $e$, the flow route $r$ that is being monitored, two connectors $c$ and $c'$. These arguments correspond to the setting that was depicted in Figure 4.8: the flow route $r$ is the sequence of elements $e_1 \ldots e_i \ldots e_n$, the local element $e$ corresponds to the element $e_i$ of the illustration, and the connectors $c$ and $c'$ are the connectors of the element $e_i$ pictured at the left and right of the element respectively, meaning that $ce_i c'$ is a flow step of $r$. Thus, any branch flow route which begins at $e$ (respectively $e_i$) must involve a third connector $c''$ of $e$ which is none of the connectors in the flow step. Because of this, the algorithm executes its body for every such connector $c''$ that it is able to find from the set of connectors of $e$. Figure 4.9 showed that a flow hop is enclosed if its first element is free of leaks and its second element is free of mixtures, and this idea is followed by this procedure in a flexible manner: if the connector $c'$ is a valid connector of $e$, then it means that $e$ is the first element in a flow hop, where the second element in the flow hop is reachable via $c'$; also, if the connector $c$ is a valid connector of $e$, then it means that $e$ is the second element in a flow hop, where the first element in the flow hop is reachable via $c$. The procedures which call SendInitialBranchMonitorMessages control indicate which of these situations is in effect by giving either valid connectors or a value 0 in place of these arguments. The conditions $\lambda_{e''}$ and $\mu_{e''}$ are computed as follows: $\lambda_{e''}$ indicates if
5.2 Decentralised Algorithms

A leak is to be searched along $c''$ and is true whenever $e$ is the first element in a flow hop and the connector $c''$ currently allows outbound flow; likewise, $\mu_{c''}$ indicates if a mixture is to be searched along $c''$ and is true whenever $e$ is the second element in a flow hop and the connector $c''$ currently allows inbound flow. Having evaluated these conditions, the procedure creates a branch monitoring message which carries their values in order to express the kind of situations that should be detected. The flow route $r$ is also carried by this message, as well as the branch flow route that has been found up to this point, which in this case consists of the initial element $e$ of the branch flow route. Finally, the message is sent along the branch flow route, that is, via the connector $c''$.

When a branch monitoring message is received by a component, the procedure `HandleMonitorBranchMessage` is called to handle it, and this procedure is defined as follows:

```plaintext
procedure HandleMonitorBranchMessage(e, c, m)
  $\phi_e$ ← $\sigma(e)$
  $\lambda_e$ ← $[m.MonitorLeaks and \phi^l(c)]$
  $\mu_e$ ← $[m.MonitorMixtures and \phi^o(c)]$
  if $\lambda_e$ or $\mu_e$ then
    if not Contains(m.FlowRoute, $e$) and not Contains(m.BranchFlowRoute, $e$) then
      $\alpha$ ← 0
      if $\lambda_e$ and $e \in E^\downarrow$ then
        SendBackAlertMessage($e, c, m, \text{Leak}$)
        $\alpha$ ← 1
      end if
      if $\mu_e$ and $e \in E^\uparrow$ then
        SendBackAlertMessage($e, c, m, \text{Mixture}$)
        $\alpha$ ← 1
      end if
      if not $\alpha$ then
        ForwardMonitorBranchMessage($e, c, m, \lambda_e, \mu_e$)
      end if
    end if
  end if
end procedure
```

This procedure receives a message with the Boolean data items `MonitorLeaks` and `MonitorMixtures` which respectively hold values that indicate if leaks and mixtures are to be detected. The new conditions $\lambda_e$ and $\mu_e$ are computed from them by considering the current flow allowance setting of the connector $c$: if this setting inhibits flow in the direction of the leak or the mixture, then the potential problem is already avoided and its detection need not continue. If any of these conditions is still true, then the procedure continues by first determining if the local element is part of either the flow route $r$ which is being monitored or the branch flow route that has been found up to this point. If this is the case, this means that a cycle has been detected while the branch flow route was being traversed: either a cycle which meets the flow route again at any point within its sequence of elements, or one that meets the partial branch flow route that has been traversed. In both cases the procedure performs no further action, thus aborting the branch monitoring operation. It is clear that beyond this point, no valid leak or mixture could be found which conforms to our definition of a branch flow route.
If no such cycle is found, then the procedure continues by determining if a leak or a mixture exists at the present component: if leaks are to be detected and the local element \( e \) is a product sink, then a leak has been found and the procedure \( \text{SendBackAlertMessage} \) is called; likewise, if mixtures are to be detected and the local element \( e \) is a product source, then a mixture has been found and the procedure \( \text{SendBackAlertMessage} \) is also called. In both cases, an indication of the type of alert is handed to the procedure, and the flag \( \alpha \) which indicates if an alert has been issued is set to 1. Finally, if no alert has been issued, the procedure \( \text{ForwardMonitorBranchMessage} \) is called to continue with the search of leaks and mixtures along the branch flow route. This procedure is defined as follows:

\[
\text{procedure } \text{ForwardMonitorBranchMessage}(e, c, m, \lambda_e, \mu_e) \\text{ for } \ c' \in C_e \text{ do}
\begin{align*}
&\text{if } c' \neq c \text{ then} \\
&\quad \phi_e \leftarrow \sigma(e) \\
&\quad \lambda_{c'} \leftarrow [\lambda_e \text{ and } \phi_e^O(c')] \\
&\quad \mu_{c'} \leftarrow [\mu_e \text{ and } \phi_e^I(c')] \\
&\quad \text{if } \lambda_{c'} \text{ or } \mu_{c'} \text{ then} \\
&\quad \quad m' \leftarrow \text{CreateMessage(MonitorBranch)} \\
&\quad \quad m'.\text{MonitorLeaks} \leftarrow \lambda_{c'} \\
&\quad \quad m'.\text{MonitorMixtures} \leftarrow \mu_{c'} \\
&\quad \quad m'.\text{FlowRoute} \leftarrow m.\text{FlowRoute} \\
&\quad \quad m'.\text{BranchFlowRoute} \leftarrow \text{Append}(m.\text{BranchFlowRoute}, e) \\
&\quad \quad \text{SendMessage}(c', m') \\
&\text{end if}
\end{align*}
\text{end if}
\text{end for}
\text{end procedure}
\]

This procedure first tries to find a connector \( c' \) of the local element \( e \) such that \( cec' \) conforms a middle flow step. For every such connector that is found, the algorithm computes updated conditions \( \lambda_{c'} \) and \( \mu_{c'} \) from the given \( \lambda_e \) and \( \mu_e \) and the current flow allowance setting of the connector \( c' \). If any of these conditions still hold, then a branch monitoring message is created with their values, which collectively indicate if leaks, mixtures, or both, are to be detected. The flow route that is being monitored is transferred as is to the new message, and the branch flow route is augmented with the local element \( e \). Finally, the new message is sent along the connector \( c' \), that is, in a forward direction along the branch flow route that is being searched.

### Reporting Leak and Mixture Alerts

The final stage of the product flow path monitoring algorithm deals with transferring leak and mixture alerts back towards the component of the initial element in the flow route that is being monitored, and with reporting the alert to this component when this initial location is reached. The procedure \( \text{SendBackAlertMessage} \) initiates this process and is defined as follows:

\[
\text{procedure } \text{SendBackAlertMessage}(e, c, m, t) \\
\quad m' \leftarrow \text{CreateMessage}(t) \\
\quad m'.\text{FlowRoute} \leftarrow m.\text{FlowRoute}
\]

84
5.2 Decentralised Algorithms

\[
m'.\text{BranchFlowRoute} \leftarrow \text{Append}(m.\text{BranchFlowRoute}, e)\]
\[
\text{SendMessage}(c, m')
\]

This procedure simply creates a message with the given type of alert, and sends it backwards along the branch flow route, that is, over connector \(c\). The flow route is copied from the original branch monitoring message, and the definitive branch flow route is obtained by adding the local element to the partial branch flow route that was found up to this point.

When an alert message reaches a component, the procedure \text{HandleAlertMessage} processes it as follows:

\[
\text{procedure HandleAlertMessage}(e, c, m)
\]
\[
i \leftarrow \text{IndexOf}(m.\text{FlowRoute}, e)
\]
\[
\text{if } i = 0 \text{ then}
\]
\[
\text{ReportAlert}(m.\text{MessageType}, m.\text{FlowRoute}, m.\text{BranchFlowRoute})
\]
\[
\text{else}
\]
\[
\text{ReturnAlertMessage}(e, c, m, i)
\]
\[
\text{end if}
\]
\[
\text{end procedure}
\]

Similar to the \text{HandleMonitorResultMessage} procedure, this procedure determines if the local element \(e\) is the initial location of the flow route stored in the message. If this is the case, then the message has reached its final destination and the alert is given to the current component by calling the procedure \text{ReportAlert}. The type of alert, the flow route that is being monitored and the branch flow route are taken from the alert message. If this initial location has not yet been reached, then the procedure \text{ReturnAlertMessage} is called in order to forward the alert message towards the component at the initial location in the flow route that is being monitored. This procedure is in turn defined as follows:

\[
\text{procedure ReturnAlertMessage}(e, c, m, i)
\]
\[
j \leftarrow \text{IndexOf}(m.\text{BranchFlowRoute}, e)
\]
\[
\text{if } i > 0 \text{ or } j > 0 \text{ then}
\]
\[
\text{if } i > 0 \text{ then}
\]
\[
e' \leftarrow m.\text{FlowRoute}[i - 1]
\]
\[
\text{else}
\]
\[
e' \leftarrow m.\text{BranchFlowRoute}[j - 1]
\]
\[
\text{end if}
\]
\[
\text{for } c' \in C_c \text{ do}
\]
\[
\text{if } c' \neq c \text{ and } e' = \text{GetConnectedElement}(c') \text{ then}
\]
\[
m' \leftarrow \text{CreateMessage}(m.\text{MessageType})
\]
\[
m'.\text{FlowRoute} \leftarrow m.\text{FlowRoute}
\]
\[
m'.\text{BranchFlowRoute} \leftarrow m.\text{BranchFlowRoute}
\]
\[
\text{SendMessage}(c', m')
\]
\[
\text{end if}
\]
\[
\text{end for}
\]
\[
\text{end if}
\]
\[
\text{end procedure}
\]
This procedure must first determine the location of the local element $e$ within the flow route $r$ which is being monitored, or within the branch flow route $b$ that corresponds to the alert. In any case, $e$ must be an intermediate element in one of these two flow routes, because if $e$ is the initial element of $r$, then this procedure would not have been called, and if $e$ is the initial element of $b$, then it must also be an element of $r$. If $e$ is indeed an element of $r$, then the element $e'$ corresponds to its predecessor in $r$; otherwise, $e'$ corresponds to the predecessor of $e$ in $b$. After this predecessor element $e'$ has been identified, the procedure tries to find a local connector $c'$ with a connection to the element $e'$. When this connector is found, a copy of the original alert message is created and sent along $c'$, which directs it towards the initial element of the flow route $r$ that is being monitored.

**Continuous Execution**

Because monitoring is a task which inherently operates in a continuous manner, it is assumed that this algorithm is executed repeatedly during the operation of a product flow path management system. Every execution of the algorithm consists of a sequence of messages which are exchanged among the components of the decentralised system, and these executions are carried out repeatedly at given time intervals. The fulfilment of any real-time properties by the part of the flow path monitoring algorithm ultimately depends on the lengths of the flow routes involved in the monitoring, and the speed of the communication links. An analysis of the computational complexity of this algorithm may be found at the end of this chapter.

**5.2.5 Flow Path Allocation**

The third and final decentralised algorithm that is presented in this work is called *flow path allocation*, and has the purpose of allocating plant elements for their usage by a product flow path. As explained in Chapter 2, a product flow path is a safe and exclusive area for the transport of products in a plant. The technique presented in this section was developed with the goal of providing a decentralised mechanism for ensuring this principle.

The analogy of railway control system presented in Chapter 1 motivates the goal of flow path allocation. In railway operation, before any train travels along a given track segment, it first requests that the segment be cleared for use, and waits for a corresponding authorisation signal. This signal is given to the train by the control system once the track segment has been successfully reserved, the corresponding switches have been set and locked into position, and the corresponding blocking signals have been issued to any conflicting train routes. This means that by following this sequence of operations, the safety of the train as it travels along the track segment is guaranteed. Regarding product flow paths, a similar technique is needed in order to guarantee safe and exclusive use of the plant elements which are part of a flow path. Concretely, it is important to assert the following two conditions:

1. **Mutual exclusion**: The plant elements which participate in a product flow path should not participate in any other product flow path at the same time.

2. **Enclosure**: When a product flow path is in operation, its corresponding flow route must be enclosed at the current flow allowance state $\sigma$ of the plant. This means that the flow path is free from potential leaks and mixtures of products.
In order to ensure that product flow paths have a mutually exclusive use of plant elements, the flow path allocation algorithm reserves all elements which participate in the flow path before declaring the flow path as allocated, and prevents any element from being reserved by two distinct flow paths simultaneously. An element that is reserved in this way is said to be allocated by the corresponding flow path. Furthermore, in order to ensure that the flow route of a product flow path is enclosed, the flow path allocation algorithm determines which plant elements should be constrained to use a determined flow allowance setting for this property to be satisfied. These constraints are registered such that no conflicting constraints may ever occur simultaneously in an element of the plant. An element whose flow allowance setting is restricted in this manner is said to be constrained by the corresponding flow path. The flow path allocation algorithm constrains only switchable elements, that is, those elements \( e \) whose flow allowance model \( \Phi(e) \) contains two or more flow allowance settings. All other elements are “naturally” constrained by their own singleton flow allowance models, and in order to avoid confusing interpretations, the algorithm does not create trivial constraints for these unswitchable elements.

As explained in Chapter 2, the flow path allocation algorithm offers two operations which can be applied to a product flow path: the allocation of the product flow path performs the allocation of the corresponding plant elements and the application of the necessary constraints to flow allowance settings; in turn, the deallocation of the product flow path reverses this effect by deallocating the corresponding plant elements and removing the previously added constraints. Additionally, an allocation operation may fail because of a conflict with respect to the allocation of a second product flow path; in this case, the conflict is reported and the allocation operation is halted. From this point, the allocation may be retried, for instance after the cause of the conflict has been removed, or a deallocation operation may be carried out in order to rollback the original allocation. Conflicts may also occur for any operation that tries to execute in a component that is busy executing another operation. In this case, the conflict is reported and the user of the system has the option of retrying the operation at a later time.

**Global Variables**

The algorithms of product flow path analysis and product flow path monitoring that have been presented in this chapter are stateless or memoryless, which means that every component processes incoming messages and emits outgoing messages without storing any information locally. In other words, the complete information that the algorithms handle is stored in the messages and in the local variables of the procedures, which only exist during the time that the corresponding procedure is executing. In contrast, the flow path allocation algorithm is a stateful algorithm, which means that components store information locally in addition to sending and receiving messages. This information is stored in global variables within each component which are accessible by the procedures of the flow path allocation algorithm, and which exist throughout the execution of the system.

The global variables which are used by the product flow path allocation algorithm may be classified as allocation variables, which describe the allocation state of the local element \( e \) of the component, and communication variables, which describe the communication state of the component. The allocation variables of a component are the following:

- \( \text{Alloc} \in R \cup \{0\} \): Stores the flow route \( r \) of the flow path that has allocated this component. If the component is currently not allocated, then this variable has the value
• Enclose $\subseteq R$: Stores the set of flow routes which correspond to the flow paths that have constrained this component to a flow allowance setting in order to guarantee their enclosure. If $\text{Enclose} = \emptyset$, then the component is free from any enclosure constraints.

• Setting $\in \Phi(e)$: Stores the flow allowance setting of the local element $e$ which must be maintained in order to guarantee the enclosure of the flow paths that have issued a constraint.

In turn, the communication variables of a component are the following:

• Operation $\in \{\text{None}, \text{Allocate, AllocateBranch, Deallocate, DeallocateBranch}\}$: Indicates the operation that is currently being executed by this component. The value $\text{None}$ indicates that the component is idle and may accept allocation or deallocation requests; the values $\text{Allocate, AllocateBranch, Deallocate and DeallocateBranch}$ respectively mean that an allocation, a branch allocation, a deallocation and a branch deallocation is currently being executed at the component.

• Parent $\in C_e \cup \{0\}$: Stores the local connector from where an original request message was received, in the case that an operation is currently being carried out as a consequence of having received this message. Otherwise, this variable stores the value 0.

• Children $\subseteq C_e$: Stores the set of local connectors where request messages were sent out, and where response messages are expected.

• FlowRoute $\in R \cup \{0\}$: Stores the flow route that corresponds to the allocation or deallocation operation that is being carried out. If the component is idle, then this variable has the value 0.

• BranchFlowRoute $\in R \cup \{0\}$: Stores the branch flow route that contains the local element of the component, when the allocation or deallocation of the branch flow route is being carried out as part of an operation on a flow route. If the component is idle or if it is not being allocated or deallocated as a branch flow route, then this variable has the value 0.

• Response $\subseteq M$: Stores the set of response messages that have been received by the component as part of the allocation or deallocation of a flow route.

The global variables of every component are initialised once, as soon as the component begins its execution as part of the decentralised system, by issuing a call to the procedure Initialise which is defined as follows:

```plaintext
procedure Initialise()
    Alloc $\leftarrow 0$
    Enclose $\leftarrow \emptyset$
    Operation $\leftarrow \text{None}$
    Parent $\leftarrow 0$
    Children $\leftarrow \emptyset$
```
5.2 Decentralised Algorithms

FlowRoute ← 0
BranchFlowRoute ← 0
Response ← ∅
end procedure

The initial state of the global variables indicates that the local element is not allocated, that no enclosure constraints have been set upon it, that no operation is being carried out, that no request message is being processed, and that no incoming response messages are expected or have been received. The global variable Setting is not initialised and may therefore have any value, because its value is only meaningful once an enclosure constraint has been added to the local element. If at any time during the execution of the system it is desired to “reset” the system to its initial state, this may be accomplished by invoking this procedure on every participating component, and by additionally verifying that no message is currently in transit within a communication link of the system.

Predefined Procedures

In addition to global variables, the product flow path allocation algorithm makes use of predefined procedures that are available at every component and whose behaviour depends on the actual implementation of the product flow path management system. These procedures are used by the component of the initial element $e$ of a flow route $r$ as part of an allocation or deallocation operation on $r$, and are described as follows:

- **ReportAllocation($r, S$)** This procedure reports to the component that the flow route $r$ has been successfully allocated. The argument $S$ stores a set of pairs of the form $(e, s)$, where for every such pair, $e$ is a plant element that has received a flow allowance setting constraint in order to enclose $r$, and $s$ stores the corresponding constrained flow allowance setting. The component may then update the corresponding flow path object in order to reflect this new allocated state.

- **ReportDeallocation($r$)** This procedure reports to the component that the flow route $r$ has been successfully deallocated. The component may then update the corresponding flow path object in order to reflect this new deallocated state.

- **ReportConflict($r, S$)** This procedure reports to the component that an allocation or deallocation conflict has been detected when performing an operation on the flow route $r$. In turn, this conflict occurs because the components of one or more elements cannot be allocated, deallocated or constrained for the flow route $r$ as they are already allocated or constrained for a flow route in the set $S$, or are busy performing an operation for a flow route in $S$. This information may be used by the component to update the state of the flow path object which corresponds to $r$.

Allocating Components

The allocation of plant elements, and the addition of enclosure constraints to these elements, is done by manipulating the global allocation variables of the corresponding component. This manipulation is restricted to the following procedures, which are used by other procedures of the flow path allocation algorithm to allocate and deallocate the component, as well as to add and remove enclosure constraints.
The procedure `Allocate` has the task of allocating the local element \( e \) for use by the flow route \( r \). It returns a Boolean value that indicates if the allocation was successful or not. If the component is currently free, then it is allocated by assigning \( r \) to the variable `Alloc`. If the component was already allocated, then the allocation succeeds if the current allocation corresponds precisely to \( r \), and the operation fails otherwise. This means that a postcondition of this procedure is that `Alloc = r` whenever the allocation succeeds.

The deallocation of the local component is done by the procedure `Deallocate` which is defined as follows:

```plaintext
procedure Deallocate(e, r)
  if Alloc = r then
    Alloc ← 0
  end if
end procedure
```

The deallocation of a component always succeeds and thus returns no value. If the component was currently allocated by \( r \), then the variable `Alloc` is cleared. Otherwise, the call to this procedure has no effect. The rationale behind this behaviour is that a flow route which is being deallocated should only be able to deallocate its own allocated components, and none other. If a component is already allocated by \( r \), then only an explicit deallocation of \( r \) will clear this allocation. Therefore, after the call `Deallocate(e, r)`, what we may assert is that `Alloc ≠ r`.

The manipulation of constraints on the flow allowance setting of the local element \( e \) for the purpose of enclosing product flow paths is performed by the procedures `AddEnclosureConstraint` and `RemoveEnclosureConstraint`, which are defined in the following.

```plaintext
procedure AddEnclosureConstraint(e, r, \( \phi_e \)) returns \{0, 1\}
  if |\( \Phi(e) \)| = 1 then
    return 1
  else if Enclose = \( \emptyset \) then
    Enclose ← \{r\}
    Setting ← \( \phi_e \)
    return 1
  else if Setting = \( \phi_e \) then
    if \( r \notin Enclose \) then
      Enclose ← Enclose \cup \{r\}
    end if
  end if
  return 1
```

The manipulation of constraints on the flow allowance setting of the local element \( e \) for the purpose of enclosing product flow paths is performed by the procedures `AddEnclosureConstraint` and `RemoveEnclosureConstraint`, which are defined in the following.
This procedure has the task of adding a constraint to the local element $e$ which enforces the usage of its flow allowance setting $\phi_e$ in order to enclose the flow route $r$, and returns a Boolean value which indicates if the operation was successful or not. The procedure first determines if the local element $e$ is switchable or not, by testing if it has only one flow allowance setting that it may use. If this is the case, then no constraint is added to the component, but the procedure returns the value 1 which indicates that the operation succeeded. This means that constraining any plant element which is not switchable is a trivial operation which has no effect and always succeeds. If $e$ is switchable, then the procedure succeeds by adding an enclosure constraint for $r$, and by setting the global variable Setting to the indicated flow allowance setting, if no constraint was previously added. If an existing constraint is found, then the operation succeeds only if the already constrained flow allowance setting coincides with the newly indicated one. In this case, $r$ is added to the set of flow routes that have issued enclosure constraints, which tells us that a single plant element may be constrained to a given flow allowance setting by multiple flow routes simultaneously. In turn, the removal of an enclosure constraint for the flow route $r$ is a simple task which is defined as follows:

```plaintext
procedure RemoveEnclosureConstraint(e, r)
  if $r \in \text{Enclose}$ then
    \text{Enclose} \leftarrow \text{Enclose} \setminus \{r\}
  end if
end procedure
```

This procedure tests if the given flow route $r$ has already been granted an enclosure constraint for the local element $e$. If this is the case, then $r$ is simply removed from the set stored at the global variable $\text{Enclose}$.

### Managing Communication States

As well as having an allocation state that is described by its global allocation variables, components hold a communication state which indicates the situation of a component with respect to decentralised operations. At any given time, a component is either idle, which means that it is not participating in any operation, or is active within a decentralised operation. A component that is idle may become active by issuing a call to the procedure $\text{StartOperation}$ that is defined in the following:

```plaintext
procedure StartOperation(o, p, r, b) returns \{0, 1\}
  if Operation = None then
    Operation \leftarrow o
    Parent \leftarrow p
    Children \leftarrow \emptyset
    FlowRoute \leftarrow r
    BranchFlowRoute \leftarrow b
  return 1
```


```plaintext
else if Operation = o and Parent = p and
     FlowRoute = r and BranchFlowRoute = b then
  return 1
else
  return 0
end if
end procedure
```

If the component is idle, that is, when the variable `Operation` has the value `None`, then a call to this procedure causes the component to become active within an operation whose type is indicated by the parameter `o`. The parent connector, flow route and branch flow route that correspond to this operation are also given to the procedure and stored as part of the communication state. Finally, the procedure returns the value 1 indicating that the operation was able to begin. If the component was already active, then this call will succeed if the operation that is requested matches the currently active operation. This allows an operation to be restarted, or retried, while it is still active, without requiring a previous deactivation. If none of the conditions considered so far hold, then the call to the procedure fails and the value 0 is returned.

The ending of an operation, which moves the component back to an idle state, is accomplished by calling the procedure `EndOperation` which simply resets the entire communication state. This procedure is defined as follows:

```plaintext
procedure EndOperation()
  Operation ← None
  Parent ← 0
  Children ← ∅
  FlowRoute ← 0
  BranchFlowRoute ← 0
  Response ← ∅
end procedure
```

In this manner, every component may manage its communication state in a simple and ordered way. Operations may start either by a local event at a component (mainly at the component of the initial element of a flow route), or upon receiving a request message from a neighbouring component. The communication state of the component holds the information that an active component needs to “remember” while it waits for responses from other components. This state also prevents a component from participating in more than one operation at the same time. When the component no longer needs to wait for incoming responses, it may end the operation and enter an idle state once more.

**Selecting Flow Allowance Settings**

As previously explained, in order to ensure the enclosure of a flow route that is allocated, certain switchable elements along neighbouring branch flow routes are constrained to use one of their flow allowance settings while the allocation of the flow route is in effect. This flow allowance setting should inhibit the flow of products along the branch flow route in order to avoid leaks and mixtures as defined by the formal model of enclosed flow routes. Furthermore, this flow allowance setting must be selected for each switchable element $e$ from the set
5.2 Decentralised Algorithms

$\Phi(e)$ of flow allowance settings of $e$. In turn, this selection may occur in two different ways:

1. **Manually**: A flow allowance setting $\phi_e \in \Phi(e)$ is designated a priori as the “protecting” or “blocking” setting, and only this setting may receive an enclosure constraint.

2. **Automatically**: A flow allowance setting $\phi_e \in \Phi(e)$ is chosen by an algorithm based on the context of the element with respect to the flow route that is being allocated.

The manual technique may be useful or even necessary in certain plants whose elements have complex flow allowance behaviours, or when arbitrary and predefined enclosure constraints are required. This additional information must be given together with the flow allowance model of the plant. On the other hand, the automatic technique requires no additional information apart from the flow allowance model of the plant. A simple algorithm that only takes the local element $e$ into account works well for most common plant scenarios. However, it is possible to have situations with complex plant elements and element interconnection structures where such a simple automatic approach to flow allowance setting selection does not yield the best setting that could have been chosen, and more complex algorithms which consider combinations of flow allowance settings for multiple elements would be needed in order to obtain better results. To make this clear, consider an element $e$ within a branch flow route, and consider that the branch flow route “splits” into two branches at $e$. If the element $e$ has the possibility to block product flow for only one of the two branches at a time, then it would be equivalent to block flow for any one of the two branches. However, the actual composition of the branches may require $e$ to block one branch instead of the other, and $e$ cannot do this without taking this additional information into account. Therefore, a more complex algorithmic approach would be necessary in order to select the best flow allowance setting for $e$.

With these ideas under consideration, this work presents a simple automatic solution to flow allowance setting selection that may be overridden by a user that defines the flow allowance setting that is to be chosen in a manual way. The reason for using this approach is a practical one, and makes the algorithm suitable for a wide range of application scenarios without the need to be more complex.

In Figure 5.2 it was shown how the leak and mixture flows within a plant element that is part of a branch flow route can be represented and analysed. Based on this analysis, the procedure *SelectBranchFlowRouteSetting* that is presented in the following selects a flow allowance setting for an element $e$ that is a non-initial element of a branch flow route $b$, in such a way that the leak and mixture flows represented by $\lambda_{c'}$ and $\mu_{c'}$ are blocked for every connector $c'$ that leads away from the flow route $r$ in the best possible way. For this, the procedure obtains the local element $e$, the connector $c$ which leads to the flow route $r$, and the conditions $\lambda$ and $\mu$ that respectively represent the possibility of leak and mixture flows before connector $c$, as shown in Figure 5.2.

```
procedure SelectBranchFlowRouteSetting(e, c, \lambda, \mu) returns \Phi(e)
    $\phi_{e'} \leftarrow$ GetManualBranchFlowRouteSetting(e)
    if $\phi_{e'} \in \Phi(e)$ then
        return $\phi_{e'}$
    else if $|\Phi(e)| = 1$ and $\Phi(e) = \{\phi_{e''}\}$ then
        return $\phi_{e''}$
    else
        $n \leftarrow -1$
```
The flow allowance setting selection algorithm first invokes the procedure GetManualBranchFlowRouteSetting, which basically returns the flow allowance setting of \( e \) that was manually selected, or some value not in \( \Phi(e) \) if no manual selection was made for \( e \). If a manually selected flow allowance setting is obtained, then it is immediately returned. Otherwise, the procedure determines if the element \( e \) is switchable, and directly returns its only flow allowance setting if this is not the case. If \( e \) is indeed switchable, the algorithm proceeds to select the best flow allowance setting from \( \Phi(e) \) by evaluating how well each flow allowance setting blocks leak and mixture flows, assigning each setting a score based on this evaluation. For this, the variable \( n \) stores the highest score that a flow allowance setting has received so far, and \( \phi_{e}^{'''} \) stores the flow allowance setting that obtained the score \( n \). The selection algorithm iterates over each flow allowance setting \( \phi_{e} \) of \( e \), and computes the score of \( \phi_{e} \) as a positive integer that is stored in \( n' \). For every connector \( c' \) of \( e \) that leads away from the flow route that is being allocated, it is determined if the flow allowance setting \( \phi_{e} \) that is currently being evaluated is able to block leak and mixture flows through \( e \) and along the connectors \( c \) and \( c' \). For each successful blocking that is achieved, the flow allowance setting receives a “point” by incrementing its score \( n' \) by 1. Once all connectors have been considered, the score of \( \phi_{e} \) is compared with the highest score that has been obtained so far. If \( \phi_{e} \) has a higher score, then \( \phi_{e} \) becomes the best flow allowance setting found so far, and it is stored in \( \phi_{e}^{'''} \). After having considered all flow allowance settings of \( e \), the procedure returns the best flow allowance setting that was found.

When using this flow allowance selection technique, the flow path allocation algorithm performs a greedy search [50] for an assignment of enclosure constraints to the elements involved in the allocation. In effect, the selection criterion employed by this algorithm over the set \( \Phi(e) \) aims to find a local optimum in a greedy manner. Additionally, the flow path allocation algorithm performs no backtracking, which means that once a flow allowance setting has been selected for a given plant element in this way, then this selection is kept as definitive and no additional settings are tried. Because of this, applying this algorithm to a network of
elements may not yield an globally optimal solution. Furthermore, the outcome of the selection technique is not determined in the case that two or more flow allowance settings share the best score, as the resulting flow allowance setting depends then on the order in which the set $\Phi(e)$ is visited. All in all, this algorithm offers a simple and practical solution to the problem of selecting a flow allowance setting for a plant element in an automatic manner with the purpose of inhibiting product leaks and product mixtures in a plant, and is most effective when applied to common plant elements that have simple flow allowance settings.

**Handling Incoming Messages**

Similarly to the decentralised algorithms that were previously presented in this chapter, the procedure `HandleMessage` that is part of the flow path allocation algorithm dispatches every message that is received to a procedure that handles it, depending on the type of the message. However, because of the presence of communication states, this procedure must react differently to request and response messages. This procedure is defined as follows:

```plaintext
procedure HandleMessage(e, c, m)
    if m.MessageType ∈ {Allocate, AllocateBranch, Deallocate, DeallocateBranch} then
        if StartOperation(m.MessageType, c, m.FlowRoute, m.BranchFlowRoute) then
            if m.MessageType = Allocate then
                HandleAllocateMessage(e, c, m)
            else if m.MessageType = AllocateBranch then
                HandleAllocateBranchMessage(e, c, m)
            else if m.MessageType = Deallocate then
                HandleDeallocateMessage(e, c, m)
            else if m.MessageType = DeallocateBranch then
                HandleDeallocateBranchMessage(e, c, m)
            end if
        else if m.MessageType = AllocateBranch and Operation ∈ {Allocate, AllocateBranch} and m.FlowRoute = FlowRoute then
            SendBackAllocationSuccessMessage(e, c, m)
        else if m.MessageType = DeallocateBranch and Operation ∈ {Deallocate, DeallocateBranch} and m.FlowRoute = FlowRoute then
            SendBackDeallocationSuccessMessage(e, c, m)
        else
            SendBackConflictMessage(e, c, m, {FlowRoute})
        end if
    else if m.MessageType ∈ {AllocationSuccess, DeallocationSuccess, Conflict} then
        if c ∈ Children then
            Response ← Response ∪ {m}
            Children ← Children \ {c}
            FinishOperation(e)
        end if
    end if
end if
```
end procedure

If the incoming message is an operation request, then the procedure first tries to initiate this operation by calling the procedure \texttt{StartOperation} with the request information from the message and the connector where the message entered as a parent connector. If this call succeeds, then the message is processed normally. Otherwise, it is determined if the message corresponds to a branch request for the same operation that is currently being executed at the component. If this is the case, then a corresponding allocation or deallocation success message is immediately returned, because the actual result of the operation for the current component will be reported to a different parent component once the operation finishes, and the component which sent the currently processed message cannot be kept waiting for a response message. If the request was not a branch request for the currently executed operation, then a conflict message is sent back along the same connector, and the flow route of the currently active operation is given as the conflicting route because the requested operation could not begin as another operation is currently in effect. If the incoming message corresponds to a response, then the procedure does not forward the message to a procedure for its handling, but rather stores it as part of the communication state if the corresponding local connector was expecting such a message. Additionally, the connector is removed from the \texttt{Children} set, indicating that it no longer expects a response message. After this, the procedure \texttt{FinishOperation} is called, which tries to finish the currently active operation based on the current communication state; this procedure will be described later in this section.

Messages with the types \texttt{Allocate} and \texttt{Deallocate} are respectively used to request the allocation and deallocation of elements along a flow route. Messages with the types \texttt{AllocateBranch} and \texttt{DeallocateBranch} respectively request the allocation and deallocation of elements along a branch flow route that stems from the flow route that is being allocated. The message types \texttt{AllocationSuccess} and \texttt{DeallocationSuccess} are used respectively to indicate that the allocation and deallocation of a partial flow route or branch flow route has succeeded. Likewise, the message type \texttt{Conflict} is used in a similar way whenever a conflict is detected which hinders the allocation or deallocation of an element that participates in the corresponding flow path.

As a final note, this procedure rejects request messages whenever a distinct operation is in effect, and additionally, ignores response messages that are received unexpectedly. This behaviour gives robustness to the algorithm, as it frees it from having to deal with conflicting messages.

### Allocating Flow Routes

The allocation of a flow route \( r \) begins with a call to the procedure \texttt{InitiateAllocation} at the component that corresponds to the initial element \( e \) of \( r \). This procedure is defined as follows:

```
procedure InitiateAllocation(e, r)
    if StartOperation(Allocate, 0, 0, r, 0) then
        if Allocate(e, r) then
            if LengthOf(r) = 1 then
                ReportAllocation(r, ∅)
```

96
5.2 Decentralised Algorithms

The procedure first tries to initiate the allocation operation, and immediately reports a conflict with the flow route of the currently active operation if this fails. Otherwise, it tries to allocate the local element for the given flow route \( r \). Again, if this fails a conflict is reported, this time with respect to the flow route that has already allocated the element \( e \), and the newly started operation is concluded. If the allocation of \( e \) succeeds, then the procedure \texttt{InitiateDecentralisedAllocation} is called if the flow route has more than one element. The case of a single-element trivial flow route is handled for completeness by reporting its allocation without any enclosure constraints, and by ending the current operation.

A decentralised allocation of a flow route \( r \) is started by the procedure \texttt{InitiateDecentralisedAllocation}, defined as follows:

\begin{verbatim}
procedure InitiateDecentralisedAllocation(e,r)
   for \( c \in C_e \) do
      if \( r[1] = \text{GetConnectedElement}(c) \) then
         \( m \leftarrow \text{CreateMessage}(\text{Allocate}) \)
         \( m.\text{FlowRoute} \leftarrow r \)
         \( m.\text{BranchFlowRoute} \leftarrow \{\} \)
         \( \text{SendMessage}(c,m) \)
         \( \text{Children} \leftarrow \text{Children} \cup \{c\} \)
         \( \text{SendInitialBranchAllocationMessages}(e,0,c,r) \)
      end if
   end for
end procedure
\end{verbatim}

This procedure first identifies the local connector \( c \) which leads to the second element in the flow route \( r \), and sends an allocate request message to this successor element along \( c \). The connector \( c \) is hereby registered as a child connector that awaits a response. Afterwards, the procedure \texttt{SendInitialBranchAllocationMessages} is called in order to send out the corresponding allocation messages for the branch flow routes that stem from \( r \) at \( e \).

When a component receives an allocate request message, it handles it as defined by the following procedure:

\begin{verbatim}
procedure HandleAllocateMessage(e,c,m)
   if \( \text{Allocate}(e,m.\text{FlowRoute}) \) then
      \( i \leftarrow \text{IndexOf}(m.\text{FlowRoute},e) \)
      if \( i = \text{LengthOf}(m.\text{FlowRoute}) - 1 \) then
         \( \ldots \)
      end if
   end if
end procedure
\end{verbatim}
HandleAllocateMessageAtFinalPosition(\(e, c, m\))
else
HandleAllocateMessageAtMiddlePosition(\(e, c, m, i\))
end if
else
SendBackConflictMessage(\(e, c, m, \{\text{Alloc}\}\))
EndOperation()
end if
end procedure

As in the case of InitiateAllocation, the procedure HandleAllocateMessage first tries to allocate the local element \(e\) for the flow route stored in the message. If this allocation fails, then a conflict message is sent back along connector \(c\) and the operation is ended. Otherwise, the allocation continues by calling the procedure HandleAllocateMessageAtMiddlePosition if the local element is the final element in the flow route that is being allocated, or by calling HandleAllocateMessageAtMiddlePosition otherwise. These two procedures are defined in the following.

This procedure identifies the local connector \(c'\) that links the element \(e\) to its successor element in the flow route, sends a new allocation request message to the component of this successor along \(c'\), and adds \(c'\) to the set of child connectors awaiting responses. Also, the procedure SendInitialBranchAllocationMessages is called to initiate the allocation of every flow branch that stems from the flow route at the element \(e\).

The task of handling a flow route allocation request at the final element of the flow route is accomplished by the following procedure:

It first initiates the allocation of branch flow routes, and then immediately tests if there are no child connectors waiting for incoming messages. If this is the case, then the allocation of the
flow route has succeeded for this component, and the procedure SendBackAllocationSuccessMessage is called to return an allocation success message backwards along the flow route. After this, the operation is concluded.

### Allocating Branch Flow Routes

The allocation of a branch flow route begins with a call to the procedure SendInitialBranchAllocationMessages, which is defined as follows:

```plaintext
procedure SendInitialBranchAllocationMessages(e, c, c', r)
λ_e ← [c' ∈ Ce]
µ_e ← [c ∈ Ce]
for c'' ∈ Ce do
  if c'' ≠ c and c'' ≠ c' then
    λ_{c''} ← [λ_e and ∃φ_e ∈ Φ(e) : φ_e^L(c'')]
    µ_{c''} ← [µ_e and ∃φ_e ∈ Φ(e) : φ_e^I(c'')]
    if λ_{c''} or µ_{c''} then
      m ← CreateMessage(AllocateBranch)
      m.LeakFlow ← λ_{c''}
      m.MixtureFlow ← µ_{c''}
      m.FlowRoute ← r
      m.BranchFlowRoute ← ⟨e⟩
      SendMessage(c'', m)
      Children ← Children ∪ {c''}
  end if
end for
end procedure
```

Figure 5.2 depicts the situation of an initial branch flow route element that this procedure handles. First, the conditions λ_e and µ_e are computed based on the presence of the connectors c and c', which indicates if the element e is part of an initial, middle or final flow step. For every local connector c'' that leads to a branch flow route, the conditions λ_{c''} and µ_{c''} are evaluated based on the values of λ_e and µ_e and on the existence of a flow allowance setting that permits product flow through c'' in the corresponding direction. If any of these conditions is true, a branch allocation request is sent and c'' is added to the set of child connectors that corresponds to the current operation. The conditions λ_{c''} and µ_{c''}, the flow route r and the partial branch flow route considered up to now are all sent as part of the message.

The reception of a branch allocation request message is handled by the procedure HandleAllocateBranchMessage defined in the following.

```plaintext
procedure HandleAllocateBranchMessage(e, c, m)
if Contains(m.FlowRoute, e) or Contains(m.BranchFlowRoute, e) then
  SendBackAllocationSuccessMessage(e, c, m)
  EndOperation()
else
  φ_e ← SelectBranchFlowRouteSetting(e, c, m.LeakFlow, m.MixtureFlow)
  if AddEnclosureConstraint(e, m.FlowRoute, φ_e) then
    λ_e ← [m.LeakFlow and φ_e^L(c)]
  end if
end if
```

99
μ_e ← [m.MixtureFlow and φ_e^O(c)]
if λ_e or μ_e then
  if Allocate(e, m, FlowRoute) then
    ForwardAllocateBranchMessage(e, c, m, λ_e, μ_e, φ_e)
  else
    SendBackConflictMessage(e, c, m, {Alloc})
    EndOperation()
  end if
else
  SendBackAllocationSuccessMessage(e, c, m)
  EndOperation()
end if
else
  SendBackConflictMessage(e, c, m, Enclose)
  EndOperation()
end if
end if
end procedure

If a branch allocation request reaches the component of an element e that is already contained in the flow route or in the branch flow route of the message, then it means that the branch allocation operation has encountered a cycle. Because of this, the procedure HandleAllocateBranchMessage sends back an allocation success message as soon as it detects this situation, and proceeds by ending the current operation. If no cycle has been encountered, the procedure continues by selecting a flow allowance setting for the local element e in the context of the parent connector c and the conditions of leak and mixture flow contained in the request message. A corresponding enclosure constraint is added afterwards, and a conflict message is issued if this fails. Next, the values of λ_e and μ_e are computed for the element e and its chosen flow allowance setting. If neither of these values is true, it means that the leak and mixture flows have been successfully blocked for the present branch flow route. Consequently, an allocation success message is issued and the operation is concluded. At this point, the element e is not allocated, and may receive further enclosure constraints from other flow routes, provided that the same flow allowance setting is requested. However, if leak or mixture flows still exist, then the procedure continues by allocating the element e and calling the procedure ForwardAllocateBranchMessage. As expected, if the allocation of the local component fails, then a conflict message is returned and the operation ends.

The procedure ForwardAllocateBranchMessage continues the work of the previous procedure and is defined as follows:

procedure ForwardAllocateBranchMessage(e, c, m, λ_e, μ_e, φ_e)
for c' ∈ C_e do
  if c' ≠ c then
    λ_{c'} ← [λ_e and φ_e^O(c')]
    μ_{c'} ← [μ_e and φ_e^I(c')]
    if λ_{c'} or μ_{c'} then
      m' ← CreateMessage(AllocateBranch)
      m'.LeakFlow ← λ_{c'}
      m'.MixtureFlow ← μ_{c'}
    end if
  end if
end for
end procedure
5.2 Decentralised Algorithms

This procedure first identifies every connector \( c' \) of the local element \( e \) that leads to a continuation of the current branch flow route. For each of these connectors, the values of \( \lambda_{c'} \) and \( \mu_{c'} \) that indicate the possible leak and mixture flows through \( c' \) are computed, and an allocate branch request message is sent along \( c' \) if either of these conditions is true. With respect to the message received, the branch flow route in the new message additionally contains the element \( e \). Also, the connector \( c' \) is added to the set of child connectors awaiting a response message. After processing all local connectors, the procedure tests if any child connectors have been registered. If this is not the case, then an allocation success message is immediately returned and the operation is ended, because the present branch has been completely allocated and no leak or mixture flows remain.

### Reporting Conflicts

The task of reporting allocation conflicts is fulfilled by the procedure \( \text{SendBackConflictMessage} \), which creates a conflict message as a response to a given request message \( m \) and sends it along a given connector \( c \). Furthermore, a set \( S \) of conflicting flow routes is given to this procedure and added to the message. The procedure is defined as follows:

```plaintext
procedure \( \text{SendBackConflictMessage}(e, c, m, S) \)
\[
\begin{align*}
    m' & \leftarrow \text{CreateMessage}(\text{Conflict}) \\
    m'.Element & \leftarrow e \\
    m'.FlowRoute & \leftarrow m.\text{FlowRoute} \\
    m'.\text{BranchFlowRoute} & \leftarrow m.\text{BranchFlowRoute} \\
    m'.\text{ConflictFlowRoutes} & \leftarrow S \\
    \text{SendMessage}(c, m')
\end{align*}
\end{equation}
end procedure
```

### Reporting Allocation Success

Whenever a flow route or one of its associated branch flow routes is successfully allocated, the procedure \( \text{SendBackAllocationSuccessMessage} \) is invoked to issue a response message indicating this success. This procedure is defined as follows:

```plaintext
procedure \( \text{SendBackAllocationSuccessMessage}(e, c, m) \)
\[
\begin{align*}
    m' & \leftarrow \text{CreateMessage}(\text{AllocationSuccess})
\end{align*}
\end{equation}
end procedure
```
FlowRoute ← m′.FlowRoute
BranchFlowRoute ← m′.BranchFlowRoute
if Enclose ≠ ∅ then
    m′.Constraints ← {(e, Setting)}
else
    m′.Constraints ← ∅
end if
SendMessage(c, m′)
end procedure

The allocation success message that is created by this procedure contains the flow route and the branch flow route that came with the incoming request. Additionally, an allocation success message contains a set of constraint pairs of the form \((e, \phi_e)\), where \(e\) is an element and \(\phi_e\) is a flow allowance setting of \(e\). As this procedure initiates the report of a successful allocation, the set of constraint pairs stored in the newly created message only contains the pair corresponding to the local element of the component if an allocation constraint has been added to it; otherwise, the set of constraint pairs is empty. The procedure concludes by sending this message over the given connector \(c\).

Deallocating Flow Routes

The operation of deallocating a flow route is similar to the operation which allocates a flow route, and is somewhat simpler because deallocating plant elements always succeeds. This operation is started for a flow route \(r\) at the component of the initial element \(e\) of \(r\) by invoking the procedure InitiateDeallocation that is defined as follows:

procedure InitiateDeallocation(e, r)
    if StartOperation(Deallocate, 0, r, 0) then
        if LengthOf(r) = 1 then
            Deallocate(e, r)
            ReportDeallocation(r)
            EndOperation()
        else
            InitiateDecentralisedDeallocation(e, r)
        end if
    else
        ReportConflict(r, {FlowRoute})
    end if
end procedure

A deallocation operation is first started by this procedure, and a conflict is reported at the local component if this cannot be achieved. If the flow route \(r\) contains \(e\) as its single element, then the procedure deallocates \(e\) for \(r\) and reports the successful deallocation of \(r\). Finally, the operation is concluded at the local component. On the other hand, if \(r\) is a longer flow route, then the procedure InitiateDecentralisedDeallocation is called to continue the deallocation operation in a decentralised manner. This procedure is in turn defined as follows:

procedure InitiateDecentralisedDeallocation(e, r)
    for \(c \in C_e\) do
        /* implementation */
    end for
end procedure
if \( r[1] = \text{GetConnectedElement}(c) \) then
  \( m \leftarrow \text{CreateMessage(Deallocate)} \)
  \( m.\text{FlowRoute} \leftarrow r \)
  \( m.\text{BranchFlowRoute} \leftarrow \langle \rangle \)
  \( \text{SendMessage}(c,m) \)
  \( \text{Children} \leftarrow \text{Children} \cup \{c\} \)
  \( \text{SendInitialBranchDeallocationMessages}(e,0,c,r) \)
end if
end for
end procedure

After identifying the local connector \( c \) that leads to the second element in the flow route \( r \), this procedure creates a deallocation request message and sends it along \( c \). The flow route \( r \) is stored in the message, as well as an empty branch flow route. As in the case of the allocation operation, the connector \( c \) is included in the set of child connectors awaiting a response message. Finally the procedure \( \text{SendInitialBranchDeallocationMessages} \) is called to initiate the deallocation of any branch flow routes that begin at element \( e \).

When a component receives a deallocation request message, the procedure \( \text{HandleDeallocateMessage} \) is called by \( \text{HandleMessage} \) to service the request. This procedure is defined as follows:

procedure \( \text{HandleDeallocateMessage}(e,c,m) \)
  \( i \leftarrow \text{IndexOf}(m.\text{FlowRoute},e) \)
  if \( i = \text{LengthOf}(m.\text{FlowRoute}) - 1 \) then
    \( \text{HandleDeallocateMessageAtFinalPosition}(e,c,m) \)
  else
    \( \text{HandleDeallocateMessageAtMiddlePosition}(e,c,m,i) \)
  end if
end procedure

This procedure determines if the local element \( e \) has a middle or a final position in the flow route that is being deallocated. Based on this, one of the procedures \( \text{HandleDeallocateMessageAtMiddlePosition} \) and \( \text{HandleDeallocateMessageAtFinalPosition} \) is called. These procedures are defined in the following.

procedure \( \text{HandleDeallocateMessageAtMiddlePosition}(e,c,m,i) \)
  for \( c' \in C_e \) do
    if \( c' \neq c \) and \( m.\text{FlowRoute}[i+1] = \text{GetConnectedElement}(c') \) then
      \( m' \leftarrow \text{CreateMessage(Deallocate)} \)
      \( m'.\text{FlowRoute} \leftarrow m.\text{FlowRoute} \)
      \( m'.\text{BranchFlowRoute} \leftarrow \langle \rangle \)
      \( \text{SendMessage}(c',m') \)
      \( \text{Children} \leftarrow \text{Children} \cup \{c'\} \)
      \( \text{SendInitialBranchDeallocationMessages}(e,c,c',m.\text{FlowRoute}) \)
    end if
  end for
end procedure
The procedure \texttt{HandleDeallocateMessageAtMiddlePosition} identifies the local connector $c'$ that leads to the next element in the flow route that is being deallocated, and sends a deallocation request message to it. The connector $c'$ is added to the set of child connectors, and the procedure \texttt{SendInitialBranchDeallocationMessages} is called to start the deallocation of any branch flow routes that begin at the local element $e$. In turn, the procedure \texttt{HandleDeallocateMessageAtFinalPosition} is defined as follows:

\begin{verbatim}
procedure HandleDeallocateMessageAtFinalPosition(e, c, m)
    SendInitialBranchDeallocationMessages(e, c, 0, m, FlowRoute)
    if Children = ∅ then
        Deallocate(e, m, FlowRoute)
        SendBackDeallocationSuccessMessage(e, c, m)
        EndOperation()
    end if
end procedure
\end{verbatim}

This procedure begins by calling \texttt{SendInitialBranchDeallocationMessages}, which initiates the deallocation of the branch flow routes that stem from the flow route that is being allocated at its final element $e$. Afterwards, the procedure checks if any child connectors have been registered. If this is not the case, then it means that the deallocation operation has been successful for the flow route, and the local element $e$ is then deallocated. Also, a deallocation success message is issued by calling the procedure \texttt{SendBackDeallocationSuccessMessage}, and the operation is concluded at the local component.

### Deallocating Branch Flow Routes

The deallocation of every branch flow route begins with a call to the procedure \texttt{SendInitialBranchDeallocationMessages}, whose context of operation is the initial element $e$ of a branch flow route as shown in Figure 5.2.

\begin{verbatim}
procedure SendInitialBranchDeallocationMessages(e, c, c', r)
    for $c'' \in C_e$ do
        if $c'' \neq c$ and $c'' \neq c'$ then
            m ← CreateMessage(DeallocateBranch)
            m.FlowRoute ← r
            m.BranchFlowRoute ← \langle e \rangle
            SendMessage($c''$, m)
            Children ← Children ∪ \{c''\}
        end if
    end for
end procedure
\end{verbatim}

This procedure creates a deallocate branch request message for every local connector $c''$ that leads to a branch flow route, and sends this message along $c''$. The flow route $r$ is sent as part of the message, as well as the partial branch flow route considered up to now. Finally, the connector $c''$ is added to the set of child connectors of the local component.

When a component receives a branch deallocation request, the procedure \texttt{HandleDeallocateBranchMessage} is called to handle the message, and this procedure is defined as
5.2 Decentralised Algorithms

follows:

procedure HandleDeallocateBranchMessage(e, c, m)
    if Contains(m.FlowRoute, e) or Contains(m.BranchFlowRoute, e) then
        SendBackDeallocationSuccessMessage(e, c, m)
        EndOperation()
    else if m.FlowRoute = Alloc or m.FlowRoute ∈ Enclose then
        ForwardDeallocateBranchMessage(e, c, m)
    else
        SendBackDeallocationSuccessMessage(e, c, m)
        EndOperation()
    end if
end procedure

This procedure first determines if the branch deallocation request has closed a cycle by testing if the local element e is contained in the flow route or in the branch flow route of the incoming message. If this is the case, then the deallocation need not continue at the local component, and a deallocation success message is returned along connector c. After this, the operation is ended at the local component. If no cycle is detected, then the procedure continues by determining if the local element e is currently allocated by the flow route stored in the request message, or if e holds any enclosure constraints for this flow route. In either case, the procedure ForwardDeallocateBranchMessage is called to continue with the deallocation operation at the local component. Otherwise, if e is not allocated or constrained by this flow route, then the deallocation operation has succeeded along the present branch flow route, a corresponding success message is issued, and the operation is concluded. In turn, the procedure ForwardDeallocateBranchMessage is defined as follows:

procedure ForwardDeallocateBranchMessage(e, c, m)
    for c′ ∈ Ce do
        if c′ ≠ c then
            m′ ← CreateMessage(DeallocateBranch)
            m′.FlowRoute ← m.FlowRoute
            m′.BranchFlowRoute ← Append(m.BranchFlowRoute, e)
            SendMessage(c′, m′)
            Children ← Children ∪ {c′}
        end if
    end for
    if Children = ∅ then
        Deallocate(e, m.FlowRoute)
        RemoveEnclosureConstraint(e, m.FlowRoute)
        SendBackDeallocationSuccessMessage(e, c, m)
        EndOperation()
    end if
end procedure

For every local connector c′ that leads to a continuation of the present branch flow route and away from the flow route that is being deallocated, this procedure creates a branch deallocation request message and sends it along c′. The original flow route is included in the new message, as well as the original branch flow route with the local element e appended at
its end. Finally, the connector $c'$ is added to the set of child connectors of the component. After processing all local connectors, the procedure tests if any connector has been registered as a child connector. If this is not the case, then it means that the deallocation operation has reached the end location of the present branch flow route, and its success may be reported. In doing this, the element $e$ is deallocated, and the enclosure constraints that were set upon $e$ by the corresponding flow route are removed. After sending the deallocation success message, the operation is concluded at the local component.

**Reporting Deallocation Success**

The success of a deallocation operation at a flow route or a branch flow route is indicated by a deallocation success message, which is sent as a response to a corresponding deallocation request message $m$ along a local connector $c$ by calling the procedure `SendBackDeallocationSuccessMessage` that is defined in the following.

```plaintext
procedure SendBackDeallocationSuccessMessage(e, c, m)
    m' ← CreateMessage(DeallocationSuccess)
    m'.FlowRoute ← m.FlowRoute
    m'.BranchFlowRoute ← m.BranchFlowRoute
    SendMessage(c, m')
end procedure
```

Within this new message, the flow route and branch flow route of the original request message are included verbatim.

**Finishing Operations**

The definition of the procedure `HandleMessage` reveals a fundamental difference between the messaging scheme of the flow path allocation algorithm with respect to the flow path analysis and flow path monitoring algorithms that have been presented in this chapter, namely, that response messages are not immediately processed by the component that receives them, but are rather stored as part of the communication state of the component. The reason for this is that in the general case, a component must wait for all responses to arrive before being able to emit a response, and this is accomplished by the flow path allocation algorithm by first storing all responses that have been received, and then analysing all these responses in a single operation. This analysis is performed by the procedure `FinishOperation` that is defined in the following. As we have seen, this procedure is invoked by the procedure `HandleMessage` after having received, and properly stored, a response message.

```plaintext
procedure FinishOperation(e)
    if Children = Ø then
        if Operation ∈ {Allocate, AllocateBranch} then
            FinishAllocateOperation(e)
        else if Operation ∈ {Deallocate, DeallocateBranch} then
            FinishDeallocateOperation(e)
        end if
        EndOperation()
    end if
end if
```
The procedure `FinishOperation` first determines if there are any child connectors still waiting for a response message. If this is the case, then the procedure has no effect. Otherwise, the type of operation that is currently active at the local component is classified as either an allocation operation or a deallocation operation, and the procedure `FinishAllocateOperation` or `FinishDeallocateOperation` is called accordingly. After doing this, the procedure ends the current operation at the local component.

An allocation operation is concluded by the procedure `FinishAllocateOperation` that is defined as follows:

```plaintext
procedure FinishAllocateOperation(e)
    \( t \leftarrow \text{AllocationSuccess} \)
    if Enclose \( \neq \emptyset \) then
        \( S \leftarrow \{(e,\text{Setting})\} \)
    else
        \( S \leftarrow \emptyset \)
    end if
    \( S' \leftarrow \emptyset \)
    for \( m \in \text{Response} \) do
        if \( m.\text{MessageType} = \text{AllocationSuccess} \) then
            \( S \leftarrow S \cup m.\text{Constraints} \)
        end if
        if \( m.\text{MessageType} = \text{Conflict} \) then
            \( t \leftarrow \text{Conflict} \)
            \( S' \leftarrow S' \cup m.\text{ConflictFlowRoutes} \)
        end if
    end for
    if Parent = 0 then
        if \( t = \text{AllocationSuccess} \) then
            ReportAllocation(FlowRoute, S)
        end if
        if \( t = \text{Conflict} \) then
            ReportConflict(FlowRoute, S')
        end if
    else
        \( m' \leftarrow \text{CreateMessage}(t) \)
        \( m'.\text{FlowRoute} \leftarrow \text{FlowRoute} \)
        \( m'.\text{BranchFlowRoute} \leftarrow \text{BranchFlowRoute} \)
        if \( t = \text{AllocationSuccess} \) then
            \( m'.\text{Constraints} \leftarrow S \)
        end if
        if \( t = \text{Conflict} \) then
            \( m'.\text{ConflictFlowRoutes} \leftarrow S' \)
        end if
        SendMessage(Parent, m')
    end if
end procedure
```
The variable $t$ of this procedure stores the type of response to send, and it is initially assumed to be an allocation success response. The variable $S$ stores the set of constraint pairs to send as part of the message, and it initially contains the constraint pair that corresponds to the local element $e$ if this element has received enclosure constraints; otherwise, this set is initially empty. Also, the variable $S'$ stores the set of conflicting flow routes that have been detected, and is initialised as an empty set. The procedure processes every incoming response message $m$ that has been received as follows: if the message is an allocation success response, then its set of constraint pairs is adjoined to the set $S$; if the message is an allocation conflict response, then the type $t$ of response changes to indicate that a conflict has occurred, and the set of conflicting flow routes of the message $m$ is adjoined to the set $S'$. After this processing, $t$ indicates an allocation conflict if at least one such conflict was received as a response along a child connector, $S$ holds every constraint pair that was received in an allocation success response along a child connector, and $S'$ contains every flow route that has caused a conflict with the flow route that is currently being allocated, and which was received in an allocation conflict response along a child connector of the local component. With this information, the procedure is now in a position to report the result of the allocation operation. If no parent connector has been registered, then it means that the local element $e$ is the initial element of the currently allocated flow route, and the result is reported directly to the local component. If a parent connector is given, then a response message is created with the type $t$ that was determined earlier, and the message is sent along the parent connector, that is, towards the initial element of the flow route or branch flow route. The flow route and branch flow route that were stored as part of the communication state of the component are sent with this message, and depending on the type of response that is being issued, a set of constraint pairs or a set of conflicting flow routes is added to the message as well.

The procedure FinishDeallocateOperation works in a similar manner as the previous procedure but for the purpose of deallocating flow routes, and is defined as follows:

\[
\text{procedure FinishDeallocateOperation}(e) \\
\quad t \leftarrow \text{DeallocationSuccess} \\
\quad S \leftarrow \emptyset \\
\quad \text{for } m \in \text{Response} \text{ do} \\
\quad\quad \text{if } m.\text{MessageType} = \text{Conflict} \text{ then} \\
\quad\quad\quad t \leftarrow \text{Conflict} \\
\quad\quad\quad S \leftarrow S \cup m.\text{ConflictFlowRoutes} \\
\quad\quad \text{end if} \\
\quad\text{end for} \\
\quad \text{if } t = \text{DeallocationSuccess} \text{ then} \\
\quad\quad \text{Deallocate}(e,\text{FlowRoute}) \\
\quad\quad \text{RemoveEnclosureConstraint}(e,\text{FlowRoute}) \\
\quad\text{end if} \\
\quad \text{if } \text{Parent} = 0 \text{ then} \\
\quad\quad \text{if } t = \text{DeallocationSuccess} \text{ then} \\
\quad\quad\quad \text{ReportDeallocation}(\text{FlowRoute}) \\
\quad\quad\text{end if} \\
\quad\quad \text{if } t = \text{Conflict} \text{ then} \\
\quad\quad\quad \text{ReportConflict}(\text{FlowRoute}, S) \\
\quad\quad \text{end if} \\
\quad\text{else} \\
\quad\text{end if} \\
\text{end if}
\]
5.2 Decentralised Algorithms

\[ m' \leftarrow \text{CreateMessage}(t) \]
\[ m'.\text{FlowRoute} \leftarrow \text{FlowRoute} \]
\[ m'.\text{BranchFlowRoute} \leftarrow \text{BranchFlowRoute} \]
\[ \text{if } t = \text{Conflict} \text{ then} \]
\[ m'.\text{ConflictFlowRoutes} \leftarrow S \]
\[ \text{end if} \]
\[ \text{SendMessage(Parent, } m') \]
\[ \text{end if} \]
\text{end procedure} \]

This time, the response type \( t \) is initialised to the message type for deallocation success, and the set \( S \) contains the set of conflicting flow routes that will be reported. All received response messages are processed, and if a deallocation conflict is detected, then \( t \) is changed accordingly and the set \( S \) is augmented with the set of conflicting flow routes found in the corresponding message. After this, a situation of deallocation success is handled by deallocating the local element \( e \) for the flow route stored as part of the current operation, and removing any enclosure constraint added to \( e \) for this flow route. Finally, the result of the deallocation operation is reported, either locally if no parent connector has been registered, or by means of a response message if a parent connector is present. In the case that a deallocation conflict was detected, then the set \( S \) of conflicting flow routes is reported as well.

**Similar Algorithms**

The task of allocating plant elements for their use by a product flow path must solve a resource allocation problem, where several active participants – in this case product flow paths – compete for resources – in this case plant elements – in order to achieve their goals. The following are classical problems of process synchronisation [56] which are similar to the problem setting of product flow path allocation, and which have inspired the decentralised algorithm that was presented in the present section.

- **Dining philosophers problem** [13, 56]: The dining philosophers problem represents a general description of a decentralised resource allocation problem. Several philosophers (usually 5) sit at a round table thinking and eating. A fork is placed at the table between every two philosophers that sit next to each other. The philosophers may think for some time, and eventually get hungry and try to eat. In order to eat, a philosopher must pick up both forks on each of his or her side, one after the other. If one of these forks is currently being used by the neighbouring philosopher, then the philosopher must wait until the fork is free. Once a philosopher starts eating, he or she eats for some time and eventually releases both forks and begins to think again.

  In this problem, philosophers correspond to active processes that compete for shared resources in the form of forks. Processes may either be idle (as when a philosopher thinks), activating (as when a philosopher tries to pick up the forks) or active (as when a philosopher eats). Analogously, a product flow path may be idle, in the process of allocating its corresponding plant elements, or successfully allocated and ready to be used. Also, conflicts may arise between philosophers that try to use the same forks, just as product flow paths may try to allocate common plant elements and enter a conflict.
Several solutions have been given for the dining philosophers problem [7, 8, 13], all of which must guarantee *mutual exclusion* (that no fork is ever used by two philosophers at once), *avoidance of deadlocks* (that no two philosophers wait simultaneously for each other to release a fork) and *progress, fairness or avoidance of starvation* (that no philosopher is left waiting forever). The product flow path allocation algorithm that has been presented in this section guarantees the property of mutual exclusion because no element may ever be allocated by two distinct flow routes simultaneously. Also, the components respond as soon as a conflict is detected, thereby causing the allocation of the entire flow route to eventually fail when a single conflict occurs, and this allows the decentralised algorithm to avoid deadlocks from occurring. In other words, the allocation of a flow route $r$ never waits for a component to deallocate its corresponding plant element from another flow route $r'$, but rather fails immediately if this additional allocation is detected. This is similar to a philosopher that releases a fork after confirming that the second fork that he or she needs is currently occupied, thus going back to a thinking state. Finally, the property of fairness is not relevant to the decentralised flow path allocation algorithm, because there must exist a decision maker at a higher level that initiates the operations of allocation and deallocation of product flow paths, and that is capable of retrying an operation after a conflict has been detected.

• **Readers-writers problem** [10, 56]: The readers-writers problem describes a very common situation in today’s computer systems, where a data object such as a file is shared by several active participants which are classified as either readers or writers. The members of the former group only intend to read the data in the object, whereas the members of the latter group intend to modify this data. Naturally, multiple readers may use the shared resource simultaneously, but a writer must have exclusive access to it. Any solution to the readers-writers problem must satisfy this restriction.

We may draw an analogy between the readers-writers problem and product flow path allocation as follows: a product flow path requires exclusive access to the plant elements that it contains, and may therefore be seen as a “writer” of these elements. Furthermore, a product flow path may enforce a flow allowance setting constraint to a given plant element at a neighbouring branch flow route, in order to avoid product leaks and product mixtures. Since a single element may accept compatible flow allowance setting constraints from different product flow paths simultaneously, each of these flow paths may be seen as a “reader” to this element. However, a difference between both problems is that an element may be allocated by a flow route $r$ and constrained by another flow route $r'$ at the same time, while a data object may never be written by a process and be read by another simultaneously.

The classical solution to the readers-writers problem [56] uses a synchronisation variable called a *semaphore* that is used to guarantee the mutual exclusion of the writers that access the shared data object. Additionally, a counter for storing the number of readers currently accessing the resource is kept. Using this counter, a reader that begins to read or that finishes reading when no other readers are accessing the object may detect this situation and use the semaphore as a writer does, effectively blocking writers when reading is taking place, while allowing additional readers to access the object. The manipulation of the global variables $Alloc$ and $Enclose$ at every component mimic this behaviour: $Alloc$ is used like a semaphore that indicates if the plant
5.2 Decentralised Algorithms

![Diagram of a decentralised component-based system]

Figure 5.3: An example decentralised component-based system.

...element is allocated by a flow route, and keeps other flow routes from allocating the element; also, Enclose is used like a counter that keeps track of which (and how many) flow routes have added enclosure constraints to the local plant element. In this manner, the allocation of a plant element is accomplished in a similar way to how the access of a set of processes is managed for a shared data object.

5.2.6 Examples of the Algorithms

In order to give an intuition of how the decentralised algorithms that have been presented in this chapter operate, this section presents an example which indicates the messages that are involved during the execution of the algorithms on a simple decentralised component-based system. This system is depicted in Figure 5.3, and consists of a few components labelled from A to K that are interconnected with each other as shown. In the following, the execution of each of the decentralised algorithms on this system will be exemplified.

Flow Path Analysis Example

Consider the discovery of flow routes that start at component A and end at component K. The execution of the algorithm begins by component A sending a Search message to component B with the partial flow route that contains only A. In turn, component B appends itself to the partial flow route and sends a Search message to components F and C, thus initiating parallel searches along these two branches.

When component F receives this message, it also appends itself to the partial flow route and sends a Search message to component H, and this is also done by components H and I. When component J receives the Search message, it also initiates parallel searches along component K and component G. When component K receives the Search message, it detects the discovery of the flow route ABFHIJK and issues a SearchResult message back to component J. This message is then sent back along the flow route and reaches component A, which is then able to report the first of the flow routes found by the operation. Returning to component G, it sends a Search message to component D which again initiates two parallel searches along components E and C. Component E has no other connection where to send a Search message, and therefore sends no further messages. In turn, component C forwards the Search message to component B, which is then able to detect a cycle with respect to...
the flow route ABFHIJGDC, and therefore discards the message without sending any further messages.

Returning to the second parallel search started along component C, this component sends a Search message to component D, which then forwards this message to components E and G. Component E discards the message due to lack of further connections, and component G sends a Search message to component J. This time, component J sends a Search message to components I and K, where component K is again able to detect a positive discovery and sends back a SearchResult message for the flow route ABCDGJK. In turn, component I forwards the Search message to component H, which does the same to component F. Component F also forwards this message to component B, which detects the cycle with respect to the flow route ABCDGJIHF, and discards the request message without sending any further messages.

In this manner, the flow routes ABFHIJK and ABCDGJK are found by the algorithm, and reported at component A. This example was done under the assumption that the flow allowance model of the plant permits the flow of products along all product connections; if this were not the case, then some of the messages mentioned here would have been inhibited, and the results of the flow route search would have been affected as a consequence of this.

**Flow Path Monitoring Example**

For this example, we will consider the monitoring of the flow route ABFHIJK that was detected by the flow path analysis algorithm in the previous section. Furthermore, we will consider that component E is configured as a product source and sink.

Component A begins by sending a Monitor message to component B. When component B receives this message, it detects the branching structure of the plant and sends two messages: a Monitor message to component F, and a MonitorBranch message to component C. The Monitor message is then sent from component F to component H, then to component I, and then to component J. Component J is also able to detect a branching structure, and sends two messages: a Monitor message to component K, and a MonitorBranch message to component G. Component K detects that it is the last element in the flow route, and sends a MonitorResult message back along the flow route, which contains the value of the open condition that was computed by the components along the route. When this message reaches component A, it is able to report the open status of the flow route.

We will now consider the message sent along the branch flow routes in this example. Component C had received a MonitorBranch message, and forwards this to component D. In turn, component D forwards this message to components E and G. Because component E is a product source and sink, it issues leak and mixture alerts for the branch flow route BCDE and sends a Leak and a Mixture message that contain this branch flow route backwards along the branch and then along the flow route. When these messages reach component A, the corresponding alerts are reported. When component G receives the MonitorBranch message, it sends it to component J which is able to detect a cycle with respect to the flow route that is being monitored, and therefore discards this message without issuing any further messages.

Finally, upon receiving the MonitorBranch message, component G forwards it to component D, which again forwards the message to components C and E. Once again, component E detects the leak and mixture conditions and issues Leak and Mixture messages for
the branch flow route JGDE. These messages are sent back along the branch flow route and
flow route and finally reach component A, which then reports the corresponding alerts. Com-
ponent C forwards the MonitorBranch message to component B after receiving it, and
due to the cycle that is detected by component B with respect to the flow route, the message
is discarded.

In this manner, the flow route is monitored and the potential leak and mixture situations
are detected and reported. As in the case of the flow path analysis algorithm, it has been
assumed that the flow allowance model and state of the plant permits the flow of products
along all product connections, and any exception to this assumption would cause the inhibi-
tion of some of the messages that have been presented. For instance, if component C does
not allow product flow through it at the current plant state, then the leak and mixture alerts for
the branch flow route BCDE would not be issued. The same applies for the determination of
the open condition of the flow route.

Flow Path Allocation Example

For this final example, we will consider the allocation of the flow route ABFHIJK. This time,
we will also consider that components C and G have flow allowance models that contain
two flow allowance settings: closed, where no flow is allowed through the component in any
direction, and open, where flow is allowed through the component in all directions. All other
components are assumed to have a single flow allowance setting that permits flow through
the component in all directions.

The allocation of the flow route begins with the allocation of component A by the flow route
and the sending of an Allocate message from component A to component B. In turn, com-
ponent B allocates itself and sends two messages: an Allocate message to component
F and an AllocateBranch message to component C. Component F then allocates itself
and forwards the Allocate message to component H, which does the same, after which
component I also does the same. When component J receives the Allocate message,
it allocates itself and sends an Allocate message to component K and an Allocate-
Branch message to component G. Component K then allocates itself and, being the last
element in the flow route, sends an AllocationSuccess message back to component J
because it does not need to wait for any other incoming messages.

When component C receives the AllocateBranch message, it tries to find a flow al-
lowance setting that inhibits flow through the component, and selects the closed setting for
this. It then adds an enclosure constraint for this setting and for the given flow route, and
after doing this, it sends an AllocationSuccess message back to component B which
includes the enclosure constraint that was added. The same happens in the case of compo-
nent G, which again selects its closed setting, adds an enclosure constraint, and returns an
AllocationSuccess message to component J.

After component J has received the AllocationSuccess messages from components
K and G, it then issues an AllocationSuccess message back to component I. This com-
ponent was only waiting for this message, and therefore sends an AllocationSuccess
message back to component H. The same happens with components H and F, and when
component B has received both AllocationSuccess messages from components C and
F, it issues an AllocationSuccess message to component A, which is then able to report
the allocation of the flow route together with the enclosure constraints that were added as
part of the allocation operation.
The deallocation of this flow route occurs in a similar fashion, where Deallocate and DeallocateSuccess messages are issued by the components, and where the allocation and enclosure constraints of the components are released. If another flow route, say KJGDE, were allocated while this allocation process was executed, then the component J would immediately issue a Conflict message upon receiving the Allocate message from component I, and which includes the conflicting flow route as part of the message. This would cause the allocation to fail and the conflict to be reported at component A when the Conflict message is delivered to this component from component B.

5.2.7 Termination Properties

An important aspect of the execution of an algorithm is the assurance of its termination, that is, if it may be guaranteed that every run of the algorithm on some valid input eventually terminates and yields a correct output. In the case of decentralised algorithms such as the ones presented in this chapter, the interactions between the decentralised components can be complex and the assertion of termination is usually not trivial. Because of this, the flow path management algorithms presented in this chapter have been designed with the goal of having simple termination properties.

During an execution of any one of the algorithms, messages are sent across the decentralised system in the form of two message sequences: a request message sequence that travels in a forward direction, and a result message sequence that moves in the opposite direction. The request message sequence may operate in an exploratory fashion – when the components to visit next are not yet known – or in a routed fashion – when the complete trajectory of the messages is known in advance. In turn, the result message sequence always operates in a routed fashion. In the case of the exploratory operation, the messages always contain information about which elements have been already visited by the corresponding message sequence, and the decentralised algorithms are therefore able to detect and avoid cycles in the messaging trajectory. Because messages move back and forth along sequential and finite trajectories, it may be stated that any message trace that corresponds to the execution of any one of the decentralised algorithms of this chapter is finite. Therefore, in order to assert the termination of these algorithms, it suffices to demonstrate the progress of the system, that is, that no message sequence is paused indefinitely. The use of reliable communication channels guarantees that progress is not affected by the communication infrastructure. Also, the use of a fair scheduling scheme guarantees that there cannot occur any starvation of the decentralised components which may affect the progress of the system. Therefore, the only violation to the progress of the system could be caused by a component that fails to send a message that it should have sent as part of a message sequence. Consequently, the termination of these decentralised algorithms may be asserted by demonstrating that every component eventually sends all messages that it should send.

Termination of Flow Path Analysis

As mentioned earlier in this chapter, the flow path analysis algorithm is stateless, which means that every decentralised component reacts to each input message regardless of any additional input messages that were previously received by the component. By examining the way in which incoming messages are handled by the components that are executing this
algorithm, it may be confirmed that every incoming message causes the component to emit
a corresponding outgoing message, or no message at all if the incoming message is to be
discarded due to a given situation. Upon receiving a Search message, a component either
forwards the message to the next components, discards the message if a cycle is detected,
or sends back an SearchResult message if the end location of the flow route specification
is reached. Also, when a SearchResult is received, the message is sent back towards
the start of the flow route for as long as the initial location of the flow route has not been
reached; otherwise, the flow route is reported. Therefore, it is easy to see that every flow
route that satisfies the given flow route specification is eventually found and reported, and
that the algorithm always terminates.

Termination of Flow Path Monitoring

Just like the flow path analysis algorithm, the flow path monitoring algorithm is stateless and
its execution always terminates because of similar reasons. When a component receives
a Monitor message, it either forwards the message to the next component, or returns a
MonitorResult message if the end location of the flow route has been reached. Also, the
MonitorResult messages are all sent back towards the start of the flow route. Thus, the
monitoring of any flow route eventually causes a monitoring result to be reported at the initial
location of the flow route. Similarly, the MonitorBranch messages that are sent along
branch flow routes are either forwarded by the components, discarded if a cycle is found
(with respect to the flow route or the branch flow route), or returned in the form of a Leak
or Mixture message if a source or sink element is reached, and these messages are also
sent back towards the initial element of the flow route. Therefore, we may also conclude that
any leak and mixture situation is identified and ultimately reported by the flow path monitoring
algorithm.

Termination of Flow Path Allocation

Because of the stateful nature of the flow path allocation algorithm, the analysis of its termi-
nation properties requires us to follow a different approach than the one used for the first two
algorithms. Although the messages sent during the execution of this algorithm also advance
first in a request message sequence and then in a result message sequence, the components
enter a waiting state after issuing request messages, and do not send a result message until
all results that are expected by the component have been received. Therefore, the progress
of the system depends on the property that no component is ever kept waiting indefinitely for
a result message.

In order to verify that it is indeed the case that no component is ever kept waiting indefinitely,
it helps to observe that the sending of messages across the decentralised system forms a
tree structure, where the root of the tree is the initial element of the flow route that is
being allocated or deallocated, and where the paths of the tree are formed by the flow route
together with the associated branch flow routes. For every component that is active within
the operation of the algorithm, the Parent variable holds the identifier of the element which
leads to the root of the tree, and the Children variable holds the identifiers of the elements
which lead to the leaves of the tree that are dominated by the component’s element. Since
a component only waits for its child components to send back result messages, and since
the tree has a finite size, it may be demonstrated by induction that no component ever waits
indefinitely for a result message: the leaves of the tree do not wait for any other component, and a non-leaf component does not wait indefinitely if and only if its children do not wait indefinitely either. In this manner, deadlocks are avoided by eliminating any form of cyclic dependencies in the communication structure.

The case of concurrent allocation and deallocation of different flow routes in the same system is also handled adequately by the flow path allocation algorithm. The concurrent execution of these operations corresponds to a form of race condition where components are allocated on a first-come, first-serve fashion. This means for any two operations that are in conflict with each other, one or both of them could fail if they are executed concurrently. Nevertheless, they would never cause the system to deadlock, because any two components that are allocated for different flow routes are also contained in different communication trees, and will never wait for each other’s messages. Thus, every component involved in each of the operations would either perform a successful allocation, or would detect a conflict report it, allowing the algorithm to terminate.

5.3 Model-based Synthesis

One of the advantages of the decentralised component model architecture that has been adopted in this work is that it enables a simple way to automatically construct a component-based system for a given plant, based on the abstract model of this plant. This synthesis approach is therefore regarded as model-based, and can be compared to similar approaches like domain-driven design [16], model-driven architectures [37] or the use of domain-specific languages [58]. In all of these approaches, a model that describes an object is created using concepts which belong to the object’s domain, and this model is used to derive a system for this object, either manually, automatically, or using a combination of both. The main advantage of such techniques is that the model may be created by experts from the application domain, without requiring any knowledge about the desired system. Also, errors are easier to detect in the model than in the corresponding system.

The approach that this work presents takes advantage of the fact that plants are usually easier to understand and describe than their corresponding control systems. Plants are well understood by the plant engineers and chemical engineers that design them, by the process control engineers that design the corresponding control systems, and by plant operators which work with the plant on a daily basis. This common understanding may be used to create and verify abstract plant models in a cooperative manner. Once a model is obtained, a product flow path management system may be constructed from it in an automatic manner based on this model, thereby saving a significant amount of effort, and cost. Furthermore, errors due to incorrect construction and parametrisation of the many components of the system are avoided by following this algorithmic synthesis approach.

5.3.1 Synthesis Environment

The algorithm that performs the synthesis of a product flow path management system requires an operation environment that provides two basic services:

1. access to the abstract plant model and flow allowance model of a plant, and
2. the ability to manipulate the constituents of a decentralised component-based system.
5.3 Model-based Synthesis

The synthesis technique has no further requirements apart from these, and may be implemented in a variety of ways. In order to formulate the synthesis algorithm, we assume that the first requirement is fulfilled by passing this information to the algorithm as an argument, but an actual implementation may need to retrieve this information from a database or a file. For fulfilling the second requirement, we assume the availability of the following predefined procedures that permit the manipulation of the components and communication links of the decentralised system:

- **CreateComponent** \( (e, t, C_e, F) \): Creates a new component for the plant element \( e \), with the element type \( t \), the set of local connectors \( C_e \) and the flow allowance model \( F \).

- **GetComponent** \( (e) \): Retrieves the component that was created by the procedure CreateComponent for the plant element \( e \).

- **CreateCommunicationLink** \( (o_1, c_1, o_2, c_2) \): Creates a new directed communication link from the component \( o_1 \) to the component \( o_2 \). On the side of the component \( o_1 \), the link is associated to the plant connector \( c_1 \), and on the side of the component \( o_2 \), the link is associated to the plant connector \( c_2 \). After this communication link is created, the following assertions hold:
  - If the component \( o_1 \) calls SendMessage \( (c_1, m) \), then the message \( m \) is sent over this link and reaches the component \( o_2 \).
  - If the component \( o_2 \) calls GetConnectedElement \( (c_2) \), the result is the local element of the component \( o_1 \).

The fulfilment of these properties only requires a flow of information from \( o_1 \) to \( o_2 \), and therefore, these communication links can be implemented using a medium that offers unidirectional communication. For this, the component \( o_1 \) can send its own identifier through the communication link from time to time, in order to identify itself and allow the procedure GetConnectedElement at \( o_2 \) to offer its result without the need to send a message to \( o_1 \).

5.3.2 Component Synthesis

Given an adequate operation environment that adheres to the requirements that were presented in the previous section, an algorithm may perform the construction of a component-based system for a given plant. We may formulate this algorithm by means of the procedure SynthesiseComponents, defined in the following, that receives an abstract plant model and a flow allowance model of the plant as arguments.

**procedure** SynthesiseComponents \( (\Delta T, E, C, \tau, \epsilon, \circ, \Phi) \)

**for** \( e \in E \) **do**

CreateComponent \( (e, \tau(e), C_e, \Phi(e)) \)

**end for**

**for** \( (c_1, c_2) \in \circ \) **do**

\( o_1 \leftarrow \text{GetComponent}(\epsilon(c_1)) \)

\( o_2 \leftarrow \text{GetComponent}(\epsilon(c_2)) \)

CreateCommunicationLink \( (o_1, c_1, o_2, c_2) \)

CreateCommunicationLink \( (o_2, c_2, o_1, c_1) \)

117
end for
end procedure

The synthesis algorithm first iterates over all plant elements found in the abstract plant model, and creates a corresponding component for each. The type of the element is provided, as well as the set of connectors of the element and the flow allowance model of the element. In this manner, the component receives all the information that it needs for executing the decentralised algorithms that have been presented in this chapter. Additionally, components disregard all the details of the plant that do not concern its local element, and in this way, the knowledge about the plant is distributed across the decentralised system.

After creating all components, the synthesis algorithm processes all pairs of connectors \((c_1, c_2)\) that belong to the connection relation \(\circ\). For every such pair, the components that correspond to the respective elements of both connectors are retrieved, and two communication links are established between these components, one in each direction. In this way, by means of two unidirectional communication links, a bidirectional communication channel is created between both components.

After having created all components and all communication links of the decentralised system, the synthesis algorithm has concluded its task. In practice, additional steps may be required to complete the synthesis operation, such as adding or enabling the algorithms at every component. Also, the actual physical location of a component within a decentralised control system must be specified or automatically decided. As these details are implementation-specific, they are not treated in this presentation, but are nonetheless considered important for an actual deployment of a decentralised component-based system.

### 5.4 Computational Complexity

We now consider the computational complexity of the algorithms that have been presented in this chapter. The complexity of an algorithm is a measure of the amount of resources that the algorithm needs to fulfil its task, under consideration of the size of the task, that is, the size of the input [1]. In the present analysis, we consider the following resources: execution time, required memory (space), and in the case of decentralised algorithms, the number of messages that are sent. This assessment is done in a theoretical way, and helps us to estimate the dimensions of the resources that are needed by a decentralised flow path management system. Table 5.1 at the end of this chapter gives a summary of the computational complexity of the algorithms that have been presented.

The big \(O\) notation is used to denote an upper bound on the growth rate of the demand for the respective resource, and is determined by considering the worst case scenario that the algorithm may face. A general assumption that is made in this analysis is that the sets \(C_e\) and \(\Phi(e)\) have sizes that are constants in the range \([1, n]\), where \(n\) is a small integer. Experience with the abstract plant model has shown that, for instance, a given plant element rarely has more than 3 or 4 connectors, and more than 2 or 3 flow allowance settings. Therefore, it is expected that this assumption, which permits the simplification of the complexity analysis, is in accordance with the majority of application scenarios.

The complexity of the decentralised algorithms is analysed in two ways: component-wise and system-wise. The former estimates the greatest amount of resources that a single component may need for a single invocation, say the servicing of a local event or the handling of
an incoming message. The latter assesses the total amount of resources that are required system-wide to perform a given task, and depend on the component-wise complexity.

5.4.1 Complexity of Flow Path Analysis

We begin by examining the component-wise complexity of the flow path analysis algorithm. The initiation of a search at a component requires time which is $O(|C_c| \cdot |\Phi(e)|) \approx O(1)$, that is, constant time according to assumptions of a small number of connectors and flow allowance settings per element. Handling a search message requires time $O(|\Phi(e)| + |E| + |C_c| \cdot |\Phi(e)|) \approx O(|E|)$, that is, linear with respect to the number of elements in the plant. Furthermore, handling a search result message requires time $O(|r| + |C_c|) \approx O(|r|)$, that is, linear with respect to the length of the longest flow route that can be found. For all these operations, the space complexity per component is constant ($O(1)$), because the procedures use a constant amount of local storage space. Thus, in general a component that executes this algorithm requires time $O(|E|)$ and space $O(1)$ to execute a single invocation in the worst case.

At the system level, we may calculate the number of messages that are needed to complete a flow path analysis operation for a flow route specification $(c, e')$. For a flow route search, the number of search messages sent is

$$O \left( f^0 + f^1 + \ldots + f^{|E|-1} \right) = O \left( \sum_{i=0}^{|E|-1} f^i \right) = O \left( \frac{1 - f^{|E|}}{1 - f} \right) \approx O \left( f^{|E|} \right),$$

where $f = |C_c| - 1$. This is because each search message that arrives at a component is forwarded to at most $|C_c| - 1$ neighbours. In this setting, $f$ is called the branching factor of the search [50], and this branching behaviour causes the number of messages to grow exponentially with respect to the number of elements in the plant. However, it must be noted here that since $|C_c|$ is usually a small number and in most of the cases equal to 2, we can expect $f$ to be close to 1 on average. Thus, the exponential behaviour may be lessened on a real plant, and could even approximate a polynomial behaviour depending on the plant’s topology.

Reporting every flow route search result requires a number of messages $O \left( f^{|E|}-1 \cdot (|r| - 1) \right) \approx O \left( |r| \cdot f^{|E|} \right)$ because there can be at most $f^{|E|}-1$ flow routes that connect two distinct elements in the plant, and each of these flow routes requires at most $|r| - 1$ messages to be reported to the component that corresponds to its initial element. Therefore, the total number of messages required by a product flow path analysis is $O \left( f^{|E|} + |r| \cdot f^{|E|} \right) \approx O \left( |r| \cdot f^{|E|} \right).

The time complexity of the entire, system-wide operation may now be determined as follows. The initiation of the search occurs only once, hence requires $O(1)$ time. The handling of all search messages requires time $O \left( |E| \cdot f^{|E|} \right)$, because there are a total of $f^{|E|}$ messages that each requires $|E|$ time to be handled. In the same manner, we may determine that the time complexity of the handling of result messages is $O \left( |r|^2 \cdot f^{|E|} \right)$, where $|r|$ is the length of the longest flow route that can be found. Therefore, the time required by the system to perform a complete flow path analysis operation is

$$O \left( 1 + |E| \cdot f^{|E|} + |r|^2 \cdot f^{|E|} \right) \approx O \left( \left( |E| + |r|^2 \right) \cdot f^{|E|} \right)$$
which means that the flow path analysis time grows exponentially with the number of elements in the plant, and linearly with respect to the length of the longest flow route that it may find. As mentioned, the fact that \( f \) is expected to be close to 1 on average indicates that the algorithm is not necessarily impractical in spite of its exponential nature.

The space complexity of entire flow path analysis operation may be determined as \( O(1 \cdot |E|) \approx O(|E|) \), because every component requires constant space to operate but there are exactly \(|E|\) components in the system.

### 5.4.2 Complexity of Flow Path Monitoring

At the level of the individual components, the time complexity of the flow path monitoring algorithm may be estimated as follows. Initiating a monitoring operation for a flow route requires \( O(|C_r| + |C_e|) \approx O(1) \) time, handling a monitor message requires time \( O(|r| + |C_e| + |C_e|) \approx O(|r|) \), handling a monitor result message requires \( O(|r| + |C_e|) \approx O(|r|) \) time, handling a monitor branch message requires \( O(|r| + |b| + |C_e|) \approx O(|r| + |b|) \) time where \(|b|\) is the length of the longest branch flow route that can be monitored, and handling an alert message requires \( O(|r| + |b| + |C_e|) \approx O(|r| + |b|) \) time. Therefore, a component requires at most \( O(|r| + |b|) \) time to conclude a single operation. The space requirement of every component, as in the case of the flow path analysis algorithm, is constant \( O(1) \).

System-wide, we may estimate the number of messages that are needed in order to monitor a flow route as follows. For a flow route \( r \), exactly \(|r| - 1\) monitor messages and an equal number of monitor result messages must be sent, which gives us a message complexity of \( O(|r|) \) for each of these message types. In the case of branch monitoring messages, the number of messages that are sent from a single branching point, that is from a single component along the flow route \( r \), is \( O(f^0 + f^1 + \ldots + f^{[b]-1}) \approx O(f^{[b]}) \) because these messages are sent in a similar manner to the flow route search messages from the flow path analysis algorithm. In this case, the length of the longest branch flow route \( b \) that can be monitored gives the limit of the succession of messages. Consequently, the total number of branch monitor messages sent is \( O(|r| \cdot f^{[b]}) \).

The number of alert messages that are sent as a result of the branch monitoring from a single branching point may be calculated from the number of possible alert origins \( f^{[b]} \), the length of the longest branch flow route \( |b| \), and the length of the longest flow route \(|r|\), yielding a maximal message requirement of \( O(f^{[b]} \cdot (|b| + |r|)) \). Therefore, the total number of alert messages needed for monitoring all branches of a flow route \( r \) is \( O(|r| \cdot (|r| + |b|) \cdot f^{[b]}) \).

Putting the monitor, monitor result, branch monitor and alert messages together, the total number of messages that are needed by the decentralised system to monitor a flow route \( r \) is

\[
O(|r| + |r| + |r| \cdot f^{[b]} + |r| \cdot (|r| + |b|) \cdot f^{[b]}) \approx O(|r| \cdot (|r| + |b|) \cdot f^{[b]})
\]

The system-wide time complexity of the product flow path monitoring algorithm is now presented. As in the case of the previous algorithm, initiating the decentralised operation requires constant time. Handling all monitor request messages requires \( O(|r| \cdot |r|) \approx O(|r|^2) \) time, handling all monitor result messages requires \( O(|r| \cdot |r|) \approx O(|r|^2) \) time, handling all branch monitor messages requires \( O(|r| \cdot f^{[b]} \cdot (|r| + |b|)) \) time, and handling all alert
messages requires \( O \left( |r| \cdot (|r| + |b|) \cdot f^{[b]} \cdot (|r| + |b|) \right) \approx O \left( |r| \cdot (|r| + |b|)^2 \cdot f^{[b]} \right) \) time. Therefore, the total time required by the decentralised algorithm is

\[
O \left( 1 + |r|^2 + |r|^2 + |r| \cdot f^{[b]} \cdot (|r| + |b|) + |r| \cdot (|r| + |b|)^2 \cdot f^{[b]} \right)
\]

\[
\approx O \left( |r| \cdot (|r| + |b|)^2 \cdot f^{[b]} \right).
\]

Just as in the case of the product flow path analysis algorithm, the space complexity of the entire flow path monitoring operation is \( O \left( 1 \cdot |E| \right) \approx O \left( |E| \right) \).

### 5.4.3 Complexity of Flow Path Allocation

In order to analyse the complexity of the product flow path allocation algorithm, we begin by studying the behaviour of each of the components. Because starting an operation at a component requires \( O \left( |r| + |b| \right) \) time, all of the following operations which make use of this functionality have this same time complexity: initiating an allocation of a flow route, handling an allocate request message, handling an allocate branch request message, initiating a deallocation, handling a deallocate request message, and handling a deallocate branch request message. Also, because finishing an operation requires \( O \left( |r| \right) \) time, then handling any success or conflict response message has this same time complexity. As for the space needed by a component, we may estimate the largest state that a given component may store at any point during the execution of the algorithm as \( O \left( |r| + |r| \cdot |C_r| + |C_r| + |r| + |b| + |C_r| \right) \approx O \left( |r| + |b| \right) \). Here, we estimate the size bound of the local variable Enclose as \( |r| \cdot |C_r| \), because an element receives enclosure constraints via its connectors, and the same connector may not contribute to constraints from two different flow routes.

At the level of the entire system, the number of messages needed by the allocation or deallocation of a flow route \( r \) may be calculated as follows. Because the communication structure of this decentralised algorithm forms a spanning tree [1] of the graph of the plant, the maximum number of allocate or deallocate requests, be it along the flow route or along a branch flow route, coincides with the maximum number of edges of this tree which is \( O \left( |E| - 1 \right) \approx O \left( |E| \right) \). Consequently, the number of responses to these messages is also \( O \left( |E| \right) \). Therefore, the total number of messages needed by an allocation or a deallocation operation is \( O \left( |E| + |E| \right) \approx O \left( |E| \right) \).

The total time needed by a system-wide allocation or deallocation operation may be formulated as \( O \left( |r| + |b| + |E| \cdot (|r| + |b|) + |E| \cdot |r| \right) \approx O \left( |E| \cdot (|r| + |b|) \right) \) by considering the time it takes to initiate the operation, to handle all request messages and to handle all responses. Furthermore, the total space needed by the decentralised system is \( O \left( |E| \cdot (|r| + |b|) \right) \) because as mentioned earlier, each component in the system has a space requirement that is \( O \left( |r| + |b| \right) \).

### 5.4.4 Complexity of Component Synthesis

The last algorithm that was presented in this chapter deals with the synthesis of a decentralised component-based system for a plant based on the plant’s abstract plant model. The time requirement of this simple algorithm is \( O \left( |E| + |o| \right) \), because it iterates over all
Table 5.1: Computational complexity of the algorithms that have been presented in this chapter. \(|E|\) represents the number of elements in the plant, \(|r|\) is the length of the longest flow route that may be found, monitored or allocated, \(|b|\) is the length of the longest branch flow route that may be monitored or allocated, \(f\) is the branching factor of the plant and is given by \(|C_r| - 1\), and \(|\circ|\) is the number of product connections in the plant.

### Individual Components

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow Path Analysis</td>
<td>(O(</td>
<td>E</td>
</tr>
<tr>
<td>Flow Path Monitoring</td>
<td>(O(</td>
<td>r</td>
</tr>
<tr>
<td>Flow Path Allocation</td>
<td>(O(</td>
<td>r</td>
</tr>
</tbody>
</table>

### Decentralised System

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Space</th>
<th>Messaging</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow Path Analysis</td>
<td>(O\left((</td>
<td>E</td>
<td>+</td>
</tr>
<tr>
<td>Flow Path Monitoring</td>
<td>(O\left(</td>
<td>r</td>
<td>\cdot (</td>
</tr>
<tr>
<td>Flow Path Allocation</td>
<td>(O\left(</td>
<td>E</td>
<td>\cdot (</td>
</tr>
</tbody>
</table>

### Synthesis

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component Synthesis</td>
<td>(O(</td>
<td>E</td>
</tr>
</tbody>
</table>
6 Analysis of Resources and Tools

The present work introduces a technology for the decentralised automatic management of product flow paths which follows the design outlined in Chapter 2, which is based on the formal models presented in Chapter 4, and which implements the decentralised algorithms explained in Chapter 5. This technology has been implemented in the form of a prototypical proof of concept that is the topic of Chapter 7. The present chapter gives an overview of the technological resources and tools that made this realisation possible, which consist of programming languages and environments, communication infrastructures and plant modelling technologies.

6.1 Programming Languages

A programming language is an artificial language – as opposed to a natural language used by human beings to communicate with one another – that is used to write computer programs. Practically any piece of software that executes on a computer is written in some programming language. This section outlines the languages that were used in the implementation of the prototypical flow path management system.

6.1.1 C

The C programming language [31] is a language designed for systems programming, which is the development of infrastructure systems such as operating systems, communication systems, device drivers and input/output systems. Originally developed by Dennis Ritchie at AT&T Bell Labs, it was used as the programming language of the popular Unix operating system and has since then become one of the most widespread and well-known programming languages in the world. It has been standardised by ANSI [4] and ISO/IEC [26].

C is a simple procedural and structured language that provides the programmer with basic numerical data types, pointers or typed addresses which support arithmetic operations, data structure definitions, type-safe enumerations, and first-class procedures (pointers to procedures). The C language allows the memory of the system to be used with great freedom, thus supporting the implementation of memory-management tasks and the use of memory-mapped input/output. Because of this, C programmers must be specially careful about the way in which the memory is handled, and errors related to improper memory handling are rather common in C programs.

C is a compiled language, which means that a C compiler is used to translate C programs to machine code, which is then directly executable by a given computer. The relative simplicity and efficiency of this implementation approach, coupled with the simplicity and effectiveness of the language, has enabled the widespread adoption of C for programming a variety of machine types, ranging from embedded micro-controllers to supercomputers.
6.1.2 Tcl

Tcl [62], which stands for “tool command language”, is a popular scripting language, meaning that it is used to write scripts or programs that are executed by an interpreter. The language has a very simple syntax that consists of commands, which are invocations to command objects that include sequences of argument values. Commands represent not only procedure invocations, but also control structures for conditional execution, iteration, and error handling. This makes the language very easy to learn, and is commonly used for rapid prototyping.

The Tcl language is extensible, which means that it is possible to incorporate new features into the language by providing extension packages, which offer new types of commands to the programmer. The best-known extension for Tcl is Tk, which is a toolkit for the construction and use of graphical user interfaces. The pairing of Tcl and Tk usually bears the name of Tcl/Tk, which provides an environment for developing cross-platform graphical applications. Many other extensions for the Tcl language exist, which offer functionalities such as access to operating system services, network connectivity, world wide web services, database access, and the use of multimedia resources.

6.2 ACPLT Technologies

The term ACPLT Technologies is a comprehension of all reference models and implementations that are developed by the Chair of Process Control Engineering at RWTH Aachen University in Aachen, Germany, where the present work has been developed. The aim of the ACPLT technology body is to provide a vendor- and platform-independent realisation of process control solutions that are fuelled by the current research in the field, and to strive for widespread adoption and standardisation.

The prototypical implementation of a product flow path management system that is presented in Chapter 7 is based on the ACPLT technologies that are introduced in this section.

6.2.1 ACPLT/KS Communication System

ACPLT/KS [3] is a communication system that was designed specifically to support the needs of process control systems and applications. The fundamental concept behind ACPLT/KS is meta-modelling, where the fixed and predefined elements of the communication protocol are generic elements that may be used to describe any concrete object model of a process control system: variables, structures, histories, domains (element containers), and links. By using these basic elements, a client/server communication scheme may be established without specifying the actual object model that the server employs to the client; with the use of the basic elements of the protocol’s meta-model, the server is able to describe its own model for the client through ACPLT/KS, thus enabling a flexible form of communicating complex and system-dependent information between clients and servers in a semantically-clear manner.

Using ACPLT/KS, a client connects to a server and requests one or more ACPLT/KS services, which consist of information requests such as “get variable” and “get engineered properties”, as well as model-changing requests such as “set variable”, “create object”, “delete object”, “create link” and “delete link”. Every service request is accompanied by a path, which is a text string that is used to indicate the location of an object within the tree-based structure of the server – similar to the path of a file in a hierarchical file system. The services offered
by a ACPLT/KS server permit the use of this communication system for various different pur-
poses such as online acquisition of plant data and the engineering of control systems.

The ACPLT/KS C++ Communication Library is the reference implementation of an AC-
PLT/KS application programming interface, and it is written in the C++ programming lan-
guage. It offers the client the same object-oriented view of the model offered by the server,
thus simplifying the development of ACPLT/KS client applications. The library additionally
features object classes that aid in the implementation of ACPLT/KS servers. Presently there
exist many applications operating in industrial settings that make use of this communication
library and the ACPLT/KS communication system.

Siemens PCS 7 OCS

The Siemens PCS 7 OCS [54], which stands for “PCS 7 Open Communication Server”, is an
add-on package for the SIMATIC PCS 7 process control system. It consists of an ACPLT/KS
server that allows access to the object model and operation data of a PCS 7 system through
the ACPLT/KS protocol. For instance, the configuration and process variables of the process
control system can be accessed as ACPLT/KS variables, and objects such as function blocks
and alarms can be explored as ACPLT/KS domains. This permits the access of online plant
information from a PCS 7 control system to any application that can act as an ACPLT/KS
client.

6.2.2 ACPLT/OV Object Management System

ACPLT/OV [40] is an object management system for use in process control settings. It con-
sists of a server program (OV server), a programming library (OV-Lib), a language for defining
object-oriented class models (OVM), and a collection of development support utilities. AC-
PLT/OV permits the development of object-oriented applications that can operate in real-time
process control environments.

An ACPLT/OV application consists of an OV server that hosts a collection of software ob-
jects. The server can load class libraries that contain definitions of object classes, and may
instantiate these classes to create objects that can operate as part of the system. The class
libraries are written in the OVM language, and the methods of the corresponding classes
are written in the C language. The OVM files and the corresponding C files are compiled
together in order to produce class libraries that the OV server may load. The object meth-
ods that are written in C have access to the OV-Lib application programming interface, which
provides object-oriented programming facilities and additional services such as real-time task
management, logging, and object persistence.

The OVM language permits the definition of object classes which may use inheritance
to reuse and extend the functionality of objects in a hierarchical manner. Additionally, AC-
PLT/OV offers the possibility of defining *associations* between classes as part of the class
model defined in an OVM file. These associations represent links between the objects of the
corresponding classes at runtime, which are type-safe, enforce the specified cardinalities,
and may be explored and modified during the operation of the system.

An ACPLT/OV server is an ACPLT/KS server at the same time. This means that the object
model within an OV server is accessible by ACPLT/KS clients in the ways that were previously
explained. The base ACPLT/OV object library is designed as a self-describing meta-model,
which allows the querying of properties such as the class of an object, and the instances of a class, by exploring the objects and links in the model from within the OV server as well as through the ACPLT/KS interface. This provides a very flexible and expressive model-based environment for developing applications within process control settings.

### 6.2.3 iFBSpro Function Block System

The iFBSpro function block system [42] is a commercial engineering environment for developing process control applications using the model of function blocks. It consists of a function block server and an engineering client.

The iFBSpro function block server is an OV server that has been extended to offer the features of a function block system, such as a function block object model, support for the cyclic execution of tasks, and backup services for function block databases. It provides a function block class library that may be used as a basis for developing new function block types through the use of inheritance. This permits the user to extend the functionalities offered by the system by developing and loading new function block libraries.

The iFBSpro engineering client, shown in Figure 6.1, is a graphical application that can connect to multiple iFBSpro function block servers via ACPLT/KS, and which permits the exploration, creation, configuration, interconnection and deletion of function blocks in these servers. Function block libraries, in the form of ACPLT/OV object libraries, may be loaded by the engineering client into a server in order to instantiate the function block classes contained in the libraries. The engineering client also permits the creation of backup copies of the object model contained in a function block server, and the restoration of this model from a backup copy.

### 6.2.4 fbkslib

The fbkslib [34] is a function block library for the iFBSpro function block system that offers function block classes for communicating with ACPLT/KS servers. When using this library, function blocks may be created which are able to read and write variable values that are specified by a given path within an ACPLT/KS server. Although this library may be used in any situation when access to remote values is required, it is specially useful for implementing the acquisition of process values in a periodic fashion.

### 6.2.5 Tks

Tks [41] is an extension package for the Tcl language that offers an application programming interface for interacting with ACPLT/KS servers using the ACPLT/KS protocol. Using Tks, a Tcl program can act as a full-fledged ACPLT/KS client, querying and setting variable values, exploring the structure of objects, creating, linking, unlinking and deleting objects. Given the ease of use of the Tcl language, the use of Tcl with the Tks extension package permits rapid prototyping of ACPLT/KS client applications, as well as a convenient manner of interacting with ACPLT/KS servers in a flexible way.
6.3 Formal Plant Meta-models

Processing plants are complex physical systems that may be described using different levels of detail. The traditional and canonical description of a processing plant is a P&ID diagram, which as explained in Chapter 3, offers a graphical representation of the structure of the plant that is intended to be interpreted by engineers. However, the need for processing plant information by means of computers has led to the development of formal plant descriptions that may be unambiguously represented and interpreted by machines. This section presents two examples of this kind of formal plant descriptions.

6.3.1 CAEX

CAEX [15] stands for “Computer-Aided Engineering eXchange”, and is a data exchange format for CAE systems. As mentioned in Chapter 3, CAE systems are engineering environments that permit the design of processing and production plants. They manage large amounts of plant information, and usually have proprietary models for representing and storing this information. Many different and incompatible CAE systems are in use at present, and there are many situations where it is desirable to be able to exchange this kind of information between CAE systems, as well as to offer 3rd-party applications the access to this information. For this, CAEX has been designed as a vendor-neutral file format, based on XML, that may be used to represent the information handled by CAE systems in a common way. By exporting their engineering databases as CAEX files, a CAE system is able to offer its plant information to external applications in a way that preserves the semantics of this information,
6.3.2 RIVA

RIVA \cite{28, 29} is a plant meta-model that may be used to describe the physical constitution of plants at various levels. The object-oriented model that is offered by RIVA may be used to describe the basic topology of the plant, the product connectivity of the plant, and the thermodynamical properties of the plant’s elements, all within a single model, but layered in a way that allows applications to use a RIVA model of a plant while concentrating on those details about the plant that are of interest. RIVA consists of a conceptual class model that has been implemented as a collection of ACPLT/OV libraries. This means that the RIVA model of a plant may be represented by an object model in an OV server, allowing applications to interact with this model via the ACPLT/KS communication system.

Figure 6.2 shows a graphical representation of a portion of a RIVA model for a given plant. The elements of the plant are described by \textit{units}, which correspond to the two bigger rectangles. These units contain \textit{interfaces}, depicted by small squares, which permit the interconnection of units, and also contain sub-elements called \textit{channels} that correspond to internal areas of the unit where products can mix freely. These channels are also connected by means of interfaces, and additionally contain sub-elements of their own that represent the various thermodynamical properties that may be exhibited by the material that occupies the channel during the operation of the plant. The principle of object containment plays a fundamental role within the RIVA model, and its clear structure and semantics makes it an ideal representation of the plant for use by applications that aim to provide model-based solutions to automation, monitoring and diagnostic problems in processing plants.

The creation of a RIVA model for a given plant can be done manually by creating,
parametrising and associating objects in an OV server. However, it is also possible to automatically convert a CAEX representation of a plant to a corresponding RIVA model. This has been implemented as an XML transformation step that generates a machine-readable representation of an instantiated RIVA model from a CAEX XML file. This technique provides a simple way of applying technologies based on RIVA to existent plants whose plans are available in the CAEX data exchange format.
7 A Prototypical Implementation

This chapter presents a prototypical implementation of a decentralised product flow path management system for the iFBSpro function block system, and which supports the automatic synthesis of a product flow path management system for a given plant based on a RIVA model of the plant. The system follows the functional requirements outlined in Chapter 2, applies the model and algorithms of Chapters 4 and 5, and its implementation is based on the ACPLT/OV and ACPLT/KS technologies presented in Chapter 6.

7.1 System Architecture

The prototypical implementation of the product flow path management system that is presented in this chapter corresponds to a function block system for the iFBSpro system described in Chapter 6, and is therefore defined by a collection of class libraries that define function block classes. Additionally, a deployed system consists of function block instances organised in a particular manner. Both of these aspects of the system are explained in the following sections.

7.1.1 Class Libraries

The class libraries of the product flow path management system are described in the following. Each of them encompasses a different aspect of the system's design, such as decentralisation, flow allowance, product flow path model, and the individual product flow path management algorithms. This clear separation of concepts and aspects in different libraries is deliberately done in order to make the system more flexible, easier to understand, and therefore, easier to implement and use. The attributes of the classes are explained, but not their methods, in order to give a unified presentation of the operation of the product flow path management system at a later point in the chapter. The complete ACPLT/OV models of these libraries are included in the appendix of this work.

Decentralisation Library

The Decentralisation library is used to construct decentralised, component-based systems as described in Chapter 5. It defines classes for decentralised components and their connectors, and additionally provides an abstract base class that defines the interface between a component and the different algorithms that the component hosts.

The classes of the Decentralisation library are described in the following.

- Component: Components are the basic objects of a decentralised system. These systems consist of a collection of components, which are connected with each other by means of connectors. Components have two attributes:
7.1 System Architecture

– **Type** (string): This attribute is used to assign a type to this component in a way that may be freely used by the system. A product flow path management system uses this variable to store the type of the plant element that the component represents.

– **Identifier** (string): This attribute is used to provide the component with an identifier which should be unique throughout the entire decentralised system.

---

**Connector:** Every component has any number of connectors that permit the bidirectional interconnection of the component with other components of the system, and these connectors correspond to instances of the **Connector** class. The attributes of a connector are the following:

– **InMsg, OutMsg** (string vector): The interconnection between connector objects permits bidirectional communication and is implemented by means of function block connections between the connectors. Every connector has the pair of function block ports **InMsg**, an input port, and **OutMsg**, an output port, and both are of type string vector. The connection between two connectors is then accomplished by creating a pair of function block connections between the connectors, so that the **OutMsg** port of each function block is connected to the **InMsg** port of the other. Because these objects are part of a decentralised system, it is possible that two connectors that are to be connected to each other reside in different function block servers. In this case, direct function block connections cannot be used to interconnect these objects, but rather, a technique that uses ACPLT/KS to establish a remote connection must be used, as is explained later on in this chapter. In any case, the connector objects are not aware of the kind of data connection that links them to other connectors, and their operation is the same in any of these contexts.

– **InCount, OutCount** (unsigned integer): The communication ports **InMsg** and **OutMsg** are used by the connector objects to send and receive multiple messages simultaneously. A message is, in its basic form, a sequence of strings that are called **items**. Therefore, multiple messages can be encoded in a string vector by concatenating the items of these messages. The attributes **InCount** and **OutCount** are used to indicate the number of messages that have been received respectively sent by the connector object through the communication ports.

---

**Algorithm:** As explained in Chapter 5, a decentralised component hosts several algorithms that are able to send and receive messages through the connectors of the component. In order to provide a flexible manner of adding algorithms to a component, the abstract class **Algorithm** has been defined as a basic interface that every algorithm implementation must adhere to. In this way, instances of subclasses of the **Algorithm** class may be added – even during operation – to a component, and may participate in the operation of the decentralised system. The component is unaware of the actual class of these algorithm objects, and of their corresponding implementation details. The **decorator design pattern** [19] is hereby followed, because each of the instances of the **Algorithm** class add new functionality to the component in a flexible manner, avoiding the opposite approach of adding algorithms to components by subclassing the **Component** class which is cumbersome because it creates monolithic component objects that are forced to implement the decentralised algorithms themselves.
The instances of the classes of the Decentralisation library are bound together at runtime by means of associations between these objects. These associations are described in the following.

- **ConnectorContainment**: This association binds one component object with all of its connector objects. In this manner, a component has a way of knowing how many and which connectors it has, and a connector is able to know which object is the component that contains it.

- **AlgorithmContainment**: Algorithms, any number of them, are bound to a single component by means of this association, which allows the component to know how many and which algorithm objects are assigned to it, and an algorithm is able to know which object is its assigned component.

**FlowAllowance Library**

The flow allowance model that was presented in Chapter 4 is implemented by the FlowAllowance library. It defines classes of objects that describe the flow allowance properties at the level of plant elements and their connectors, and because of the one-to-one mapping between plant elements and components, the flow allowance objects are associated to decentralised components and to their corresponding connectors.

The classes of the FlowAllowance library are described as follows.

- **ComponentFlowAllowance**: The flow allowance model $\Phi(e)$ of a plant element $e$ is represented by an instance of the ComponentFlowAllowance class, which is associated to the corresponding component of the plant element $e$. This instance describes the flow allowance settings $\phi_e \in \Phi(e)$ of the element $e$, and provides a way to determine the current flow allowance setting of $e$. Also, it is used to indicate if the element $e$ is a source or a sink of material, as described in Chapter 4. The attributes of this class are described as follows.

  - **Settings** (string vector): A plant element that is mapped to a component has one or more flow allowance settings, and in order to represent these settings in a way that makes it simple to identify them, every flow allowance setting of a component is given a name, for instance, “open”, “position2” or “fixed”. The **Settings** attribute is a string vector that holds the names of all flow allowance settings of the corresponding component. In this manner, three parameters are defined at the same time: first, the number $|\Phi(e)|$ of flow allowance settings of the component, given by the length of the vector; second, the names of the flow allowance settings; and third, a numeric identifier for each setting, given by the index of the setting in the vector.

  - **Source, Sink** (Boolean): These attributes indicate if the element $e$ that corresponds to the component is a product source respectively a product sink in the plant, that is, if $e \in E^\uparrow$ respectively $e \in E^\downarrow$.

  - **SettingSense** (unsigned integer): The **SettingSense** attribute is of great importance, because it represents the interface between the online plant information and the decentralised product flow path management system. This attribute
is an input variable that expects the numeric identifier of the currently active flow allowance setting of the corresponding plant element. This value must be provided to this object in a correct manner, and the way to achieve this is a plant-specific task. For instance, after modelling a two-position valve, the possible values that this attribute can receive are 0 and 1, and the actual values that the control system receives as confirmation of the position of the valve must be mapped to the expected values under consideration of the flow allowance model. Naturally, in the case of plant elements that have a single flow allowance setting, the value given to this variable must be 0, and this may be done by just setting this variable as if it were a parameter.

- **SettingCount** (unsigned integer): This attribute is an output variable that indicates the number of flow allowance settings that the corresponding element has.

- **Setting** (unsigned integer): The currently active flow allowance setting, as received by the attribute **SettingSense**, is indicated by the output variable **Setting**.

- **SettingName** (string): This attribute is an output variable that offers the name of the currently active flow allowance setting, as defined in the attribute **Settings**.

**• ConnectorFlowAllowance**: The part of the flow allowance model \( \Phi(e) \) of an element \( e \) that refers to the individual connectors of \( e \) is represented by instances of the **ConnectorFlowAllowance** class. These instances describe the input and output flow allowance model of the corresponding connector, and also determine the current flow allowance setting of the connector. The attributes of this class are described in the following.

- **InFlowSettings**, **OutFlowSettings** (Boolean vector): The values of \( \phi^I_I(e) \) and \( \phi^O_O(e) \) for every flow allowance setting \( \phi_i \in \Phi(e) \) of the element \( e \) are respectively defined for the corresponding connector \( c \) by means of these two Boolean vectors. An entry in the **InFlowSettings** vector respectively in the **OutFlowSettings** vector at the index \( i \) determines the value of \( \phi^I_I(c) \) respectively \( \phi^O_O(c) \) for the flow allowance setting of the component whose numeric identifier is \( i \). Consequently, the lengths of these two vectors must match the length of the vector **Settings** of the corresponding **ComponentFlowAllowance**, and the object will report an error if this is not the case.

- **InFlow**, **OutFlow** (Boolean): Based on the configured flow allowance model of the corresponding connector \( c \) and on the currently active flow allowance setting \( \phi_i \) of the corresponding plant element \( e \), the input and output flow allowance values for the corresponding connector \( c \), \( \phi^I_I(c) \) and \( \phi^O_O(c) \), are given by these two output variables respectively.

The associations that bind flow allowance objects to the objects they describe are outlined in the following.

- **ComponentFlowAllowanceMapping**: This association binds a component with its corresponding instance of the **ComponentFlowAllowance** class, in a one-to-one manner. This causes these two instances to be accessible from each other.
• ConnectorFlowAllowanceMapping: Analogously to the ComponentFlowAllowanceMapping association, the ConnectorFlowAllowanceMapping association binds a connector with its corresponding instance of the ConnectorFlowAllowance class in a one-to-one manner.

In this manner, the FlowAllowance library provides a way to define the flow allowance model of every element of the plant, and therefore, the flow allowance model of the entire plant, in a decentralised manner, because the information that is contained in this model and that regards each element and each connector is stored in — and accessed from — specific objects that are distributed across the system in the same manner as the objects they describe.

FlowPathAnalysis Library

The decentralised product flow path analysis algorithm that was presented in Chapter 5, whose purpose is to discover product flow routes in the plant, is implemented in the FlowPathAnalysis library. This library contains a single class that is described in the following.

• FlowPathAnalysisAlgorithm: Product flow path analysis, as presented in this work, is a decentralised algorithm, and instances of the class FlowPathAnalysisAlgorithm represent the decentralised participants of this algorithm. This class is a subclass of the Algorithm class from the Decentralisation library, and this allows each of its instances to be active within a single component of the decentralised system. For this, the algorithm object must be bound to its corresponding component by means of the AlgorithmContainment association. The attributes of this class are outlined in the following.

  – Targets (string vector): A product flow route specification is a pair of plant elements \((e, e')\). The search for flow routes that conform to this specification is initiated at the component of the initial element \(e\), and therefore, at the corresponding instance of the FlowPathAnalysisAlgorithm class. The Targets attribute is a string vector that is used to formulate flow route specifications for beginning flow route searches. This vector is an input variable that may be populated with the identifiers of components in the decentralised system. For every entry \(e'\) in this vector, a flow route specification \((e, e')\) is defined and a corresponding search is initiated, where \(e\) represents the local element of the corresponding component.

  – Open (Boolean): As mentioned in Chapter 5, a search for product flow routes can be restricted in order to only find flow routes that are currently open. This is indicated to the flow path algorithm by giving setting the Open attribute to a true value.

  – Count (unsigned integer): This attribute indicates the number of flow routes that have been found by the decentralised search. Because the algorithm works based on messages that are independent for the different flow routes that are found, the result of the search algorithm that is reported by this object can be partial, which means that at any time, the amount of flow routes that are found may increase as newly found routes are reported.
– **Hashes** (unsigned integer vector): For every flow route that is found by the flow path analysis algorithm, a hash value is computed by applying an arithmetic formula to the identifiers that compose the flow route, and with the intention of using this number for the identification of the flow route. The computed hash value of every flow route is stored in the attribute **Hashes**, which additionally assigns an index to every flow route by means of the corresponding index of the hash value in this vector. The same index is used for each flow route in the other attributes of this class.

– **Lengths** (unsigned integer vector): This attribute holds the length of every flow route that is found by the algorithm, that is, the number of elements that the flow route contains.

– **Offsets** (unsigned integer vector): This attribute assigns a number to every flow route, an offset, that is to be used when interpreting the value of the **Elements** attribute. The offsets in this vector always appear in increasing order, such that flow routes with higher indexes also have higher offsets.

– **Elements** (string vector): This attribute contains the sequence of elements that constitute every flow route that has been found by the algorithm. The element sequence of every flow route begins at the index of this vector that corresponds to the offset of the flow route as assigned by the **Offsets** attribute. This permits the use of a single vector for storing the element sequences of all flow routes that have been found.

– **Timestamps** (time vector): Every time a flow route is found, a timestamp is created in order to record the time of the finding. The attribute **Timestamps** stores the corresponding timestamp for every flow route that was found.

In this manner, the **FlowPathAnalysisAlgorithm** class provides a unified interface for initiating decentralised flow route searches, and for accessing the corresponding results.

**FlowPathMonitoring Library**

The **FlowPathMonitoring** library implements the decentralised flow path monitoring algorithm presented in Chapter 5. The form of operation of this algorithm is similar to that of the flow path analysis algorithm, and this library also contains a single class that is described in the following.

- **FlowPathMonitoringAlgorithm**: Analogously to the **FlowPathAnalysisAlgorithm** class, the **FlowPathMonitoringAlgorithm** class is a subclass of the **Algorithm** class and each of its instances may be active within a single component of the decentralised system. The attributes of this class are described in the following.

  – **Hashes** (unsigned integer vector): The monitoring of a flow path begins by giving the corresponding flow route to the flow path monitoring algorithm object that resides at the component of the initial element in the flow route. This is achieved by providing this algorithm with the flow route’s hash value, length, and complete sequence of elements. Multiple flow routes may be given simultaneously to the
algorithm for monitoring, and they are given in a specific order. The Hashes attribute of this class holds the hash value of each flow route that is to be monitored.

- **Lengths** (unsigned integer): The length of each flow route that is to be monitored is given in the Lengths attribute, following the corresponding ordering used for the Hashes vector.

- **Elements** (string vector): The sequences of elements of the flow routes to monitor are provided to the algorithm object through the Elements attribute by concatenating them in the same order as used in the Hashes and Lengths attributes. The fact the algorithm knows the order of the flow routes and the length of each flow route allows it to decompose this sequence into the individual flow route sequences for their use.

- **Count** (unsigned integer): The results of the execution of the monitoring algorithm begin with a report of the number of flow routes that are being monitored. This is given in the Count output variable.

- **OpenHashes** (unsigned integer vector): The result of the monitoring determines whether every flow route that is being monitored is open or not. This information is provided, conceptually, in the form of an open table that contains three columns: the hash value of the flow route, the timestamp of the result, and the Boolean value that indicates if the flow route is open or not. The attribute OpenHashes corresponds to the first of these columns, and the order of its entries corresponds to the order of the rows in the table.

- **OpenTimestamps** (time vector): This attribute holds the timestamp for each entry in the open table, that is, the column of timestamps of this table.

- **Open** (Boolean vector): The open indicator for each entry in the open table is provided by the Open attribute, that is, the column of open indicator values of this table.

- **AlertCount** (unsigned integer): The flow path monitoring algorithm also emits alerts when the possibility of product leaks and product mixtures is detected. The number of alerts that have been issued by the monitoring operation is given by the AlertCount attribute.

- **AlertHashes** (unsigned integer vector): The leak and mixture alerts are conceptually provided in a table, just like the values that indicate if the monitored flow routes are open. This table contains four columns: the hash value of the flow route that issued the alert, the timestamp of the alert, the type of the alert, and the sequence of elements of the branch flow route where the potential leak or mixture was detected. The flow route hash value for each entry the alerts table is given by the AlertHashes attribute.

- **AlertTimestamps** (time vector): This attribute provides the timestamp for each entry in the alerts table.

- **AlertTypes** (string vector): This attribute holds the type of alert for each entry in the alerts table. The types of alerts are, consequently, “Leak” and “Mixture”.

- **AlertOffsets** (unsigned integer vector): The sequence of elements of the branch flow route that caused the alert is provided using a combination of off-
7.1 System Architecture

sets and element sequence, just as in the case of flow routes. The offset in the element vector for the sequence of elements of each entry the alerts table is given by the AlertOffsets attribute.

– AlertElements (string vector): All the sequences of elements of the alert table are concatenated in the AlertElements attribute, following the order of the rows in this table.

This interface enables the initiation of decentralised monitoring operations of flow routes, and the access to the results of the monitoring, in a unified manner for each component of the decentralised system.

FlowPathAllocation Library

Analogously to the two algorithm libraries that have been presented, FlowPathAllocation is a single-class library that implements the decentralised product flow path allocation algorithm presented in Chapter 5. An algorithm object of this class is responsible for participating in the decentralised allocation and deallocation of flow routes, and manages the allocation state and the communication state that was defined for this algorithm. The class FlowPathAllocationAlgorithm is described in the following.

• FlowPathAllocationAlgorithm: This class is a subclass of the Algorithm class, and implements a node of the decentralised flow path allocation algorithm. The attributes of this class are described in the following. Many of these attributes may be recognised as direct representations of the global variables defined for the flow path analysis algorithm in Chapter 5.

  – Hash (unsigned integer): The flow route that is to be allocated or deallocated is given to this algorithm object in the form of the hash value of the flow route and its sequence of elements. The attribute Hash is used to indicate the hash value of the flow route to this object.

  – Elements (string vector): This attribute is used to give the sequence of elements of the flow route that is to be allocated or deallocated to the algorithm object.

  – Allocate, Deallocate (Boolean): The commands that initiate the allocation and deallocation of the given flow route are expressed through the values of the Allocate and Deallocate attributes. A value of true for any of these attributes indicates that the corresponding operation should be initiated, and the algorithm object performs this action if it is idle when the corresponding command is given. If an operation is currently being executed, the values of these attributes are ignored.

  – ManualSetting (signed integer): As explained in Chapter 5, the automatic selection of a flow allowance setting with the purpose of providing protection against leaks and mixtures along branch flow routes may be overridden by the user if a flow allowance setting is manually given. The ManualSetting attribute is used to provide this manual setting to be used for enclosure constraints. If the value of this attribute is negative, then an automatic selection procedure is employed, and if the value is zero or positive, then it is interpreted as the manually chosen flow allowance setting. An error is issued if the value of this attribute does not correspond to a valid flow allowance setting of the corresponding plant element.
- Allocation, Deallocation, Conflict (Boolean): The outcome of the last operation that was initiated by the algorithm object is indicated by the values of these three attributes. At any time, only one of them may have a value of true, and they respectively indicate that a successful allocation, successful deallocation, or a conflict has occurred.

- ResultElements (string vector): This attribute holds the elements of the flow route that has been allocated or deallocated by the last operation that was initiated by this algorithm object, or the elements of the flow route for which an allocation or deallocation conflict was issued.

- Alloc (string vector): This attribute corresponds to the variable of the same name that was presented in Chapter 5, and holds the elements of the flow route that has allocated the current component.

- EncloseHashes (unsigned integer vector): The value of the Enclose variable from the formal definition of the algorithm is represented by means of a conceptual enclose table that contains two columns: the hash value of each flow route contained in Enclose, and the sequence of elements of each flow route. The attribute EncloseHashes holds the first of these columns.

- EncloseOffsets (unsigned integer vector): This attribute stores an element offset for each flow route that is contained in the enclose table. This offset determines the index of the first element of the corresponding flow route in the EncloseElements vector.

- EncloseElements (string vector): This attribute contains the concatenation of the elements of each flow route that is contained in the enclose table.

- Setting (unsigned integer): If one or more enclosure constraints have been set upon the local component, the flow allowance setting of the enclosure constraint is indicated by the Setting attribute.

- Operation (string): The name of the operation that is currently being executed by the algorithm object is stored in the Operation attribute. The valid operations of the algorithm have been described in Chapter 5.

- Parent (string): When a request is first received by the algorithm object, the connector of the decentralised component where the request was received, denoted the parent connector, is registered in the Parent attribute by storing the object’s name.

- Children (string vector): Analogously to the parent connector, every remaining connector where a request message is sent is called a child connector, and the names of these connectors are stored in the Children attribute.

- FlowRoute (string vector): This attribute corresponds to the variable of the same name from the formal definition of the algorithm, and holds the elements of the flow route that corresponds to the current operation.

- FlowRouteHash (unsigned integer): This attribute holds the hash value of the flow route that is stored in the FlowRoute attribute.

- BranchFlowRoute (string vector): The contents of the global variable BranchFlowRoute defined for the flow path allocation algorithm is represented by the
7.1 System Architecture

value of this attribute, which stores the sequence of elements of the branch flow route that is currently being allocated or deallocated.

- **ResponseOffsets** (unsigned integer vector): An important part of the communication state of the algorithm object is the global variable Response defined in Chapter 5. This variable stores those response messages that have been received from child nodes. A conceptual response table is used to represent the contents of this variable, and the attribute ResponseOffsets assigns an item offset for every entry in this table, that is, for every response message that is stored in it.

- **ResponseItems** (string vector): This attribute stores the concatenation of the items of every response message that has been received and stored. The index of the initial item of every message is given by the corresponding offset from the ResponseOffsets vector.

- **ConstraintElements** (string vector): When a flow route is allocated, a set of flow allowance setting constraints are included which indicate which elements should be restricted to a particular flow allowance setting in order to guarantee the enclosure of the allocated flow route. These constraints are given by a conceptual constraint table with two columns: the element of the constraint, and the corresponding flow allowance setting. The attribute ConstraintElements holds the first of these columns.

- **ConstraintSettings** (unsigned integer vector): This attribute holds the column of flow allowance settings of the constraint table, in accordance with the attribute ConstraintElements.

- **ConflictCount** (unsigned integer): During allocation and deallocation operations, conflicts may be detected and reported by the decentralised algorithm. The attribute ConflictCount reports the number of conflicts that have been reported by the algorithm.

- **ConflictOffsets** (unsigned integer vector): The set of conflicting flow routes that have been found are given by a conceptual conflict table, which has two columns: the element sequence of this flow route, and a timestamp which indicates the time when the conflict was detected. The attribute ConflictOffsets assigns an element offset to every entry in the conflict table.

- **ConflictElements** (string vector): This attribute stores the concatenation of the element sequence of every flow route in the conflict table. The index of the first element of each flow route is given by the corresponding value from the ConflictOffsets vector.

- **ConflictTimestamps** (time vector): This attribute stores the column of timestamps of the conflict table.

- **Timestamp** (time): The time of the last successful allocation or deallocation operation, as well as the time of the last conflict that was reported, is indicated by the Timestamp attribute.

The large number of attributes of the FlowPathAllocationAlgorithm class provides a unified interface for allocating and deallocating flow routes that begin at the corresponding plant element of the local component. Also, it is possible to determine,
based on this interface, the allocation status of the component with respect to other flow routes.

**FlowPathManagement Library**

The model of product flow paths that was presented in Chapter 2 is implemented in the FlowPathManagement library. It provides the classes found in this model, and uses the Decentralisation library and the corresponding algorithm libraries for implementing a decentralised product flow path management system. The classes of this library are explained in the following.

- **FlowPathManager**: Instances of this class act as factories and containers of flow route and flow path objects. They provide an interface for discovering product flow routes in the plant, which consists of the following attributes:
  
  - **Discover** (Boolean): This attribute is used to activate and deactivate the automatic discovery of product flow routes, that is, the operation of the decentralised product flow path analysis algorithm and the automatic creation of flow route objects for newly found flow routes.
  
  - **Origins, Targets** (string vector): The flow route specifications that are used for the automatic discovery of flow routes are given to the flow path manager object by means of the attributes Origins and Targets, which together conform a conceptual table with a column of origin or initial plant elements, and a column of target or final plant elements. Every element that is given in the Origins attribute must correspond to a component which is local to the flow path manager object. This relationship between the objects will be explained later on in this chapter.

- **FlowRoute**: An instance of the FlowRoute class represents a product flow route in the plant, and contains the following attributes.
  
  - **Description** (string): This attribute is used to give a readable description of the flow route for practical purposes. The user is able to set this attribute to any value that is meaningful in the context of the operation of the plant.
  
  - **Origin, Target** (string): These attributes respectively hold the origin, or initial plant element, and target, or final plant element, of the flow route.
  
  - **Hash** (unsigned integer): The hash value of the flow route, which is computed by applying a hash function to the sequence of plant elements in the flow route, is stored in the corresponding Hash attribute of the flow route object.
  
  - **Length** (unsigned integer): This attribute holds the number of plant elements that the flow route contains.
  
  - **Elements** (string vector): The complete sequence of plant elements contained in the flow route is stored in the Elements vector.
  
  - **Select** (Boolean): This attribute is used to indicate that the flow route has been selected for its use by a corresponding flow path object. Based on this selection, the flow path manager object that corresponds to the flow route object creates a product flow path object if this object does not exist already.
7.1 System Architecture

- **FlowPath**: Product flow paths in the plant are represented by means of flow path objects which are instances of the FlowPath class. These instances are created by the local flow path manager object when a corresponding flow route object is selected. Flow path objects offer the interface for executing product flow path operations described in Chapter 2, and consist of the following attributes.

  - **CreationTime** (time): This attribute stores the time when the flow path object was created.
  
  - **Allocated, Locked, Active** (Boolean): These attributes act as indicators for the current state of the flow path object as described in Figure 2.1. The values of these attributes are initially false, and may change to true in an incremental manner.
  
  - **AlertCount** (unsigned int): The number of alerts that have been issued by a flow path object is given by the AlertCount attribute.
  
  - **Monitoring** (Boolean): This attribute indicates if the monitoring operation is active for this flow path object.
  
  - **Timestamp** (time): Whenever a flow path object changes its state as described in Figure 2.1, a timestamp of the state transition is made and stored in the Timestamp attribute of the object.
  
  - **Allocate, Lock, Activate, Delete, Monitor** (Boolean): The methods of a flow path object that are used to give commands to this object, as presented in Chapter 2, are implemented in this prototype by means on Boolean input variables, following the principles of function block technology. When a true value is given to these variables, it is interpreted that the corresponding command has been given to the object. Also, when a value of false is given, the complementary command is assumed. For instance, a true value at the variable Allocate is interpreted as a call to the method Allocate, and a false value at this variable is interpreted as a call to the method Deallocate. As explained in Chapter 2, the current state of the flow path object determines which of these commands may be executed, and therefore, the commands are ignored if they are issued at an incompatible state. Apart from the requests for state transitions, the variable Monitor is used to request the activation and deactivation of the monitoring operation for the corresponding flow path object.
  
  - **Open** (Boolean): The monitoring of a product flow path determines if the flow route that corresponds to the flow path is currently open, and this is indicated by the value of the Open attribute. This value is only updated if the monitoring operation is active for the flow path object.
  
  - **OpenTimestamp** (time): The timestamp of the last update of the Open attribute is given by the OpenTimestamp attribute.
  
  - **ConstraintElements** (string vector): When a flow path is allocated, this allocation is accompanied by a set of flow allowance setting constraints for plant elements as explained in Chapter 5. This set is represented in a conceptual table with two columns: a column of plant elements, and a column of flow allowance settings. The attribute ConstraintElements holds the first of these columns.
- **ConstraintSettings** *(unsigned integer vector)*: This attribute stores the second column of the table of flow allowance setting constraints, that is, the constrained flow allowance setting of each element in the vector `ConstraintElements`.

- **ExtLock, ExtActivate, ExtLocked, ExtActive** *(Boolean)*: As explained in Chapter 2, the locking and activation of product flow paths is not covered in this work, and is therefore not implemented by the prototype that is presented in this chapter. Nevertheless, it is possible to couple this system with a component that performs these tasks. When a flow path object is instructed to lock or activate itself, it first determines if the operation is allowed from its current state, and then reissues these commands through its output variables `ExtLock` and `ExtActivate`. These variables may in turn be used by the system to initiate the corresponding operations in a system-dependent manner. When the operation has been completed successfully, the system may communicate this to the flow path object by yielding a value of true to the attributes `ExtLocked` and `ExtActive`, respectively. This value is then interpreted by the flow path object as a confirmation of the new state.

- **FlowPathAlert**: The instances of the class `FlowPathAlert` represent the alerts that the flow path may issue during operation, which may be of any meaningful kind. For instance, the alerts that are issued as a result of the monitoring of a product flow path, as well as the conflicts that occur during allocation and deallocation operations, are represented by instances of this class. These objects have the following attributes.

  - **Type** *(string)*: The type of the alert, which is given by a text string that can be freely chosen to indicate the kind of alert that the object represents, is stored in the **Type** attribute.

  - **Timestamp** *(time)*: This attribute stores the timestamp of the alert, that is, the time when the alert occurred in the system.

  - **Elements, FlowPaths** *(string vector)*: An alert may refer to elements in the plant, for instance, the sink element that is reached by a leak or the source element where a mixture begins. The **Elements** attribute is used to indicate those plant elements that are related to the source of the alert. Likewise, an alert may refer to flow paths, such as those that cause conflicts when the flow path tries to perform an allocation. For this, the **FlowPaths** attribute is provided, which may be used to indicate the identifiers of those flow paths that are related to the cause of the alert.

The **FlowPathAlert** class permits a general way for reporting alerts related to product flow paths, and the information contained in their attributes can be used to present alert reports to the users of the system.

- **FlowPathLogger**: As explained in Chapter 2, the **FlowPathLogger** class describes objects that have the task of documenting the life cycle of product flow paths. In this implementation, **FlowPathLogger** is an abstract class that defines the interface to a logging service, and may be sub-classed in order to define different logging techniques. The instances of these classes are registered to flow path objects following the observer design pattern, as explained in Chapter 2.
7.1 System Architecture

- **OVLogfileFlowPathLogger**: This implementation of a product flow path management system provides the definition of a flow path logging technique that uses the AC-PLT/OV logfile service for registering the life cycle of product flow paths. This technique is implemented in the class OVLogfileFlowPathLogger.

The FlowPathManagement library defines several associations that are used to bind the objects of a product flow path management system within an iFBSpro server. These associations are outlined in the following.

- **ComponentAssignment**: Every server that participates in a decentralised product flow path management system contains a single instance of the FlowPathManager class, and at the same time, any number of instances of the Component class. The ComponentAssignment association is used to link a flow path manager object with every decentralised component that operates within the same server.

- **FlowRouteContainment, FlowPathContainment**: The flow routes and flow paths that are managed by a flow path manager object are respectively associated with this object by means of these two associations.

- **FlowRouteUse**: This association binds a flow route object to its corresponding flow path object. It is one-to-one association that guarantees that at most one flow path exists for every flow route in the plant.

- **FlowPathLoggerAssignment**: Flow path logger objects must be related to the flow path objects that they document, and this is accomplished indirectly by registering the flow path logger objects to the local instance of the flow path manager object by means of the FlowPathLoggerAssignment association. In this manner, every flow path object that is managed by the flow path manager object may be logged by every flow path logger object that is registered with the flow path manager object through this association.

- **SourceComponentFlowRouteMapping**: When a flow route is created, it is linked to the component that corresponds to the initial element of the flow route by means of the SourceComponentFlowRouteMapping association. This link is used during operation by a corresponding flow path object in order to interact directly with this component and perform the decentralised operations that must be executed by the flow path.

- **FlowPathAlertContainment**: The flow path alert objects that are created by a flow path object are linked to this object by means of the FlowPathAlertContainment association. This allows a clear way to determine which flow path alerts correspond to which flow paths during the operation of the system.

7.1.2 Instance Model

This section explains how a decentralised product flow path management system is built from instances of the classes that have been described in the previous section. Many of the details of this instance model have already been given in the description of the classes of the system, and a clear definition of this model will now be given.
Systems, Servers and Components

The instance model of a decentralised product flow path management system may be described at three levels: the level of the complete system, the level of a single server within the system, and the level of a single component within a server. Figure 7.1 presents a detailed decomposition of a product flow path management system at every one of these three levels, which will be explained throughout this section.

A decentralised product flow path management system as hereby implemented is conformed of a collection of servers, which in this case correspond to iFBSpro function block servers. These servers may communicate with one another using the ACPLT/KS communication protocol. In its simplest form, a product flow path management system contains only one server which operates in a local manner and does not need to communicate with other servers in this way. In general, multiple servers exist, and each server hosts a collection of objects. These objects are organised in a particular manner, using the associations defined in the class libraries for binding these objects together. Every ACPLT/OV system organises objects in a tree structure by using the ov/containment association, and this may used for arranging the objects of a server in order to facilitate browsing through these objects. Nevertheless, no given tree organisation is assumed or required for the correct operation of the system.

The central object of every server is an instance of the FlowPathManager class. One or more decentralised components, instances of the Component class, are bound to this object. Also, any number of flow path logger objects may be registered with this flow path manager object as well. Flow route objects may be created as a result of the discovery of flow routes in the plant, or by a direct instantiation and parametrisation from the user. In both cases, flow route objects are linked to the flow path manager object as well, and additionally, they must be linked to the component that corresponds to the first element in the corresponding flow route, which in turn must be a local object in the server. When a flow route is selected for use, the flow path manager creates a corresponding flow path object, which is bound to the flow path manager and to the flow route object as well.

Every component has a collection of objects directly associated to it. The connector objects of a component are bound to it through the ConnectorContainment association. In turn, every connector has a corresponding instance of the ConnectorFlowAllowance class that describes the flow allowance properties of the connector. This object is bound to its corresponding connector, one-to-one. Similarly to connectors, every component is linked to an instance of the ComponentFlowAllowance class that describes the flow allowance model of the plant element that corresponds to the component. Finally, the AlgorithmContainment association is used to link an instance of every decentralised flow path management algorithm to the component object. In this way, these algorithms are able to operate within the context of the component object, and communicate with other algorithm objects through the messages that are sent and received by the connectors of the component.

Interconnection of Connectors

As explained in Chapter 5, in order for decentralised components to communicate with each other, their corresponding connectors must be connected to each other by means of bidirectional communication links. The present implementation is based on function blocks, and therefore uses function block connections in order to implement these links. However, these
Figure 7.1: Diagram of the instance model of a product flow path management system at the system, server and component levels.
links must sometimes operate across two different servers, and this is not supported by function block connections under iFBSpro. Therefore, different interconnection techniques must be employed when local and remote connections are established.

Figure 7.2 depicts the two kinds of interconnection techniques that are used in order to implement bidirectional communication links between two connector objects. In the case that a local connection is required between two connectors which lie within the same server, a pair of function block connections are established, each from the OutMsg port of one connector to the InMsg port of the other connector. In this manner, each connector is able to send a message to the other connector through a corresponding function block connection. When the two connectors that are to be connected lie in different servers, a different interconnection scheme is employed with the help of the KsGetVec function block class from the library fbkslib that was presented in Chapter 6. Instances of this class implement the retrieval of vector values that are stored in the attributes of remote objects by means of the ACPLT/KS protocol. For this, the full ACPLT/KS path of the remote attribute must be given as a parameter to the function block. For every connector that is to be connected remotely, an instance of the KsGetVec is provided and its source path is configured to the ACPLT/KS path of the OutMsg port of the other connector object. Then, the StringVec output port of each of these objects, which yields the value of the remote string vector, is connected to the InMsg port of the corresponding connector object, while the OutMsg port of this object is left unconnected. In this way, each side of the communication link is responsible for retrieving the value of the OutMsg port of the connector object on the other side, and a bidirectional communication channel over the network is established.
7.1.3 Execution Model

So far, the class model and instance model of a decentralised product flow path management system have been presented. In this section, we explore the dynamics of the system, that is, the ways in which the objects interact during the operation of the system.

Messaging

The base of the operation of the decentralised system is the sending and receiving of messages on the part of the components. These messages are sent and received by the connectors of these components, by writing the outgoing messages to their OutMsg port, and by reading the incoming messages from their InMsg port. Since the objects of the system are function blocks, their execution proceeds in cycles. At each execution cycle, a connector object receives all incoming messages of the cycle at once, and sends all outgoing messages of the cycle at once. The values of the communication ports are string vectors, which means that a string vector containing all messages that were produced at the current cycle is sent and received at each cycle. The value of this string vector is structured as

| Sender | Message₁ | ... | Messageₙ |

where Sender is the identifier of the component that is sending the message and Message₁, ..., Messageₙ are n messages encoded each as a sequence of strings. The use of the Sender value allows the receiving component to know which component is connected to it over each one of its connectors, and this is required for implementing the decentralised algorithms presented in Chapter 5. In turn, every message is structured as

| Message_Type | Message_Identifier | Message_Length | ... |

where Message_Type is an indicator of the type of the message, Message_Identifier is a unique identifier of the message and Message_Length is the number of string items in the message, encoded itself as a string. Every message is issued by one of the decentralised algorithms, and the body of the message depends on the type of the message and carries the information required by the corresponding decentralised algorithm for messages of this type. Algorithms are free to format messages in any way they require, as long as the basic form shown above is maintained. This allows component and connector objects to handle messages without needing to know about their internal structure or their meaning. Also, it permits the addition of new algorithms to the system, even during operation, without having to adapt or parametrise the components and connectors of the system in any way.

The reliable communication channels that are required by the decentralised algorithms of Chapter 5 are implemented in this prototypical system as follows. When a message is sent by a component, it is placed on the output message port of the corresponding connector. The value of this port can be seen as a message buffer that is transmitted to the input message port of the corresponding connector in its entirety. When the receiving component has processed the message, it sends an acknowledgement message back to the sending component by placing this message in its own output message port, and this message contains the unique identifier of the message. When the sending component receives the acknowledgement, it removes the message from the output message port, and when the receiving component detects that the acknowledged message is no longer being sent, it removes the
acknowledgement from its own outgoing message port. After this, the delivery of the message is complete. Because the messages are resent within the message buffer while the sending component waits for the corresponding acknowledgements, it is guaranteed that all messages are eventually delivered, even in the case of asynchronous and uneven scheduling of the two components that are involved in the communication. Also, a receiving component keeps track of those messages that have been received, and is then able to discard duplicate messages and guarantee that every message is delivered to the corresponding algorithm only once. Finally, the message buffer is organised in a fifo mode, which causes the implemented communication channel to be fifo as well.

Flow Path Management Operations

The operations that correspond to the management of product flow paths begin with the creation of flow route objects. These objects may be created manually by the user, or can be created automatically by a flow path manager object. This process will now be explained with the help of an example. Figure 7.3 shows the faceplate of an active flow path manager object that has been parametrised to perform the automatic discovery of product flow routes that begin at plant element /TU10/U_B1 and that end at plant element /TU20/U_B2. The flow path manager object proceeds by first identifying the decentralised component that corresponds to the initial element of the flow route, in this case /TU10/U_B1. The flow path manager object does this by searching through all those components that are linked to it by means of the ComponentAssignment association. If no such component is found, then the operation is aborted; otherwise, the flow path analysis algorithm object that is registered with the component is identified and correctly parametrised in order to initiate the execution of the flow path analysis algorithm. The flow path manager object periodically queries the results of the flow path analysis, and for every flow route that is found in this way, a corresponding flow route object is created. Figure 7.4 shows the tree view of the flow path manager object within a server, where several flow routes that have been found in this way are displayed. The names of these objects are created by encoding the length of the flow route and the hash value of the flow route within the name.

The faceplate of the first flow route that is listed in Figure 7.4 is shown in Figure 7.5. This faceplate shows the information from the corresponding flow route that has been stored in the attributes of this flow route object. By setting the value of the Select indicator to true, the local flow path manager creates a corresponding flow path object for this flow route. Figure
7.1 System Architecture

Figure 7.4: Tree view of the objects of the flow path management system within a server.

Figure 7.5: The faceplate of a flow route object.

7.4 shows this new flow path object as a child of its corresponding flow route object, and Figure 7.6 shows the faceplate of this flow path object.

Once a flow path object such as the one shown in Figure 7.6 has been created, it may be used in conformance with the life cycle of product flow paths as specified in Figure 2.1 by giving commands to the object through the manipulation of its input variables. In order to allocate the flow path, the Allocate variable must be set to true, in which case the flow path object begins the allocation of the corresponding flow route in the plant by first identifying the decentralised component that corresponds to the initial element of this flow route. This is done by following the links from the flow path object to the flow route object, and from this object to the component of this initial element. Once this component has been identified, the instance of the corresponding flow path allocation algorithm is identified, and it is then used to request the allocation of the flow path. This allocation may succeed or not, and the faceplate of the flow path object is updated to reflect this. A similar procedure is followed for performing the monitoring of the flow path, that is, by identifying the corresponding flow path monitoring algorithm object and interacting directly with it in order to obtain monitoring information for
Figure 7.6: The faceplate of a flow path object.

the flow path. This interaction design permits the access to relatively complex decentralised operations through a single, unified interface in the form of the product flow path object.

As was mentioned earlier in this chapter, flow path objects create flow path alert objects for reporting situations such as leak and mixture alerts emitted from the flow path monitoring algorithm, and conflicts detected during the allocation or deallocation of the flow path. These flow path alert objects are created by the flow path object, but are not deleted by it. Rather, it is expected that the system deletes these objects appropriately, say, after having been acknowledged by an operator. In any case, when a flow path deletes itself, it additionally deletes any existing flow path alarm objects that correspond to it.

Every time that a flow path object changes its state or creates a new flow path alert object, the flow path logger objects that are registered with the local flow path manager object must be notified of this event in order to register it. This is done by the flow path object, which follows the link to the flow path manager object, and iterates through every flow path logger object that is registered with it, calling the corresponding methods of the flow path loggers in order to communicate the state change or the creation of the alert. In turn, each of these flow path logger objects registers this information in a way that depends on the implementation of the object. An instance of the class `OVLogfileFlowPathLogger` is shown in Figure 7.4, which commits the information it receives to the ACPLT/OV logfile service.
Finally, if a flow path object should be deleted, its `Delete` attribute can be set to true, which will cause the flow path object to delete itself as soon as the object finds itself in a deallocated state, as is specified by the life cycle of product flow paths that was shown in Figure 2.1.

### Execution Times

The execution time needed by the different operations of the prototypical flow path management system depends on the configured `cycle-time` of the iFBSpro function block system. In this system, all function blocks are executed following a round-robin scheme, and the cycle-time of the system determines the period between the start of two consecutive iterations. At each iteration, the components of the system process their incoming messages and emit the respective outgoing messages, which means that the actual minimal time for the completion of a flow path operation may be determined as a product of the cycle-time and the “distance” that the messages must travel across the system. These distances are in direct relation to the lengths of the flow route and the branch flow routes that are traversed by the decentralised algorithms, in agreement with the time complexities that were presented in Table 5.1 when considering the full decentralised system. For instance, the discovery of a flow route of length $l$ requires at least $2l$ iterations, and at least $2lc$ time units if $c$ is the configured cycle-time of the system. Furthermore, the monitoring and allocation of flow paths both require at least $2l'c$ time units for their completion, where $l'$ is the maximal length of an element sequence from the initial location of the corresponding flow route that follows the flow route and any branch flow route that is involved in the algorithm. In the case of deployments that are distributed across multiple function block servers, the communication delays and the differences among the cycle-times of the various servers must also be taken into consideration.

In any case, the cycle-time $c$ must be greater than the time $i$ it takes for all components to execute one iteration. If this is not the case, then the execution of the system cannot satisfy the configured cycle-time and the real-time properties of the system are violated. Therefore, $c$ must be configured such that $i < c$ is satisfied, and because $c$ determines the minimal execution time of the flow path operations, then the value of $c$ should also be as small as possible. Here, the time complexities of the individual components that were presented in Table 5.1 are to be taken into account, and these scale linearly with the size of the plant in the worst case. In Chapter 8, empirical determinations of $i$ for given applications are presented, which suggest lower bounds for the cycle-time $c$ used for the flow path management system.

### 7.2 System Synthesis

The object structure that corresponds to the instance model of a product flow path management system may be automatically created from a representation of the structure of the plant, as specified in Chapter 2 and explained in Chapter 5. The prototype that is presented here accomplishes this form of system synthesis based on a RIVA model of the plant. This section outlines the details of this approach.
7 A Prototypical Implementation

7.2.1 Synthesis Program

The synthesis of decentralised component-based systems based on a RIVA model of the plant is implemented as a program written in the Tcl language, and Figure 7.7 depicts the general operation of this program. The synthesis program uses the TkS extension for Tcl, which enables the program to access iFBSPro servers using the ACPLT/KS protocol. When the program is executed in an engineering station machine, the program connects to an iFBSPro server that hosts an instantiated RIVA model of the plant, and explores this model by performing a traversal of the object tree in the server. Based on the information found in this model, the synthesis program creates and parametrises objects in one or more iFBSPro servers that are designated to host the target system, and these objects correspond to the instance model of a flow path management system as shown in Figure 7.1. Also, the synthesis program reads several parameter files from the engineering station where the program executes. The purpose of these files will be explained later on in this section.

The synthesis program applies mappings from object structures found in the RIVA model to object structures in the flow path management system, and the kind of these mappings is exemplified in Figure 7.8. A RIVA unit object that represents a tank \texttt{U_B1} with three product connectors is shown on the left. The unit contains a single channel that permits the free mixture of products, and three corresponding internal connectors for this channel. In addition, other objects that represent sensor signals and thermodynamic properties are shown. When the synthesis program encounters this structure, it creates a decentralised component that corresponds to this plant element, which is shown on the right. In following the principle that a component represents a single plant element that permits the mixture of materials in its interior, the synthesis program maps every RIVA channel to a component, and therefore, the component that is created in this case bears the name of the channel. This means that a unit that contains multiple channels will be mapped to an equal number of decentralised components. In order to still reflect the structure of the plant in a one-to-one manner, a domain object is created for every unit, and every component that is created for a channel contained in the unit is created within this domain object, as may be seen in Figure 7.8. The synthesis program also creates one component connector for every one of the product connectors found
7.2 System Synthesis

Figure 7.8: Synthesis of decentralised components from RIVA units.

in the corresponding channel. For each one of these connectors, a connector flow allowance object is created and associated to the connector. Finally, a component flow allowance object and the necessary algorithm objects are created for the component and associated to it. This procedure is repeated for every channel of every unit that is found in the RIVA model of the plant.

The decentralised components in the target system are interconnected by the synthesis program in the same manner as their corresponding units and channels in the RIVA model of the plant. As was explained earlier, and shown in Figure 7.2, component connectors are interconnected in different ways, depending on the relative location the connector objects. The synthesis program follows this interconnection scheme and creates local function block connections between connectors that are placed on the same server, and remote connections using objects of the \( \text{KsGetVec} \) class between connectors that are hosted on different servers. This decision is taken automatically by the synthesis program based on the location of the connector objects, and therefore, in a way that is transparent for the user.

7.2.2 Parameter Files

There are some objects, such as the flow path manager object, which must be created once in every server of the decentralised product flow path management system. However, the synthesis program must decide in which server to place the decentralised components of the target system that it creates, and this is done with the help of a target server parameter file, which is one of the parameter files that are indicated in Figure 7.7. A target server parameter
file consists of a sequence of lines of text, where each line has the format
\[\langle \text{Pattern} \rangle \langle \text{Host Name} \rangle \langle \text{Server Name} \rangle \langle \text{ACPLT/KS Path} \rangle.\]

The pattern of each line is a regular expression that must conform to the format used by Tcl for string matching. This pattern is used by the synthesis program to select those RIVA objects whose ACPLT/KS path matches the pattern. For these objects, the name of the server that is indicated in the third position of the line is used as the target server for creating the corresponding decentralised component and its related objects; for this, the name of the host indicated in the second position is used to locate the server within the network. The ACPLT/KS path that is given in the fourth position of the line is used as a prefix for the creation of these objects. The use of a target server parameter file provides the user with a very simple yet very expressive technique for defining the deployment of the decentralised system. For instance, if a target server parameter file with the content
\[*/TU10/* tu10 fb_tu10 /TechUnits/TU10\]
\[*/TU20/* tu20 fb_tu20 /TechUnits/TU20\]
\[*/TU30/* tu30 fb_tu30 /TechUnits/TU30\]
is used, then every RIVA unit that is contained within a domain TU10 will be mapped to a component in the server tu10/fb_tu10 and within the domain /TechUnits/TU10; the same holds analogously for the RIVA objects within the domains TU20 and TU30. This is very useful, for instance, when the RIVA model is organised to reflect the subdivision of the plant into plant sections, and it is desired to use this same sectioning scheme for deploying flow path management objects in separate servers. However, the technique is general enough to support many different deployment schemes.

The objects that are created by the synthesis program must also be parametrised in order to function properly. For instance, the flow allowance objects of components and their connectors must be parametrised with the information corresponding to the flow allowance model of the plant elements represented by the original RIVA objects. Whereas RIVA supports the representation of the type of each plant unit, the actual flow allowance model of these elements is not represented therein, and this information must therefore be obtained elsewhere. For this, a component parameter file is used, which is another one of the parameter files shown in Figure 7.7. The goal of a component parameter file is to permit any type of parametrisation to be specified for decentralised components, and the format of this file is designed to be general enough to accomplish this. A component parameter file consists of a sequence of lines of text, where each line has the format
\[\langle \text{Pattern} \rangle \langle \text{Relative ACPLT/KS Path} \rangle \langle \text{Variable Name} \rangle \langle \text{Value} \rangle.\]

The synthesis program uses a component parameter file as follows. First, the pattern at the first position of each line, which again must be a valid regular expression for Tcl, is used to select those components whose type or whose identifier matches the pattern. For these components, the object that is found at the relative path given by the second item of the line is identified, where this path is interpreted as being relative to the component object. Having identified this object, the third item of the line is interpreted as an attribute of the object, and this attribute is set to the value given by the fourth item of the corresponding line. In this way, any variable of any object that may be indicated by a relative path with respect to the component may be set to a given value. For instance, the component parameter file with the content
may be used to parametrise the flow allowance objects of components that represent two-position valves and tanks. Assuming that the pattern of the first five lines matches the identifier of every two-position valve in the system, the synthesis program sets the Settings attribute of the corresponding component flow allowance object to the values CLOSED and OPEN, which indicate two flow allowance settings for the component. Also, the input and output flow allowance settings of the connector flow allowance object corresponding to both connectors of the component are set to the values FALSE and TRUE accordingly, thus describing the behaviour of the two-position valve. Also, the last two lines of the file instruct the synthesis program to configure every tank in the plant as both a product source and a product sink, assuming that the identifiers of these elements match the corresponding pattern in these lines. In this manner, a fine-grained parametrisation of the objects in the decentralised flow path management system may be accomplished by creating an adequate component parameter file which is both simple and expressive at the same time.
8 Case Studies

The purpose of this chapter is to present the experiences and results obtained while applying the prototypical implementation of the product flow path management technology from Chapter 7 to real-life scenarios. For this, two very different plants were studied: a large petrochemical plant, and a model pumping station at the Chair of Process Control Engineering of RWTH Aachen University.

8.1 Petrochemical Plant

As part of an industry project of the Chair of Process Control Engineering of RWTH Aachen University, a large petrochemical plant – concretely an oil refinery – was studied with the purpose of developing a flexible plant asset management solution. In this plant, which is shown in Figure 8.1, many different pumps are in operation, and their physical states need to be monitored in order to allow for an early detection of the conditions that can cause equipment damage and breakdown. For this, the performance of the pumps can be monitored with the help of online process data. Nevertheless, flexible structures in the plant make it difficult to determine which data sources are of relevance for making this analysis, and a context-sensitive approach to plant asset management must be applied [36, 38]. The analysis of product flow paths in the plant can serve as a basis for this kind of application, and studying this approach was one of the goals of this industry project. Due to the confidentiality of the project, it is not possible to present its details in this work. Rather, a general description of the usage of the product flow path management system with this large-scale plant will be presented. In section 8.2, a more detailed description of the operation of the prototypical system will be given for the case of the pumping station.

8.1.1 Description of the Plant

The plant under consideration is a section of the oil refinery that serves as a tank farm for storing petrochemical products. The structure of this plant is flexible and allows pumping these products between the different tanks, as well as to and from product output and input connections to tanker vehicles. In total, this tank farm contains 21 pumps, 11 tanks, 226 valves, and more than 1500 piping sections used to connect these plant elements to each other.

8.1.2 Plant Model

The original plant model that was obtained for this plant was a P&ID. In order to have a formal model of the plant’s structure, a list of all plant elements and their interconnections was manually created, and a tool was developed which converted this list into a complete
8.1 Petrochemical Plant

Figure 8.1: Aerial view of the petrochemical plant.

RIVA model of the plant. When this RIVA model is loaded into an iFBSpro function block server, it measures a total of 43.8 MB. After this, the model can be accessed by ACPLT/KS applications.

8.1.3 System Synthesis

Based on the RIVA model of the plant, the system synthesis program described in Chapter 7 may be used to automatically create a flow path management system for the tank farm. When this is done, a total of 23605 objects and 43388 object links are created, and 61353 variables are set. The entire operation requires approximately 20 minutes, and the resulting system measures 9.3 MB in size, which is about 21% of the original model’s size. Although the mapping from plant elements to decentralised components is one-to-one, the RIVA model additionally contains objects that describe the thermodynamical properties of the plant elements, and these objects have no correspondents in the flow path management system. This explains the smaller size of this system with respect to the original RIVA model.

In order to configure the flow allowance settings of the various plant elements, a component parameter file was employed. In this file, the types of the various plant elements were used to determine the configuration of each corresponding component. For instance, the configuration of a two-position valve was encoded as follows:

```
Ventil_auf_zu /FlowAllowance Settings {CLOSED OPEN}
Ventil_auf_zu /IF_Con1Ch1/FlowAllowance InFlowSettings {FALSE TRUE}
Ventil_auf_zu /IF_Con1Ch1/FlowAllowance OutFlowSettings {FALSE TRUE}
Ventil_auf_zu /IF_Con2Ch1/FlowAllowance InFlowSettings {FALSE TRUE}
Ventil_auf_zu /IF_Con2Ch1/FlowAllowance OutFlowSettings {FALSE TRUE}
```
Here, the type **Ventil_auf_zu** is associated to this kind of valve, and the two connectors of the valve are respectively designated by **IF_Con1Ch1** and **IF_Con2Ch1**. The component is configured to use two flow allowance settings, namely **CLOSED** and **OPEN**, and the input and output flow allowance for both connectors of each valve are also configured in a corresponding manner. Descriptions similar to this one were used to configure all components of the generated flow path management system.

### 8.1.4 Operation of the Flow Path Management System

Once the flow path management system has been created by the synthesis program, it may be activated and used to discover and monitor flow routes in the plant. Due to the large size of the plant, the analysis of product flow paths can be limited to open flow paths, which reduces the load of the system and yields current flow path information that can be used for the purpose of context-sensitive plant asset management. Likewise, the monitoring of product flow paths may be used to determine a safety status for the active flow paths in the plant. This information may be then collected for its presentation to the users of the system in the form of an integrated graphical human-machine interface.

### 8.2 Pumping Station

The pumping station at the Chair of Process Control Engineering is a scaled-down industrial plant which is used for hands-on teaching about process control engineering to undergraduate students. This plant is shown in Figure 8.2. Although small in size, the plant features state-of-the-art plant devices, control systems, operator and engineering stations and communication infrastructure which collectively reflect the technological environment found in modern industrial settings. Therefore, it serves well for the purpose of testing the application of new technologies, such as the one presented in this work, to real plants. The present section describes in detail the use of the prototypical flow path management system with this plant.

#### 8.2.1 Description of the Plant

The pumping station models a water treatment plant. However, instead of handling contaminated water, the plant uses clean water only, and the level of purity of the water is simulated by means of the water's temperature: the warmer the water is, the greater its contamination level. For this, a manually-controlled heater is used to vary the temperature of the water. The purpose of the plant's operation is to determine if the water which flows from a source tank is either clean (cold) or dirty (warm), and to route it to a different target tank in each case. The water from both of these tanks is later pumped back to the source tank, allowing the water to be reused and thus avoiding the continuous consumption of water by the plant.

Figure 8.3 shows a P&ID of the pumping station. The tank B1 is used as the source tank, and water may be pumped out of this tank by means of two different pumps: P13 and P18. The heater W43 may be used to heat the water which is pumped by P13 only; the water pumped by P18 cannot be heated. The water coming from these two pumps flows through the corresponding control valves V16 and V21 and later merges into a single piping section.
At this point, the temperature sensor T23 measures the temperature of the water, and this information is used by the control logic to open the valve V24 if the water is colder than a given limiting temperature value, or to open the valve V25 otherwise. This causes the cold water to flow into the tank B3 and the warm water to flow into the tank B2, which is the main purpose of the plant's operation. In order to keep the tanks B2 and B3 from overflowing, a recirculating operation is started when either of these tanks is full, which pumps water from the tank that is full back to the source tank B1. For this, a single pump is used in the case of B2, and a plant section featuring redundant pumps is available in the case of B3.

The pumping station is subdivided into three plant sections, with the purpose of exemplifying the structural organisation employed in the design of modern processing plants. The section TU10 contains the source tank and the piping sections leading to the two target tanks; the section TU20 includes the tank B2 and the piping section that allows recirculation from this tank back to the source tank B1; finally, the section TU30 includes the tank B3 and the piping section that is used to pump water from this tank back to the tank B1.

8.2.2 Plant Model

Plant Structure Model

A model of the pumping station was originally available in CAEX format. By applying an XML transformation to this input file, a RIVA model of the pumping station was obtained, which was then loaded into an iFBSpro server named fb_pumpwerk_riva as shown in Figure 8.4. The amount of memory used by this instantiated model is 434 KB. As explained
Figure 8.3: Piping and instrumentation diagram of the pumping station.
8.2 Pumping Station

Figure 8.4: Instantiated RIVA model and flow path management system for the pumping station.

In Chapter 6, this model is an object-oriented representation of the structure of the plant, which features all plant elements and process control points, as well as the interconnections between them. The object hierarchy in this model follows the sectioning structure of the plant, where the three plant sections are represented by corresponding object containers. These containers are then used to hold the objects that represent each of the plant elements contained in the corresponding plant section. The identifiers used for these objects are of the form `<SECTION>/<NAME>`, in order to clearly identify the plant element and plant section of every object.

Flow Allocation Model

The flow allocation model of the pumping station was defined by means of a component parameter file as described in Chapter 7, and the contents of this file is shown in the following:

```
/TU*/U_[V]* /FlowAllocation Settings {CLOSED OPEN}
/TU*/U_[V]* /C_N1/FlowAllocation InFlowSettings {FALSE TRUE}
/TU*/U_[V]* /C_N1/FlowAllocation OutFlowSettings {FALSE TRUE}
/TU*/U_[V]* /C_N2/FlowAllocation InFlowSettings {FALSE TRUE}
/TU*/U_[V]* /C_N2/FlowAllocation OutFlowSettings {FALSE TRUE}
/TU*/U_[B]* /FlowAllocation Sink TRUE
/TU*/U_[B]* /FlowAllocation Source TRUE
/TU*/U_[BKRPWZ]* /FlowAllocation Settings {CONST}
```
8.2.3 System Synthesis

As in the case of the petrochemical plant, the synthesis program was run with the corresponding RIVA model and parameter file as input, and a flow path management system was instantiated in an iFBSpro server named \texttt{fb\_pumpwerk\_fpm} as shown in Figure 8.4. In this case, a total of 643 objects and 1162 object links were created in this server, and 1646 variables were set, in a process that took around 35 seconds. The generated system was 343 KB in size, which is approximately 79% of the size of the original model.
8.2.4 Access to Online Plant Data

The access to online plant data, which allows the flow path management system to determine the current flow allowance state of the plant, was achieved with the help of a Siemens PCS 7 process control system as shown in Figure 8.5. Firstly, the PCS 7 system was used to control the operation of the plant, and an accompanying PCS 7 OCS server made the online plant data available to ACPLT/KS clients. Then, by using function blocks from the fbkslib in the fb_pumpwerk_fpm server, it was possible to dynamically obtain information about the state of the elements in the plant during its operation. Figure 8.6 shows the instance of the KsGetPkg function block used by the flow path management system to retrieve the information from the PCS 7 OCS server in a periodic manner.

For this test, the online data of the valves in the plant was retrieved, and all other plant elements were considered to have a constant flow allowance state. For this, an instance of the KsGet class was created for every valve in the plant, and was configured to read the check-back-closed signal of the valve, which is an acknowledgement signal that comes from the plant device that confirms that it is in a closed position. This signal corresponds to a Boolean value, which must be converted to the integer flow allowance setting value that the component flow allowance object expects at its SettingSense input port. This was achieved with the help of a not function block, which implements the logical not (¬) operator, and a BOOL2UINT function block that converts the Boolean value to an unsigned integer. In this way, the value false corresponding to the input signal is mapped to the value 1, and consequently, the signal value true is mapped to the value 0, thus achieving a translation of the input signal to an indication of the current flow allowance state of the
Figure 8.6: Connection to the PCS 7 OCS server.

plant element. Figure 8.7 depicts the faceplate of the flow allowance function block for the valve V16 while the valve is open. The converted online value is 1, which causes the function block to report the current flow allowance setting 1 which bears the name “OPEN” and which allows incoming and outgoing flow through both connectors of the valve, as specified in the component parameter file that was shown.

The conversion strategy that has been explained worked for all valves except for V37, which does not offer a check-back-closed signal; instead, this valve offers a real value that indicates its position, and which ranges between 0 and 100. For this valve, a \texttt{gt} function block was used, which implements the \textit{greater than} (\textgreater) operator, in order to determine if the value of the input signal was greater than a given limiting value. The use of the value 0 as a limiting value proved to be useful in this case, but in practise, control valves such as this one might never report an actual value of 0 as its position, which means that more adequate limiting values should be used. After having determined if the valve is closed in this manner, a \texttt{BOOL2UINT} function block was again used to convert the Boolean value to an integer value that represents the current flow allowance setting of the plant element.

Because the number of valves in this plant is relatively small, the control logic for the retrieval and conversion of online plant data was manually designed. Nevertheless, the use of automated procedures for laying down the online data retrieval and conversion components is certainly possible. This approach would need to consider the technique which is to be used for the retrieval of data, as well as the type and meaning of the input signal, and the flow allowance model of the corresponding plant elements, in order to create the adequate mappings between these values.
8.2 Pumping Station

Once the product flow path management system for the pumping station was properly constructed and activated, and the connection to the online plant data was established, it was possible to conduct some tests on the operation of the system. A sequence of relevant tests was carried out, and their results were documented and are explained in the following.

**Flow Route Discovery**

The first test consists of the discovery of those flow routes which begin at tank B1 and which end at tank B2, and this is shown in Figure 8.8. The Origins and Targets variables of the flow path manager object of the system were configured in order to perform this flow path search, and the search was triggered by enabling the Discover condition. In this case, the execution of the flow path analysis algorithm yields two flow routes – one through the pump P18 and the other through the pump P13 – and the flow path manager object creates the corresponding flow route objects as shown in Figure 8.8.

In the tests that follow, the second of the flow routes from tank B1 to tank B2 was used, that is, the one which passes through the pump P13. Figure 8.9 shows the faceplate of its corresponding function block, and Figure 8.10 shows a portion of the value of the variable Elements of this flow route object, which contains the complete element sequence of the flow route in the plant.
Figure 8.8: Discovery of flow routes from tank B1 to tank B2.

Figure 8.9: Faceplate of the second flow route from tank B1 to tank B2.
Flow Path Creation

In order to create a flow path object for the second flow route from tank B1 to tank B2, the input condition Select of the flow route object was enabled. This triggers the creation of the flow path function block that is pictured in Figure 8.11. Simultaneously, the flow path logger object that uses the OV log file service for documenting the life cycle of product flow paths emits the following report to the log:

```plaintext
[IFBS Info] Flow path: /TechUnits/FlowPathManager/
FlowRoute_0000000015_2916948391/FlowPath_12581242830000849967
[IFBS Info] Description: From /TU10/U_B1 to /TU20/U_B2
[IFBS Info] Origin: /TU10/U_B1
[IFBS Info] Target: /TU20/U_B2
[IFBS Info] Length: 15
[IFBS Info] Hash: 2916948391
[IFBS Info] Element 0000000000: /TU10/U_B1
[IFBS Info] Element 0000000001: /TU10/U_R2
[IFBS Info] Element 0000000003: /TU10/U_R3
[IFBS Info] Element 0000000004: /TU10/U_W1
[IFBS Info] Element 0000000005: /TU10/U_R4
[IFBS Info] Element 0000000006: /TU10/U_V16
[IFBS Info] Element 0000000007: /TU10/U_R5
[IFBS Info] Element 0000000008: /TU10/U_Z2
[IFBS Info] Element 0000000009: /TU10/U_R9
[IFBS Info] Element 0000000010: /TU10/U_Z3
[IFBS Info] Element 0000000011: /TU10/U_R12
[IFBS Info] Allocate: false
[IFBS Info] Lock: false
```
This report contains the information about the corresponding flow route and the entire state of the flow path object at its creation. Reports similar to this one are also issued at every state change of the flow path object, and allow a detailed tracking of the life cycle of the product flow path.

**Flow Path Monitoring**

The monitoring of the product flow path created in the previous section was activated by enabling the Monitor input condition of the flow path object. The Monitoring output condition is then enabled by the object, and the Open output condition starts indicating the open...
or closed status of the corresponding flow route in the plant. Also, the OpenTimestamp variable is continuously modified to reflect the time of the last update.

Because all valves of the pumping station were initially closed, the flow path was also shown as closed. Only after opening both of the valves V16 and V25 – the two valves along the corresponding flow route – does the Open condition show a value of true, as expected. Furthermore, if the valve V24 is open while the flow path is being monitored, a leak and a mixture alert is created by the flow path object as shown in Figure 8.12. Additionally, the AlertCount output variable indicates the value 2 for the two alerts that are issued, and the flow path logger object creates the following entries in the flow path management system’s log:

```
[IFBS Info] Flow path: /TechUnits/FlowPathManager/
FlowRoute_0000000015_2916948391/FlowPath_12581242830000849967
[IFBS Info] Alert type: LEAK
[IFBS Info] Element 0000000000: /TU10/U_Z3
[IFBS Info] Element 0000000001: /TU10/U_R10
[IFBS Info] Element 0000000002: /TU10/U_V24
[IFBS Info] Element 0000000004: /TU30/U_B3
[IFBS Info] FlowPathLog: End Alert

[IFBS Info] Flow path: /TechUnits/FlowPathManager/
FlowRoute_0000000015_2916948391/FlowPath_12581242830000849967
[IFBS Info] Alert type: MIXTURE
[IFBS Info] Element 0000000000: /TU10/U_Z3
[IFBS Info] Element 0000000001: /TU10/U_R10
[IFBS Info] Element 0000000002: /TU10/U_V24
[IFBS Info] Element 0000000004: /TU30/U_B3
[IFBS Info] FlowPathLog: End Alert
```

A log entry is created for each one of the alerts, which contains the branch flow route that stems from the flow route of the corresponding flow path object and which leads to (respectively comes from) a product sink (respectively product source) in the plant.

After closing valve V24, the alert count reported by the flow path object changes to 0, as the alert messages are no longer being received. However, the alert objects shown in Figure 8.12 remain in the system, and after being acknowledged, can be manually deleted.

Flow Path Allocation

For testing the allocation of flow paths in the pumping station, the case of recirculating water back to the tank B1 from the tanks B2 and B3 was considered. For this, the discovery of flow routes was activated once more, this time by searching for flow routes using two flow route specifications simultaneously: from the tank B2 to the tank B1, and from the tank B3 to the tank B1. This operation causes the creation of three flow route objects, which correspond to the three flow routes that respectively pass through the pumps P29, P36 and P38, as may be observed in Figure 8.3.
After creating a flow path object by selecting the flow route which starts at B2, passes through P29 and ends at B1, the flow path allocation operation was started for this object by setting the Allocate input of this function block to a value of true. The successfully allocated flow path object is shown in Figure 8.13, where the function block’s output variables indicate the allocated state of the flow path as well as the set of enclosure constraints associated to the allocation. In this case, valves V37 and V39 receive an enclosure constraint that enforces their closed flow allowance settings, and the reason for this may be seen clearly in Figure 8.3: closing these valves inhibits the flow of water to and from the corresponding flow route to the tank B3. Although part of this is already guaranteed by the holding valves next to the pumps P36 and P38, the messages of the flow path allocation algorithm reach the valves V37 and V39 before these holding valves and the algorithm is then able to guarantee the enclosure of the flow route by constraining the flow allowance settings of these two valves.

With the intention of testing the allocation of conflicting flow paths, a flow path object was created for the flow route that begins at the tank B3, passes through the pump P36 and ends at the tank B1. While the flow path shown in Figure 8.13 was still allocated, the allocation of this new flow path was requested. As expected, an allocation conflict was detected and reported, as may be seen in Figure 8.14. In this case, a flow path alert object is created which indicates the conflict, and its Elements output variable provides the sequence of elements of the conflicting flow route. At the same time, the flow path logger object outputs the following report to the system’s log:

```
[IFBS Info] Flow path: /TechUnits/FlowPathManager/FlowRoute_0000000015_2876266582/FlowPath_12585560450000493741
```
In this manner, a clear indication of the conflict and its cause is provided by the flow path alert. The allocation is automatically retried as long as the Allocate input variable of the flow path object is enabled, while the time-stamp of flow path alert object that corresponds to the conflict is continuously updated to reflect the recurrence of the conflict. If the user disables the Allocate input variable, the allocation is cancelled and a deallocation of the already-allocated components is executed. Finally, after deallocating the flow route from tank B2 to tank B1, the allocation of the flow route from tank B3 to tank B1 may be executed successfully.

Figure 8.13: Allocated flow path from tank B2 to tank B1.
8.2.6 Distributed Flow Path Management System

A second variant of the flow path management system for the pumping station was tested, where the system was distributed across three function block servers. For this, the organisation of the pumping station in three plant sections was used as the criterion for the distribution of the components, and this was configured using the following server parameter file:

*/TU10/* localhost fb_pumpwerk_tu10_fpm /TechUnits
*/TU20/* localhost fb_pumpwerk_tu20_fpm /TechUnits
*/TU30/* localhost fb_pumpwerk_tu30_fpm /TechUnits

The system was synthesised with respect to this configuration file, and the resulting system is shown in Figure 8.15. Each server contains its own flow path manager and flow path logger instances, and additionally, makes use of KsGetPkg function blocks retrieving incoming messages from other servers, as was explained in Chapter 7. Apart from the usage of the server parameter file, the synthesis process for this system was practically identical to that of the single-server system.

8.3 Performance Analysis

In this section, the performance of the prototypical product flow path management system is analysed based on empirical evaluations. Concretely, a quantitative analysis of the runtime and space required by the system is presented. Both of the case studies in this chapter are considered, and for their corresponding systems, a series of tests were carried out in order
8.3 Performance Analysis

Figure 8.15: Distributed flow path management system for the pumping station.

to determine the required space and execution times of the different flow path management algorithms.

The machine that was used for the tests that are described in this section was a laptop computer with an Intel(R) Pentium(R) M processor running at 1400 MHz and with 768 MB of RAM. The iFBSpro Function Block System version 2.5 was used, as well as version 8.4 of the Tcl language and version 4.3.3 of the GCC compiler.

8.3.1 Space

The iFBSpro function block system uses a database with a fixed size, and the execution of the system need only maintain its memory requirements within these bounds. For the case of the petrochemical plant, the database that was used for the system was 100 MB in size, and this was enough to host the execution of the system. In turn, the flow path management system for the pumping station operated in a database of 10 MB. For these two cases, the size that was used for the database was approximately 10 times greater than the size of the inactive function block system. This rule of thumb was chosen for determining the size of the database because the overall space complexity of the system, as shown in Table 5.1, is linear with respect to the size of the plant and to the size of the longest flow route or branch flow route, and because the messages that are sent by the system are stored in the same memory area, and therefore, the messaging complexity – which can be exponential with respect to the size of the plant – must also be taken into account. For the case studies presented in this chapter, the chosen database sizes were effective for the operation of the flow path management system.
### Table 8.1: Execution times of a single iteration of the flow path management system.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Pumping station</th>
<th>Petrochemical plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>0.013 s</td>
<td>0.14 s</td>
</tr>
<tr>
<td>Discovery (open flow routes, one-shot)</td>
<td>–</td>
<td>0.15 s</td>
</tr>
<tr>
<td>Discovery (open flow routes, continuous)</td>
<td>–</td>
<td>2.5 s</td>
</tr>
<tr>
<td>Discovery (all flow routes, continuous)</td>
<td>0.043 s</td>
<td>–</td>
</tr>
<tr>
<td>Monitoring</td>
<td>0.023 s</td>
<td>1.9 s</td>
</tr>
<tr>
<td>Discovery and monitoring</td>
<td>0.067 s</td>
<td>–</td>
</tr>
<tr>
<td>Allocation</td>
<td>0.013 s</td>
<td>1.25 s</td>
</tr>
</tbody>
</table>

### 8.3.2 Execution Times

As explained in Chapter 7, the execution time of a single iteration of the components shall not exceed the configured cycle-time of the function block system. Therefore, this execution time is a lower bound for the cycle-time of the system, and also gives a lower bound for the execution time of the flow path management operations. In the case of the flow path management systems for the case studies presented in this chapter, these execution times were determined empirically and are provided in Table 8.1. The times shown in this table are the execution times reported by the iFBSpro function block system for a single iteration, and averaged across several consecutive operations of the system.

### Execution Times for the Pumping Station

In the case of the pumping station, the flow path operations were applied to all paths between any two tanks in the plant at once, with the exception of the allocation operation where a single flow path was allocated and deallocated. Thus, the flow paths specifications B1 to B2, B1 to B3, B2 to B1 and B3 to B1 were considered.

The execution time of the idle system, that is, when no operation is being performed, is indistinguishable from the execution time during an allocation or deallocation operation. This may be explained by the fact that during the execution of this algorithm, most components are waiting for incoming messages and only a few elements are processing them. In the case of the discovery of this set of flow paths, the execution time of the system was more than three times that of the idle system, because of the exponential increase in search messages due to branching structures found in the plant. For this operation, all flow routes were searched simultaneously and continuously, which means that a new search was started with every execution cycle of the system. The execution time of the simultaneous monitoring of all flow paths was greater than the execution time of the idle system, and about one half of the time required by the discovery operation. Finally, performing the discovery and monitoring of all flow paths simultaneously corresponds to the addition of the individual execution times of these operations, which is due to the fact that the different decentralised operations are orthogonal to each other.

It may be concluded from this benchmark that the prototypical flow path management system of the pumping station should safely handle cycle-times of 0.1 seconds, considering the fluctuation of the execution times. This means that the operations for these flow paths, whose corresponding flow routes are at most 15 elements in length, can be expected to complete in...
just a few seconds.

**Execution Times for the Petrochemical Plant**

In the case of the petrochemical plant, the flow path discovery operation was applied to the flow routes between two chosen tanks in the plant, and all other operations were applied to one of the flow routes found by this analysis, and which contains a total of 56 plant elements.

The execution time of the idle system was 0.14 seconds, which already suggests that for this system a cycle time of more that 0.1 seconds is required. The discovery of flow routes between the two chosen tanks was restricted to open flow routes only, and was first tried in a one-shot manner, which means that the search is initiated only once. A very slight increase in the execution time of the system was detected due to the search messages, but the execution time of this operation remains close to the execution time of the idle system. However, when the search is carried out in a continuous fashion, the execution time of the system grows almost 20 times and reaches 2.5 seconds, which suggests that for this operation, the configured cycle time of the system should not fall below 3 seconds. As already mentioned, the discovery of both open and closed flow routes was not tried for this plant. The flow path analysis operation found two flow routes for the given flow route specification, which respectively contain 56 and 72 elements. The shortest of these two routes was chosen and a flow path object was created for conducting the rest of the tests on this system. In the case of flow path monitoring, an execution time of 1.9 seconds was achieved, which is again lower than the time of a discovery operation. Finally, the allocation of the chosen flow path yielded an execution time which falls between the idle time of the system and the time required by the monitoring operation.

In general, the execution times of this system are between one and two orders of magnitude greater than the corresponding execution times of the system for the pumping station. The relation in sizes of these two plants accounts for this difference. By looking at these execution times, a configured cycle time of 3 seconds can be suggested for the use of this system.

**Analysis of the Execution Times**

The execution times displayed in Table 8.1 reflect the theoretical time complexities of the decentralised algorithms that were presented in Table 5.1. The greater time complexity and messaging complexity of the flow path analysis algorithm accounts for the highest individual execution times in both of the cases that were studied. Flow path monitoring and flow path allocation have equivalent time complexities with respect to the execution of the individual components, but the former has a greater messaging complexity, and this accounts for the difference in execution time that is exhibited between these two algorithms.

In general, the prototypical flow path management system performs well when applied to small and medium plants, and its performance degrades when applied to very large plants such as the petrochemical plant considered in this chapter. However, the implementation of this system offers room for improvements and for optimisations which suggests that the corresponding flow path management technology should still be applicable to large plants in practise. For instance, the messages that are sent between the various decentralised components are encoded as vectors of character strings, and its values must be copied by the system every time one of these messages is sent. This design decision was made in order to provide a simple message format that can be easily used within the function block system,
and which may be easily interpreted by the user during system diagnostics. However, an equivalent but faster implementation that uses a more efficient data format such as bit vectors, and which applies optimisation techniques such as caching of locally-sent messages, could offer a substantial improvement of the system's performance. All in all, the prototypical flow path management system that was presented in Chapter 7 can be regarded as a proof of concept of the flow path management technology of this work. In this respect, it has shown the viability of this technology by experimenting with the model-based synthesis approach, the decentralisation of the product flow path management operations and the usage of the product flow path object model.
9 Conclusions

The present work has introduced a new concept for conducting product flow operations in processing plants: the model of product flow paths. This chapter presents the closing remarks of this work. A summary of this work is given, followed by a discussion of its results within the context of process control engineering. Finally, this chapter provides an insight into future research directions that can stem from the study of the use of product flow paths in processing plants.

9.1 Summary

Processing plants are ubiquitous and fundamental at present. The automation of processing plants has permitted the execution of more complex processes, it has increased the efficiency and quality of the production, and it has supported the assurance of safety. However, as plants become more complex and more widespread, the engineering tasks required for designing and implementing control functionality become increasingly complex, costly, and harder to verify. The task of engineering the control of product transport operations in processing plants suffers from this problem as it is commonly accomplished manually, based on experience and common sense, and without the use of a model that guides the design. This incurs in greater engineering costs, and in a greater chance of introducing errors in the design.

The model of product flow paths is introduced as a novel attempt to provide a formal framework for the correct execution of product flow operations in processing plants. This model is centred around the idea of a product flow path, which is a software object that is responsible for controlling, monitoring and documenting the movement of products along a determined flow route in the plant, and which represents a temporarily and spatially isolated area for the safe and correct transport of products along this flow route. Product flow paths have a well-defined life cycle that characterises their operation and aims to guarantee their safe and correct use, inspired by the common procedures used by railway systems for achieving the safe movement of trains along the tracks of a railway network. The management of product flow paths comprises the automated tasks that are necessary for the adoption of the product flow path model within a process control system. It includes the discovery of flow routes in a plant, the creation and deletion of product flow path objects, the allocation, locking and activation of product flow paths, the monitoring of active product flow paths, the issuing of flow path alerts to report problematic situations and the documentation of the complete life cycle of every product flow path in the system. An object model for a product flow path management system has been defined, which provides a guide for the implementation of object-oriented product flow path management systems.

The automation of product flow path management tasks requires precise information about the possibility of product flow in the plant, which is provided by means of a formal model that represents the structure of the plant and the flow allowance of the elements in the plant.
Based on this model, a formal definition of the concept of a product flow route in a plant has been given, considering the possible displacement of material through the structure of the plant. Also, the properties of openness and enclosure of a flow route at a given flow allowance state of the plant have been formulated in a way which may be used by algorithms to detect potentially hazardous situations, such as product leaks and product mixtures, or to determine the absence of these situations for a given flow route. This formal framework conforms the base of an algorithmic solution for the automation of product flow path management tasks.

Modern process control systems operate in a decentralised manner because this offers many advantages over a fully centralised deployment of the control functionality. In accordance with this principle, this work has presented a decentralisation scheme that may be applied to the tasks involved in the management of product flow paths in a plant. Inspired by the decentralised operation of geographical railway control systems, a model for designing and implementing decentralised product flow path management systems has been developed by following a decentralised component-based approach, where the structure of the decentralised system follows the structure of the corresponding plant. The objects of a decentralised product flow path management system may be distributed across multiple controllers in a process control system, and in a way that follows the logical sectioning of the plant.

Based on the formal model of flow routes, and on the decentralised component-based approach that is followed in this work, algorithms have been presented which accomplish the tasks of a product flow path management system in a decentralised manner. The algorithms consist of procedures that execute within the different decentralised components of the system, which communicate with each other by exchanging messages that are delivered along the interconnections of the components. Flow path analysis is an algorithm that discovers flow routes in a plant by performing a decentralised breadth-first search over the graph of the plant. Flow path monitoring is an algorithm that determines the open condition of a flow route, which additionally detects and reports potential product leaks and product mixtures that affect the flow route that is being monitored. Flow path allocation is an algorithm that assigns plant elements to a product flow path in an exclusive manner, which also constrains the settings of some plant elements in order to guarantee the enclosure property of the flow path, and therefore its safe use. The objects of a product flow path management system make use of these algorithms in order to provide a unified access to the functionalities of discovering, monitoring and allocating product flow paths in a plant.

In order to provide a way of reducing the complexity and the cost of constructing a decentralised product flow path management system, and at the same time, a way of reducing the chance of introducing errors in the system, an automatic synthesis approach for this kind of systems has been developed, based on a formal model of the structure of the plant and on the flow allowance model of the elements in the plant. By accessing the information in these models, a synthesis algorithm automatically creates and parametrises the objects of a product flow path management system, which is then ready to begin its operation. This model-based synthesis technique is advantageous because it replaces the task of constructing a flow path management system for a plant – which is relatively complex and error-prone – with the simpler task of creating and verifying a model of the plant.

A prototypical implementation of the technology presented in this work has been developed using ACPLT technologies as a proof of concept. A decentralised product flow path management system is realised as a collection of iFBSpro function block servers. These servers host function blocks that correspond to the flow path objects, the decentralised components, the algorithm objects and the rest of the objects of this system. Furthermore, the decen-
centralised components communicate with each other and across the servers of the system, and this communication is accomplished using the ACPLT/KS protocol. The automatic synthesis approach described in this work is implemented for this prototype by means of a synthesis program that explores a RIVA model of the plant and based on the information contained therein, creates and parametrises the objects of a decentralised product flow path management system for the modelled plant. The interaction between the synthesis program and the iFBSpro servers that host the RIVA model and the target system is also accomplished by means of the ACPLT/KS protocol.

The prototypical product flow path management system was tested with the models of two real-life plants: a large petrochemical plant, and a model pumping station. For these plants, a flow path management system was automatically constructed based on a corresponding RIVA model of the plant’s structure, and on a description of the flow allowance model of the plant’s elements. The decentralised algorithms which perform the discovery, monitoring and allocation of product flow paths were tested with these systems, as well as the usage of the product flow path object model according to the life cycle of product flow paths that was presented in this work. In the case of the pumping station, the corresponding flow path management system was connected to a process control system of the plant in order to allow the decentralised algorithms to operate with online plant data.

The case studies that have been presented in this work have shown the feasibility of the product flow path management technology. Concretely, the model-based synthesis approach, the decentralisation of the product flow path management operations and the usage of the product flow path object model have all been applied in a successful manner in these cases. Because of this, this prototype can be seen as a reference implementation for industrial-strength flow path management systems.

9.2 Discussion

9.2.1 Methodology

The main objective that the present work has aimed to achieve is to provide a better way of performing product flow operations in processing plants, when compared to the current engineering practises. The general method that has been applied for reaching this objective consists of three activities, namely, to analyse, to model, and to design. Although it may be argued that these activities are inherent to all engineering fields, there are many problems that are handled by engineers in less structured ways within practical settings. In many of these cases, there are pragmatic reasons that prevent the use of “theories” and “full automatisms” for solving day-to-day problems, but in other cases, there are real opportunities for improvements along the lines of cost-reduction and quality assurance of the engineering labour, as well as for a general optimisation of this activity. The concrete task of executing product transport operations in processing plants represents an example of this circumstance.

The analysis that has initiated this work has been the task of reasoning about the flow of products in a plant. For instance, answering questions such as: Why does material flow? How should material flow? Which situations are undesired when material is flowing, how may they be characterised, and how can they be prevented? What information is needed in order to answer these questions in the setting of a real plant? These questions have been answered in this work, to a certain degree. Finding answers to these questions which are universally
valid is likely to be an impossible task. Nevertheless, useful knowledge can be obtained when limiting the scope of the problem, as well as generalising the answers so that they are probably correct in most of the settings that are considered. Also, further refinements in the definition of the scope of the problem, and in the generalisation of the answers, should lead to more precise knowledge. This is certainly true with regard to the present work, which has deliberately kept the scope of the problem within the realm of piping systems and generalised the characterisation of plant devices.

The analysis phase that has been described was succeeded by a modelling phase, which has striven to develop an adequate representation of the plant knowledge that has been determined to be of importance for solving the engineering problem at hand. The product of this activity is a meta-model for processing plants, which may be instantiated in order to represent the required information about a concrete plant. As stated in [5], “a model is a simplification of reality”, and it is crucial that this be so, because a model permits the definition of problems and their solutions in a way that would be difficult or impossible to accomplish when regarding the real system alone. However, the right degree of simplification must be obtained, so that the problems and solutions that were formulated in the model are still valid, to an acceptable degree, when applied to the real system. Or as Albert Einstein said, “make everything as simple as possible, but not simpler”. Because of this, the plant model that has been presented and used in this work contains the fundamental information about the plant that is needed for implementing a product flow path management system: structure of the plant in terms of plant elements and their interconnections, basic characterisation of the role of plant elements with respect to the flow of products, and a detailed description of the possibility of product flow through every element in the plant, and at every state of the plant. Just as the analysis phase allows refinements with the purpose of obtaining more precise knowledge about the flow of products in a plant, the plant meta-model that has been presented can be refined and extended to include more information that may be useful or necessary in given contexts. The basic form of the model should remain the same, nonetheless, because it represents the fundamental information that is required for implementing any kind of automatic solution for the tasks of a product flow path management system.

The design phase of this work has corresponded to the development of algorithms and systems that, based on the plant meta-model, are capable of solving the required tasks related to product flow paths in practise. Many different elements have played a part in the design of the technology that has been presented in this work. Firstly, an object-oriented design methodology has been adopted and followed with the purpose of specifying the software technology in a clear and understandable manner, so that it may be easily incorporated into modern control systems. When looking for practical guidelines to be followed in the solution to the problem, the results and current practises in the field of railway engineering were found to be a good reference model because of the similarities between the problem settings in both fields. An adequate mapping of the concepts between both fields effectively permitted the reuse of knowledge across subject boundaries, which can be regarded as very valuable given the years of accumulated experience in the field of railway engineering. Also, the importance of decentralisation in the design of process control systems has been a strong motivation to follow a decentralised approach in this work. The algorithms that have been formulated as a solution for the various tasks of a product flow path management system have been designed as inherently decentralised algorithms, and the object model of product flow paths has provided a scheme for distributing its objects across multiple decentralised controllers. Hopefully, the features of this design will benefit the adoption of the product flow
path model within decentralised process control systems.

The way in which the design of the decentralised product flow path management system has followed the model of the structure of the plant, has permitted not only the simple formulation of a decentralisation scheme for flow path management algorithms, but also the development of a simple technique for the automatic synthesis of a product flow path management system based on this model. This synthesis technique replaces the task of manually instantiating and parametrising the objects of the corresponding system, and is therefore a highly effective procedure for applying the ideas of the \textit{automation of automation} to this system. The prototypical implementation of this technique is a proof of the feasibility of automating some engineering processes related to process control systems, with the goal of reducing the engineering effort together with its associated costs and the chance of manually introduced errors.

The methodology of analysing, modelling and designing that has been followed in this work represents a valid option for developing model-based, automatic solutions to engineering problems in the general sense. Furthermore, the models that are needed for solving different problems need not be very different from each other, if they model the same real-life object. In this work, the RIVA model has been used not only as a conceptual basis for the abstract plant model, but also as a fundamental technology for the development of the prototypical implementation. A RIVA model represents the constitution of the plant at many levels, and has been successfully used in other projects for different purposes such as the automatic determination of mass and energy balances in processing plants [29]. This suggests that the fundamental idea behind a successful application of automatic solutions to process control engineering problems is the need for complete and accurate plant models, which are designed and validated by experts, and which are available to the control system during its operation. Only in this manner is the system able to apply automatic model-based techniques which consist of algorithms that replace certain engineering tasks based on the knowledge about the plant that is represented in the plant’s model.

\subsection*{9.2.2 Application Domain}

The concept of product flow paths presents a general approach for performing flow operations in a plant in a correct and safe manner, which applies to plants that execute both continuous and batch processes. However, because of the very nature of batch processes, their corresponding plants are likely to benefit from embracing a product flow path management system in a special way. The operation of batch plants incurs in frequent switching of product flow operations, which is an ideal environment for a product flow path management system. The usage of a flow path to move material may be preceded by a search for a proper flow route at the current state in the operation of the plant, and this flow route need not have been defined or configured in advance. This allows a very flexible utilisation of the plant, which is highly regarded in batch plants. Plants that execute continuous processes do not exhibit this transient behaviour, but can nevertheless benefit from the services of a product flow path management system, specially with regard to the monitoring functionality of product flow paths.
9.2.3 Related Work

Regarding the design of the control of product transport operations in processing plants, the manual engineering of the corresponding control logic can be considered the state of the art. When compared to this technique, the technology presented in this work offers many advantages such as greater flexibility and a reduction of the engineering effort, and in general constitutes a more convenient way of performing flow operations.

The Route Control package for SIMATIC PCS 7 has similar goals to those of the technology presented in this work, and coincides with this work in some of the techniques used, most notably, in the use of a search algorithm for discovering flow routes in the plant. Nevertheless, the approaches taken by both technologies are fundamentally different. The product flow path management system presented in this work is based on a formal model of the plant which represents the structure and behaviour of the plant with respect to the flow of material, and it uses this model not only to define the structure of the system, but also to define the notions of flow routes, leaks, and mixtures, and to define the algorithms used by the system. Therefore, it follows a model-based approach, where the model determines the structure and the operation of the system. In contrast, the Route Control package presents the user with an engineering environment that is used to design a flow-route-oriented control logic, and offers a flexible way to control flow routes during operation, but lacks a plant model in the actual sense of the term. This may be seen in the fact that the engineer must define every partial route in the system, in a process that defines the topology of the plant as an overlay structure within the system’s object model. It may also be seen in the fact that, to every partial flow route that is defined, a traditional engineering approach must be applied. It is clear that the system has been designed in this way for practical reasons, and that the technology presented in this work stems from a research perspective, but the distinction of these two approaches is important because it suggests that a greater degree of automation of the engineering tasks may be achieved by following a model-based approach, and therefore, that products such as Route Control may benefit from embracing this form of technology.

The information about active product flow paths in a plant may be additionally useful for purposes other than conducting flow operations. In [38], an approach is presented that uses this information for achieving plant asset management goals. A pump that is operating in a plant may be diagnosed with the help of the measurement of the material’s pressure at both sides of the pump. However, plants do not always have pressure sensors placed near their pumps, which makes it difficult to apply this technique. If the active flow path that is used by the plant is known, this information can be used to locate pressure sensors that are far from the pump, but which measure precisely the pressure that is needed for performing the diagnosis of the pump. Therefore, during the time that the flow path is active, the diagnosis can take place. This provides a general approach for performing context-related diagnosis of plant elements, based on the information about the activity of flow paths in a plant.

9.3 Future Research Directions

The present work has introduced a new technological concept and approach, and because of its novel nature, it provides room for many extensions and refinements. Some of these are outlined as follows.

- The product flow path management tasks of discovery, monitoring and allocation of
product flow paths have been implemented by means of model-based, decentralised algorithms. However, the tasks of locking and activating product flow paths have been assumed to be implemented externally, and an adequate interface to this functionality has been provided. Solving these tasks using a decentralised, model-based technique analogous to the one outlined in this work is an important direction of research. This would require determining which information about the plant and its control system is necessary for implementing the model-based interlocking and actuation on the plant elements that are involved in a flow path, and developing decentralised algorithms that realise these tasks based on the information contained in an enriched plant model. The information that is provided by the flow path analysis and flow path allocation algorithms is intended to be useful for implementing these new algorithms.

- The integration of a product flow path management system within a batch process control system is an interesting research opportunity. A batch process control system must coordinate many process control tasks, included the transport of materials in the plant, and may therefore use the services provided by product flow path objects. Also, batch systems perform the tasks of scheduling operations and allocating plant resources for these operations. Here, flow paths may be seen as high-level resources that could be directly regarded by batch systems, which may be of help in optimising the use of the plant. Also, the operation of a plant based on product flow paths may help to accomplish a detailed tracking of the products in the plant. The documentation of the life cycle of product flow paths should be helpful for tracing the flow of products in the plant along arbitrarily long periods of operation.

- The product flow routes that have been considered in this work consist of a simple sequential structure. Although this simple definition of a product flow route is general enough to be applicable to most scenarios where product flows within plants, there exist applications that could benefit from directly handling product flow paths with more complex structures that exhibit branching and cycles. This extended representation of product flow routes could be accomplished by composing simple flow routes, and thus reusing the results of this work. Further research in this direction may prove to be beneficial.

- The Route Control package offers several features that are not a part of the technology presented in this work, which appear to be useful in practise. These include the explicit management of the information about the kinds of materials that flow through the plant, as well as their relationships – in the form of valid successor sequences – which allow the system to assert the correctness of the sequential use of the flow routes in the plant and to enforce required cleaning operations by incorporating the types of materials in the state information of the flow routes. This functionality could be included as an additional aspect of the flow path model presented in this work. Also, Route Control assigns numerical priorities to the partial flow routes in the plant in order to help the system in selecting flow routes for their use in an automatic manner. With regard to this topic, the approach that this work has taken consists of letting the user of the system – the operator of the plant – decide which flow route to use at a given time. The idea of having the system perform an automatic selection of flow routes may be researched not only in the direction of priorities, but by additionally considering other criteria such as material type, characteristics of the plant elements, optimisation of plant usage, and
even power consumption. For this, a model-based technique may again prove to be the most adequate approach.

- The monitoring of product flow paths can be enriched to retrieve more information about an active flow route. For instance, information that can be obtained from sensors, such as the flow rate, the temperature and the pressure of the material that flows along a flow route, may be collected by the decentralised product flow path monitoring algorithm and reported to the corresponding flow path object. The RIVA model of a plant includes a representation of the sensors that are attached to the various plant elements, and this information can be used to support a model-based technique for monitoring sensor values along a flow route.

- The operator stations of process control systems usually offer graphical user interfaces that superpose plant diagrams with live sensor values and activity indicators. The model of product flow paths presented in this work may inspire new forms of visualisation and monitoring of plant operations that are centred around the concept of product flow paths, such as the one presented in [27].

- Finally, the ultimate goal of this work is to encourage the adoption of the flow path model within process control systems. The way in which this can be accomplished in a practical manner is a research topic in its own right, because of the many aspects that surround the design and use of process control systems: technological, practical, economical, environmental and even legislative. Nevertheless, the concept of product flow paths and the associated technology of product flow path management have been designed with regard to current process control systems, and therefore, the adoption of this novel approach on the part of the vendors of process control systems should prove to be a viable and sensible undertaking.
A ACPLT/OV Models

A.1 Decentralisation.ovm

/*
 * Decentralisation Library
 * Gustavo Quiros
 * Chair of Process Control Engineering, RWTH Aachen University
 * g.quiros@plt.rwth-aachen.de
 */

/* This library provides the base for decentralised, component-based
 systems. It is meant to be used as an algorithmic layer over
 structured systems as modelled by the 'gss' library from RIVA. */

#include "gss.ovm"

LIBRARY Decentralisation

AUTHOR = "Gustavo Quiros";
COMMENT =
"Function block library for decentralised, component-based systems."

CLASS Component : CLASS gss/FBSystem

IS_INSTANTIABLE;
IS_FINAL;

COMMENT = "A component or node of a decentralised system."

VARIABLES

 Type : STRING FLAGS = "I"
 COMMENT = "The type of this component."

 Identifier : STRING FLAGS = "I"
 COMMENT = "The identifier of this component."

END_VARIABLES;

OPERATIONS

 startup : C_FUNCTION <OV_FNC_STARTUP>;
 shutdown : C_FUNCTION <OV_FNC_SHUTDOWN>;
 typemethod : C_FUNCTION <FB_FNC_TYPEMETHOD>;
Reset : C_FUNCTION <FNC_Decentralisation_Component_Reset>;
END_OPERATIONS;
END_CLASS;

CLASS Connector : CLASS gss/FBInterface

IS_INSTANTIABLE;
IS_FINAL;

COMMENT =  
"A connection point for bidirectional communication between components."

VARIABLES

InMsg[] : STRING FLAGS = "I"
COMMENT = "Incoming message port."

OutMsg[] : STRING FLAGS = "O"
COMMENT = "Outgoing message port."

InCount : UINT FLAGS = "O"
COMMENT = "Number of processed incoming messages."

OutCount : UINT FLAGS = "O"
COMMENT = "Number of processed outgoing messages."

END_VARIABLES;

OPERATIONS

constructor : C_FUNCTION <OV_FNC_CONSTRUCTOR>;
typemethod : C_FUNCTION <FB_FNC_TYPEMETHOD>;

GetIncomingSenderIdentifier : C_FUNCTION
<FNC_Decentralisation_Connector_GetIncomingSenderIdentifier>;

GetNextIncomingMessage : C_FUNCTION
<FNC_Decentralisation_Connector_GetNextIncomingMessage>;

GetIncomingMessageType : C_FUNCTION
<FNC_Decentralisation_Connector_GetIncomingMessageType>;

GetIncomingMessageLength : C_FUNCTION
<FNC_Decentralisation_Connector_GetIncomingMessageLength>;

GetIncomingMessageItem : C_FUNCTION
<FNC_Decentralisation_Connector_GetIncomingMessageItem>;

InitialiseOutgoingMessagePort : C_FUNCTION
<FNC_Decentralisation_Connector_InitialiseOutgoingMessagePort>;

CreateOutgoingMessage : C_FUNCTION
A.1 Decentralisation.ovm

```plaintext
A.1 Decentralisation.ovm

CLASS Algorithm : CLASS fb/functionblock

COMMENT = "Base class for decentralised algorithms.";

OPERATIONS

BeginExecution : C_FUNCTION
<FNC_Decentralisation_Algorithm_BeginExecution>
IS_ABSTRACT;

HandleIncomingMessage : C_FUNCTION
<FNC_Decentralisation_Algorithm_HandleIncomingMessage>
IS_ABSTRACT;

EndExecution : C_FUNCTION
<FNC_Decentralisation_Algorithm_EndExecution>
IS_ABSTRACT;

Reset : C_FUNCTION
<FNC_Decentralisation_Algorithm_Reset>
IS_ABSTRACT;

END_OPERATIONS;

END_CLASS;

ASSOCIATION ConnectorContainment : ONE_TO_MANY

IS_LOCAL;

PARENT Component : CLASS Decentralisation/Component
COMMENT = "The component that contains the connectors."

CHILD Connectors : CLASS Decentralisation/Connector
COMMENT = "The connectors contained by the component."

END_ASSOCIATION;

ASSOCIATION AlgorithmContainment : ONE_TO_MANY

IS_LOCAL;

PARENT Component : CLASS Decentralisation/Component
COMMENT = "The component that contains the algorithms."

END_ASSOCIATION;
```

187
A.2 FlowAllowance.ovm

/ * 
* Flow Allowance Library 
* 
* Gustavo Quiros 
* Chair of Process Control Engineering, RWTH Aachen University 
* g.quiros@plt.rwth-aachen.de 
*/

#include "Decentralisation.ovm"

LIBRARY FlowAllowance

AUTHOR = "Gustavo Quiros";
COMMENT = "Flow allowance model for decentralised component-based systems."

CLASS ComponentFlowAllowance : CLASS fb/functionblock

IS_INSTANTIABLE;

COMMENT = "Flow allowance model for a component.";

VARIABLES

Settings[] : STRING FLAGS = "I"
COMMENT = "The names of the flow allowance settings of the component."

Source : BOOL FLAGS = "I"
COMMENT = "Indicates if the component represents a source of flow."

Sink : BOOL FLAGS = "I"
COMMENT = "Indicates if the component represents a sink of flow."

SettingSense : UINT FLAGS = "I"
COMMENT = "Input indicator for the currently active setting."

SettingCount : UINT FLAGS = "O"
COMMENT = "The number of settings of this component."

Setting : UINT FLAGS = "O"
COMMENT = "The currently active setting."

SettingName : STRING FLAGS = "O"
COMMENT = "The name of the currently active setting."

END_VARIABLES;

OPERATIONS

typemethod : C_FUNCTION <FB_FNC_TYPEMETHOD>;

END_OPERATIONS;

END_CLASS;

CLASS ConnectorFlowAllowance : CLASS fb/functionblock
IS_INSTANTIABLE;

COMMENT = "Flow allowance model for a connector."

VARIABLES

InFlowSettings[] : BOOL FLAGS = "I"
COMMENT = "The input flow allowance settings of the connector."

OutFlowSettings[] : BOOL FLAGS = "I"
COMMENT = "The output flow allowance settings of the connector."

InFlow : BOOL FLAGS = "O"
COMMENT = "The current input flow allowance of the connector"

OutFlow : BOOL FLAGS = "O"
COMMENT = "The current output flow allowance of the connector"

END_VARIABLES;

OPERATIONS

typemethod : C_FUNCTION <FB_FNC_TYPEMETHOD>;

ExistsInFlow : C_FUNCTION
<FNC_FlowAllowance_ConnectorFlowAllowance_ExistsInFlow>;

ExistsOutFlow : C_FUNCTION
<FNC_FlowAllowance_ConnectorFlowAllowance_ExistsOutFlow>;

ExistsThroughFlow : C_FUNCTION
<FNC_FlowAllowance_ConnectorFlowAllowance_ExistsThroughFlow>;

END_OPERATIONS;

END_CLASS;

ASSOCIATION ComponentFlowAllowanceMapping : ONE_TO_ONE
A.3 FlowPathAnalysis.ovm

/*
 * Flow Path Analysis Library
 * Gustavo Quiros
 * Chair of Process Control Engineering, RWTH Aachen University
 * g.quiros@plt.rwth-aachen.de
 * */

/* This library provides the implementation of the decentralised flow path analysis algorithm. */
#include "FlowAllowance.ovm"

LIBRARY FlowPathAnalysis

AUTHOR = "Gustavo Quiros";
COMMENT = "Decentralised analysis of product flow paths.";

CLASS FlowPathAnalysisAlgorithm : CLASS Decentralisation/Algorithm

IS_INSTANTIABLE;
IS_FINAL;

COMMENT = "A node of the decentralised flow path analysis algorithm.";

VARIABLES

Targets[] : STRING FLAGS = "I"
COMMENT = "The target elements to search for.";
A.4 FlowPathMonitoring.ovm

Open : BOOL FLAGS = "I"
COMMENT = "Restrict search to currently open flow routes."

Count : UINT FLAGS = "O"
COMMENT = "The number of flow routes found."

Hashes[] : UINT FLAGS = "O"
COMMENT = "The hash value of each flow route found."

Lengths[] : UINT FLAGS = "O"
COMMENT = "The length of each flow route found."

Offsets[] : UINT FLAGS = "O"
COMMENT = "The offset of each flow route in the element vector."

Elements[] : STRING FLAGS = "O"
COMMENT = "The elements of the flow routes found."

Timestamps[] : TIME FLAGS = "O"
COMMENT = "The update timestamp for each flow route."

END_VARIABLES;

OPERATIONS

typemethod : C_FUNCTION <FB_FNC_TYPEMETHOD>;

BeginExecution : C_FUNCTION
<FNC_Decentralisation_Algorithm_BeginExecution>;

HandleIncomingMessage : C_FUNCTION
<FNC_Decentralisation_Algorithm_HandleIncomingMessage>;

EndExecution : C_FUNCTION
<FNC_Decentralisation_Algorithm_EndExecution>;

Reset : C_FUNCTION
<FNC_Decentralisation_Algorithm_Reset>;

END_OPERATIONS;

END_CLASS;

END_LIBRARY;

A.4 FlowPathMonitoring.ovm

/*
 * Flow Path Monitoring Library
 * Gustavo Quiros
 * Chair of Process Control Engineering, RWTH Aachen University
 * g.quiros@plt.rwth-aachen.de
*/
/*
This library provides the implementation of the decentralised flow path monitoring algorithm. */

#include "FlowAllowance.ovm"

LIBRARY FlowPathMonitoring

AUTHOR = "Gustavo Quiros";
COMMENT = "Decentralised monitoring of product flow paths.";

CLASS FlowPathMonitoringAlgorithm : CLASS Decentralisation/Algorithm

IS_INSTANTIABLE;
IS_FINAL;

COMMENT = "A node of the decentralised flow path monitoring algorithm.";

VARIABLES

Hashes[] : UINT FLAGS = "I"
COMMENT = "The hash value of each flow route to monitor.";

Lengths[] : UINT FLAGS = "I"
COMMENT = "The length of each flow route to monitor.";

Elements[] : STRING FLAGS = "I"
COMMENT = "The elements of the flow routes to monitor.";

Count : UINT FLAGS = "O"
COMMENT = "The number of flow routes monitored.";

OpenHashes[] : UINT FLAGS = "O"
COMMENT = "The flow route hash value for each entry in the open table.";

OpenTimestamps[] : TIME FLAGS = "O"
COMMENT = "The timestamp for each entry in the open table.";

Open[] : BOOL FLAGS = "O"
COMMENT = "The flow allowance setting for each entry in the open table.";

AlertCount : UINT FLAGS = "O"
COMMENT = "The number of alerts.";

AlertHashes[] : UINT FLAGS = "O"
COMMENT = "The flow route hash value for each entry in the alerts table.";

AlertTimestamps[] : TIME FLAGS = "O"
COMMENT = "The timestamp for each entry in the alerts table.";

AlertTypes[] : STRING FLAGS = "O"
COMMENT = "The type of alert for each entry in the alerts table.";

AlertOffsets[] : UINT FLAGS = "O"
COMMENT = "The offset in the element vector for each entry in the alerts table.";

AlertElements[] : STRING FLAGS = "O"
COMMENT = "The elements of the alert for each entry in the alert table.";

END_VARIABLES;

OPERATIONS

typemethod : C_FUNCTION <FB_FNC_TYPEMETHOD>;

BeginExecution : C_FUNCTION
<FNC_Decentralisation_Algorithm_BeginExecution>;

HandleIncomingMessage : C_FUNCTION
<FNC_Decentralisation_Algorithm_HandleIncomingMessage>;

EndExecution : C_FUNCTION
<FNC_Decentralisation_Algorithm_EndExecution>;

Reset : C_FUNCTION
<FNC_Decentralisation_Algorithm_Reset>;

END_OPERATIONS;

END_CLASS;

END_LIBRARY;

A.5 FlowPathAllocation.ovm

/*
 * Flow Path Allocation Library
 * Gustavo Quiros
 * Chair of Process Control Engineering, RWTH Aachen University
 * g.quiros@plt.rwth-aachen.de
 * */

/* This library provides the implementation of the decentralised flow path allocation algorithm. */

#include "FlowAllowance.ovm"

LIBRARY FlowPathAllocation

AUTHOR = "Gustavo Quiros";
COMMENT = "Decentralised allocation of product flow paths."

CLASS FlowPathAllocationAlgorithm : CLASS Decentralisation/Algorithm

IS_INSTANTIABLE;
IS_FINAL;

COMMENT = "A node of the decentralised flow path allocation algorithm."

VARIABLES

Hash : UINT FLAGS = "I"
COMMENT = "The hash value of the flow route to allocate or deallocate."

Elements[] : STRING FLAGS = "I"
COMMENT = "The elements of the flow route to allocate or deallocate."

Allocate : BOOL FLAGS = "I"
COMMENT = "Indicates if the flow route should be allocated."

Deallocate : BOOL FLAGS = "I"
COMMENT = "Indicates if the flow route should be deallocated."

ManualSetting : INT FLAGS = "I"
COMMENT = "The manual flow allowance setting for the enclosure constraint."

Allocation : BOOL FLAGS = "O"
COMMENT = "Indicates if an allocation is being reported."

Deallocation : BOOL FLAGS = "O"
COMMENT = "Indicates if a deallocation is being reported."

Conflict : BOOL FLAGS = "O"
COMMENT = "Indicates if a conflict is being reported."

ResultElements[] : STRING FLAGS = "O"
COMMENT = "The elements of the flow route that has been allocated or deallocated."

Alloc[] : STRING FLAGS = "N"
COMMENT = "The elements of the flow route that has allocated this component."

EncloseHashes[] : UINT FLAGS = "N"
COMMENT = "The hash value of each flow route in the enclose table."

EncloseOffsets[] : UINT FLAGS = "N"
COMMENT = "The element offset of each flow route in the enclose table."

EncloseElements[] : STRING FLAGS = "N"
COMMENT = "The elements of each flow route in the enclose table."

Setting : UINT FLAGS = "N"
COMMENT = "The flow allowance setting for the enclosure constraint."
Operation : STRING FLAGS = "N"
COMMENT = "The name of the operation that is currently being executed."

Parent : STRING FLAGS = "N"
COMMENT = "The name of the parent connector of the current operation."

Children[] : STRING FLAGS = "N"
COMMENT = "The names of the child connectors of the current operation."

FlowRoute[] : STRING FLAGS = "N"
COMMENT = "The elements of the flow route of the current operation."

FlowRouteHash : UINT FLAGS = "N"
COMMENT = "The hash value of the flow route of the current operation."

BranchFlowRoute[] : STRING FLAGS = "N"
COMMENT = "The elements of the branch flow route of the current operation."

ResponseOffsets[] : UINT FLAGS = "N"
COMMENT = "The item offset of each message in the response table."

ResponseItems[] : STRING FLAGS = "N"
COMMENT = "The items of each message in the response table."

ConstraintElements[] : STRING FLAGS = "O"
COMMENT = "The elements of the constraint table."

ConstraintSettings[] : UINT FLAGS = "O"
COMMENT = "The flow allowance settings of the constraint table."

ConflictCount : UINT FLAGS = "O"
COMMENT = "The number of conflicts found during the allocation."

ConflictOffsets[] : UINT FLAGS = "O"
COMMENT = "The element offset of each flow route in the conflict table."

ConflictElements[] : STRING FLAGS = "O"
COMMENT = "The elements of each flow route in the conflict table."

ConflictTimestamps[] : TIME FLAGS = "O"
COMMENT = "The timestamp of each entry in the conflict table."

Timestamp : TIME FLAGS = "O"
COMMENT = "The timestamp of the last allocation or deallocation."

END_VARIABLES;

OPERATIONS

typemethod : C_FUNCTION <FB_FNC_TYPEMETHOD>;

BeginExecution : C_FUNCTION
<FNC_Decentralisation_Algorithm_BeginExecution>;}
A.6 FlowPathManagement.ovm

/*
 * Flow Path Management Library
 * Gustavo Quiros
 * Chair of Process Control Engineering, RWTH Aachen University
 * g.quiros@plt.rwth-aachen.de
 * /

/* This library provides the implementation of the flow path management functionality. */
#include "FlowPathAnalysis.ovm"
#include "FlowPathMonitoring.ovm"
#include "FlowPathAllocation.ovm"

LIBRARY FlowPathManagement

AUTHOR = "Gustavo Quiros";
COMMENT = "Product flow path management library.";

CLASS FlowPathManager : CLASS fb/functionblock

IS_INSTANTIABLE;
IS_FINAL;

COMMENT = "Manages flow route and flow path instances.";

VARIABLES

Discover : BOOL FLAGS = "I"
COMMENT = "Activate the discovery of product flow routes.";

Origins[] : STRING FLAGS = "I"
COMMENT = "The origin elements of each flow route specification.";
Targets[] : STRING FLAGS = "I"
COMMENT = "The target elements of each flow route specification."

Reset : BOOL FLAGS = "I"
COMMENT = "Reset all local components."

END_VARIABLES;

OPERATIONS

typemethod : C_FUNCTION <FB_FNC_TYPEMETHOD>;

END_OPERATIONS;

END_CLASS;

CLASS FlowRoute : CLASS fb/functionblock

IS_INSTANTIABLE;
IS_FINAL;

COMMENT = "Object representation of a product flow route."

VARIABLES

Description : STRING FLAGS = "I"
COMMENT = "A description of this flow route."

Origin : STRING FLAGS = "I"
COMMENT = "The origin element of this flow route."

Target : STRING FLAGS = "I"
COMMENT = "The target element of this flow route."

Hash : UINT FLAGS = "I"
COMMENT = "The hash value of this flow route object."

Length : UINT FLAGS = "I"
COMMENT = "The number of elements contained in this flow route."

Elements[] : STRING FLAGS = "I"
COMMENT = "The elements contained in this flow route."

Select : BOOL FLAGS = "I"
COMMENT = "Select this flow route to create a flow path object."

END_VARIABLES;

OPERATIONS

typemethod : C_FUNCTION <FB_FNC_TYPEMETHOD>;

END_OPERATIONS;

END_CLASS;
CLASS FlowPath : CLASS fb/functionblock

IS_INasmine; IS_FINAL;

COMMENT = "Object representation of a product flow path."

VARIABLES

CreationTime : TIME FLAGS = "O"
COMMENT = "The creation time of this flow path object."

Allocated : BOOL FLAGS = "O"
COMMENT = "Indicates if this flow path is allocated."

Locked : BOOL FLAGS = "O"
COMMENT = "Indicates if this flow path is locked."

Active : BOOL FLAGS = "O"
COMMENT = "Indicates if this flow path is active."

AlertCount : UINT FLAGS = "O"
COMMENT = "Indicates the number of alerts that this flow path has issued."

Monitoring : BOOL FLAGS = "O"
COMMENT = "Indicates if this flow path is being monitored."

Timestamp : TIME FLAGS = "O"
COMMENT = "The timestamp of the last state change of this flow path."

Allocate : BOOL FLAGS = "I"
COMMENT = "Command to allocate this flow path."

Lock : BOOL FLAGS = "I"
COMMENT = "Command to lock this flow path."

Activate : BOOL FLAGS = "I"
COMMENT = "Command to activate this flow path."

Delete : BOOL FLAGS = "I"
COMMENT = "Command to delete this flow path."

Monitor : BOOL FLAGS = "I"
COMMENT = "Command to monitor this flow path."

Open : BOOL FLAGS = "O"
COMMENT = "The open flow route indicator of this flow path."

OpenTimestamp : TIME FLAGS = "O"
COMMENT = "The timestamp of the open flow route indicator."

ConstraintElements[] : STRING FLAGS = "O"
COMMENT = "The elements of the constraint table."
ConstraintSettings[] : UINT FLAGS = "O"
COMMENT = "The flow allowance settings of the constraint table.";

ExtLock : BOOL FLAGS = "O"
COMMENT = "Command to lock this flow path externally."

ExtActivate : BOOL FLAGS = "O"
COMMENT = "Command to activate this flow path externally."

ExtLocked : BOOL FLAGS = "I"
COMMENT = "Confirmation of external lock of this flow path."

ExtActive : BOOL FLAGS = "I"
COMMENT = "Confirmation of external activation of this flow path."

END_VARIABLES;

OPERATIONS

typemethod : C_FUNCTION <FB_FNC_TYPEMETHOD>;

END_OPERATIONS;

END_CLASS;

CLASS FlowPathAlert : CLASS fb/functionblock

IS_INSTANTIABLE;
IS_FINAL;

COMMENT = "An alert from a product flow path."

VARIABLES

Type : STRING FLAGS = "O"
COMMENT = "The type of this flow path alert."

Timestamp : TIME FLAGS = "O"
COMMENT = "The timestamp of the flow path alert."

Elements[] : STRING FLAGS = "O"
COMMENT = "The elements involved in the flow path alert."

FlowPaths[] : STRING FLAGS = "O"
COMMENT = "The flow paths involved in the flow path alert."

END_VARIABLES;

OPERATIONS

typemethod : C_FUNCTION <FB_FNC_TYPEMETHOD>;

END_OPERATIONS;

END_CLASS;
CLASS FlowPathLogger : CLASS fb/functionblock

COMMENT = "Abstract base class for flow path logging services."

OPERATIONS

LogFlowPathCreation : C_FUNCTION
<FNC_FlowPathManagement_FlowPathLogger_LogFlowPathCreation>
IS_ABSTRACT;

LogFlowPathState : C_FUNCTION
<FNC_FlowPathManagement_FlowPathLogger_LogFlowPathState>
IS_ABSTRACT;

LogFlowPathDeletion : C_FUNCTION
<FNC_FlowPathManagement_FlowPathLogger_LogFlowPathDeletion>
IS_ABSTRACT;

LogFlowPathAlert : C_FUNCTION
<FNC_FlowPathManagement_FlowPathLogger_LogFlowPathAlert>
IS_ABSTRACT;

END_OPERATIONS;

END_CLASS;

CLASS OVLogfileFlowPathLogger : CLASS FlowPathManagement/FlowPathLogger

IS_INSTANTIABLE;
IS_FINAL;

COMMENT = "Flow path logger which uses the OV logfile services."

OPERATIONS

typemethod : C_FUNCTION <FB_FNC_TYPEMETHOD>;

LogFlowPathCreation : C_FUNCTION
<FNC_FlowPathManagement_FlowPathLogger_LogFlowPathCreation>;

LogFlowPathState : C_FUNCTION
<FNC_FlowPathManagement_FlowPathLogger_LogFlowPathState>;

LogFlowPathDeletion : C_FUNCTION
<FNC_FlowPathManagement_FlowPathLogger_LogFlowPathDeletion>;

LogFlowPathAlert : C_FUNCTION
<FNC_FlowPathManagement_FlowPathLogger_LogFlowPathAlert>;

END_OPERATIONS;

END_CLASS;

ASSOCIATION ComponentAssignment : ONE_TO_MANY

PARENT FlowPathManager : CLASS FlowPathManagement/FlowPathManager
COMMENT = "The flow path manager for the components.";

CHILD Components : CLASS Decentralisation/Component
COMMENT = "The components assigned to the flow path manager.";
END_ASSOCIATION;

ASSOCIATION FlowRouteContainment : ONE_TO_MANY
IS_LOCAL;

PARENT FlowPathManager : CLASS FlowPathManagement/FlowPathManager
COMMENT = "The flow path manager for the flow route objects.";

CHILD FlowRoutes : CLASS FlowPathManagement/FlowRoute
COMMENT = "The flow route objects managed by the flow path manager.";
END_ASSOCIATION;

ASSOCIATION FlowPathContainment : ONE_TO_MANY
IS_LOCAL;

PARENT FlowPathManager : CLASS FlowPathManagement/FlowPathManager
COMMENT = "The flow path manager for the flow path objects.";

CHILD FlowPaths : CLASS FlowPathManagement/FlowPath
COMMENT = "The flow path objects managed by the flow path manager.";
END_ASSOCIATION;

ASSOCIATION FlowRouteUse : ONE_TO_ONE

PARENT FlowRoute : CLASS FlowPathManagement/FlowRoute
COMMENT = "The flow route corresponding to the flow path object.";

CHILD FlowPath : CLASS FlowPathManagement/FlowPath
COMMENT = "The flow path object corresponding to the flow route.";
END_ASSOCIATION;

ASSOCIATION FlowPathLoggerAssignment : ONE_TO_MANY

PARENT FlowPathManager : CLASS FlowPathManagement/FlowPathManager
COMMENT = "The corresponding flow path manager for the loggers.";

CHILD FlowPathLoggers : CLASS FlowPathManagement/FlowPathLogger
COMMENT = "The flow path loggers assigned to the flow path manager.";
END_ASSOCIATION;

ASSOCIATION SourceComponentFlowRouteMapping : ONE_TO_MANY

PARENT SourceComponent : CLASS Decentralisation/Component
COMMENT = "The source component of the flow route objects.";
CHILD FlowRoutes : CLASS FlowPathManagement/FlowRoute
COMMENT = "The flow route objects corresponding to the source component."
END_ASSOCIATION;

ASSOCIATION FlowPathAlertContainment : ONE_TO_MANY

PARENT FlowPath : CLASS FlowPathManagement/FlowPath
COMMENT = "The flow path corresponding to the flow path alert objects."

CHILD FlowPathAlerts : CLASS FlowPathManagement/FlowPathAlert
COMMENT = "The flow path alert objects corresponding to the flow path."
END_ASSOCIATION;

END_LIBRARY;
Bibliography


[54] Siemens AG (2008, June). Add-ons for the SIMATIC PCS 7 process control system, Catalog ST PCS 7.1. Siemens AG.


Résumé

Personal Information
Name: Gustavo Arturo Quirós Araya
Born: March 17, 1976 in San José, Costa Rica

Education
1982 – 1987 Primary School, Saint Mary’s School, San Rafael de Escazú, Costa Rica
1988 – 1993 Primary and Secondary School, Saint Paul’s School and High School, Alajuela, Costa Rica

Studies
1998 – 2002 Bachelor of Science in Computer Engineering, Costa Rica Institute of Technology (Instituto Tecnológico de Costa Rica, ITCR)
2003 – 2006 Master of Science in Computer Science, RWTH Aachen University
2006 – 2009 Doctoral scholarship holder, Chair of Process Control Engineering (PLT), DFG Research Training Group (DFG-Graduiertenkolleg) 1298 “Algorithmic synthesis of reactive and discrete-continuous systems” (AlgoSyn), RWTH Aachen University

Professional Work
1999 Technical instructor, FundaTEC, ITCR, San José, Costa Rica
2000 Programming instructor, Center for Technological Information (Centro de Información Tecnológica, CIT), ITCR, San José, Costa Rica

Academic Work
2003 – 2005 Student assistant, Chair of Technical Computer Science (LTI), RWTH Aachen University
2005 Student assistant, Chair of Computer Science 2 – Software Modeling and Verification (MOVES), RWTH Aachen University
2006 Student assistant, Institute of Automatic Control (IRT), RWTH Aachen University
since 2009 Research assistant, Chair of Process Control Engineering (PLT), RWTH Aachen University