LUND UNIVERSITY

A Graphical Language for Batch Control

Johnsson, Charlotta

1999

Document Version:
Publisher's PDF, also known as Version of record

Link to publication

Citation for published version (APA):
Johnsson, C. (1999). *A Graphical Language for Batch Control*. [Doctoral Thesis (monograph), Department of Automatic Control]. Department of Automatic Control, Lund Institute of Technology (LTH).

Total number of authors:
1

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

A Graphical Language for Batch Control

Charlotta Johnsson

Automatic Control

A Graphical Language for Batch Control

A Graphical Language for Batch Control

Charlotta Johnsson

Lund 1999

To Hans and Josefine

Published by Department of Automatic Control
Lund Institute of Technology
Box 118
SE-221 00 LUND
Sweden

ISSN 0280–5316
ISRN LUTFD2/TFRT–1051–SE

©1999 by Charlotta Johnsson
All rights reserved

Printed in Sweden by Lunds Offset AB
Lund 1999

Contents

Acknowledgments . . . 9

1. Introduction . . . 11

1.1 Contribution of the Thesis . . . 15

1.2 Published Papers . . . 16

1.3 Outline of the Thesis . . . 18

2. Petri Nets . . . 21

2.1 Basic Concepts . . . 22

2.2 Formal Definition . . . 24

2.3 Petri Net Properties . . . 26

2.4 Petri Net Analysis Methods . . . 26

2.5 Generalized Petri Nets . . . 29

2.6 Other Petri Net Classes . . . 30

2.7 Summary . . . 32

3. Grafcet . . . 33

3.1 Syntax . . . 33

3.2 Dynamic Behavior . . . 37

3.3 Formal Definition . . . 40

3.4 Grafcet vs Petri Nets . . . 46

3.5 Grafcet vs Finite State Machines . . . 47

3.6 Summary . . . 49

4. High-Level Nets 50
4.1 Coloured Petri Nets 50
4.2 Coloured Grafcet 57
4.3 Object Petri Nets and LOOPN 60
4.4 OBJSA Nets 62
4.5 Other High-Level Languages Influenced by Petri Nets . 63
4.6 Summary 65
5. Grafchart 66
5.1 Graphical Language Elements 68
5.2 Actions and Receptivities 73
5.3 Parameterization and Methods 74
5.4 Grafchart – *the basic version* 76
5.5 Grafchart – *the high-level version* 86
5.6 Error Handling 100
5.7 Foundations of Grafchart 104
5.8 Grafchart vs Grafcet/Petri Nets 106
5.9 Implementation 118
5.10 Applications 128
5.11 Summary 130
6. Batch Control Systems 131
6.1 Batch Processing 132
6.2 Batch Control 134
6.3 The Batch Standard ISA-S88.01 134
6.4 Scheduling 141
6.5 Industry Practice 142
6.6 Summary 147
7. Batch Control Systems and Grafchart 149
7.1 Related Work 149
7.2 Grafchart for Batch Control 151

7.3 A Batch Scenario Implemented in G2 155
7.4 Summary 163

8. Recipe Structuring using Grafchart 164
8.1 Control Recipes as Grafchart Function Charts 165
8.2 Control Recipes as Object Tokens 170
8.3 Multi-dimensional Recipes 175
8.4 Process Cell Structured Recipes 180
8.5 Resource Allocation 184
8.6 Distributed Execution 186
8.7 Grafchart Structured Recipes vs Industry Practice . . . 188
8.8 Other Batch Related Applications of Grafchart 191
8.9 Summary 192

9. Recipe Analysis 193
9.1 The Deadlock Problem 194
9.2 Batch Recipes 196
9.3 Batch Recipe Transformation 199
9.4 Analysis 204
9.5 Using the Analysis Results 206
9.6 Grafchart Supervisor 213
9.7 Other Analysis Approaches 221
9.8 Resource Allocation and Scheduling 224
9.9 Summary 224

10. Conclusions 225
10.1 Future Research Directions 227

11. Bibliography 230

A. Abbreviations 239

B. Petri Net Reduction Methods 240

C. Formal Definition of Grafchart 243
C.1 Notation 245
C.2 Grafchart – *the Grafcet version* 247
C.3 Grafchart – *the basic version* 255
C.4 Grafchart – *the high-level version* 272
C.5 Remarks 280
D. Dynamic Simulator 281
D.1 Total Mass Balance 282
D.2 Component Mass Balance 282
D.3 Energy Balance 282
D.4 Level and Volume 284
D.5 Reaction Rates 284
D.6 Implementation 285
D.7 Notation and Constants 287
E. An Introduction to G2 289

Acknowledgments

The work presented in this thesis started in August 1993 when I started as a PhD-student at the department. Now, in February 1999, when the thesis is completed, there are several people that I would like to extend my special thanks to.

I have enjoyed my employment at the department very much. A great part of this is thanks to the persons working here; four cheerful and caring secretaries, a handful very helpful technicians, many skillful professors, associate professors and lecturers, and a lot of friendly, joyful, crazy and funny PhD-students. Thanks to all of you!

Throughout the project, Karl-Erik Årzén, my advisor, has helped and encouraged me. He has been an excellent advisor with many great ideas and a deep knowledge in my research area. Thank you Karl-Erik!

Karl Johan Åström, the head of the department, led me into the world of Automatic Control and for this I am very grateful. He has also been a never ending source of inspiration and optimism.

During the work with the thesis, I have had the opportunity to visit and work with colleagues from universities abroad. I have enormously appreciated these sessions and I would like to thank Professor René David and Professor Hassane Alla at Laboratoire d'Automatic de Grenoble, Institut National Polytechnique de Grenoble, France, Professor Venkat Venkatasubramanian at Laboratory for Intelligent Process Systems, Purdue University, USA, and Jo Simensen at Department of Engineering Cybernetics, Norwegian University of Science and Technology, Norway.

An industrial reference committee has been linked to the project and the feedback from the members has been very valuable. Participating in this committee were: Tage Jonsson (ABB Industrial Systems), Lars Pernebo (Alfa Laval Automation), Stefan Johansson (Astra Production Chemicals), Björn Perned (Van den Berghs Food), and Carl-Erik Flodmark (Kabi Pharmacia).

The work has been supported by NUTEK, the Swedish National Board for Industrial and Technical Development, Project Regina, 97-00061 and by TFR, the Swedish Research Council for Engineering Sciences, 95-759.

I would also like to thank some people, not directly involved in the project. Thanks to my friends for giving me many great moments far away from batch control systems. Thanks to my parent and my sisters for always

supporting and encouraging me. The last and most important thanks go to Hans, my beloved husband, for his love, enthusiasm and help through times of hard work. As a sign of my gratitude, the thesis is dedicated to him and to our newborn daughter Josefine.

Charlotta Johnsson

1

Introduction

Industrial manufacturing processes and production processes can generally be classified as continuous, discrete or batch. The classification depends mainly upon the appearance of the process output, [Fisher, 1990]. The output of a continuous process is a continuous flow. The output from a discrete process is individual units or groups of individual units. The output of batch processes appears as lots or quantities of materials. The product produced by a batch process is called a batch.

Batch processes are common and important in, e.g., the chemical, food and pharmaceutical industries. Batch processes are often of the multipurpose, multiproduct type, which means that different batches of different products can be produced in the same plant at the same time. The specification of how to produce a batch is called a recipe. The batch plants can have a network structure which means that the batch can take several different paths when passing through the plant. The combination of multiproduct and network-structured batch plants is the most effective and the most economical profitable plant structure. However, the multiproduct and network-structured batch plants are also the most difficult plants to control.

Economical interests have caused an increased interest for small scale productions, quick product changes, and order-based production. This means that there is a need for a flexible way of modifying the production in a plant. Recipe-based production is suitable for these purposes. Batch processes and the control of such processes have therefore lately received great interest. Even though batch production traditionally has belonged to the food, pharmaceutical and chemical industries, the idea of having a recipe-based system is applicable also in many other industries.

Recipe-based control can be seen as a special type of sequential control. Sequential control is important, not only in batch production but also in the continuous and discrete industries. All industries have devices that

have to be controlled. This is usually done sequentially, e.g. a device should first be opened and then closed, or first turned-on and later turned-off. Industrial production runs in different modes, e.g. start-up, operation and shut-down. In addition to this it is sometimes possible to decompose the production into sequential steps. Sequential control is therefore important both on a local level, compare the device control, and on a supervisory level, compare different operation modes.

Grafcet is a well known and well accepted representation format used for sequential control at the local control level. Grafcet has a well accepted and intuitive graphical syntax, and a well specified semantics. Two international standards, IEC 848 and IEC 1131-3, define Grafcet and as a result of this Grafcet has become well known in industry. In the standards, however, Grafcet is refered to as Sequential Function Charts (SFC). In industry, Grafcet is mainly used for PLC implementations, i.e., for local control. No commonly accepted representation format exists for sequential control at the supervisory control level. Grafcet is based on Petri nets. Formal analysis methods that can be used to verify properties of a system exist for Petri nets. In parallel with the development of Grafcet, Petri nets have been developed into high-level Petri nets, a richer and more elaborate version of Petri nets. Even though a strong relation exists between Grafcet and Petri nets, Petri nets are not well-known in industry.

From a system theoretic point of view sequential control logic belongs to the area of Discrete Event Dynamic Systems (DEDS). This is a relatively new area in the control community where, currently, a large amount of research is performed, e.g., [Gunnarsson, 1997], [Charbonnier, 1996], [Tittus, 1995], [Cassandras, 1993], [Ho, 1991], etc. A large number of alternative approaches have been introduced for modeling, analysis, and, in certain restricted cases also for synthesis of DEDS. However, their practical use is severely restricted. This is due to the combinatorial complexity which makes it very difficult to scale up DEDS approaches so that they can handle industrial size applications.

The research on discrete event dynamic systems can be divided into two main areas:

1. Formal methods for verification and synthesis.
2. Improved tools and languages for controller implementation.

In the first research area, formal specification languages and formal modeling techniques are used to describe the behavior of a system. Formal analysis techniques are then used to verify certain critical properties. A number of approaches have been developed. They are based on, e.g., state

machines, Petri nets, logics, and process algebras. The approaches must provide appropriate means for modeling the process and the controller. The model must be able to represent the dynamics and reactive nature of the process, and allow for proper expression of timing properties. Hence, a strong focus of formal methods is modeling and modeling languages.

Most of the proposed approaches are only concerned with verification. The verifier is presented with a formal model of the process and the control system and a specification of the desired behavior of the system. The verification problem consists of demonstrating that the model satisfies the specification. In the synthesis problem a specification is given that the process should satisfy. The synthesis problem consists of the construction of a controller that ensures that the combined model of the process and the controller fulfills the specification.

One formal method approach that has gained a lot of interest in the control community is the Supervisory Control Theory (SCT) [Ramadge and Wonham, 1989]. This approach has also been combined with Grafcet [Charbonnier, 1996], [Charbonnier *et al.*, 1995]. SCT has also been developed in to Procedural Control Theory (PCT) [Sanchez *et al.*, 1995]. Petri nets have been used as the basis for controller analysis and synthesis, in the form of Controlled Petri Nets [Holloway and Krogh, 1994], [Holloway and Krogh, 1990].

The second research area focuses on development of tools and languages for implementation of discrete event controllers. With better abstraction and structuring possibilities it is easier to get a good overview of the control problem and the implementation is simplified drastically. The focus in this research area is, in most cases, on graphical programming languages rather than on modeling and specification languages.

The two research areas complement each other. Most formal approaches only support verification. The designer must develop and implement the controller manually. If the approach also supports synthesis, a controller is constructed and presented in terms of a specification language. Also in this case, the user must implement the controller. It is therefore important to have good structuring mechanisms and good programming languages for controller implementations. Of course, it is also important to be able to use formal methods for verification of systems that have already been designed and implemented with, e.g., a high-level implementation language.

Two lines of research have been pursued in this thesis

- Developing a programming language and tool for sequential control applications.

- Finding suitable ways to represent recipe-based control. The application area has been batch processes.

The two lines can be regarded as two separate activities. However, there has been a strong interaction. Both research lines belong to the latter of the two research areas of DEDS. However, elements better categorized to the first area can be found.

The work of developing a programming language and tool for sequential control applications has consisted in providing better abstraction and structuring mechanisms for Grafcet. A secondary aim of the work has been to make use of the available formal methods that exists for Petri nets and Grafcet. The first goal is met by Grafchart, a new high-level programming language for sequential control applications presented in this thesis. Grafchart can, unlike Grafcet, be used to implement all levels in a control system, supervisory as well as local. The situation, of having better abstraction and structuring possibilities, can be compared with moving from assembly languages to object-oriented high-level languages in ordinary programming. Grafcet can be compared to an assembly language whereas Grafchart is a high-level object-oriented language. The approach taken for the secondary goal is to show that Grafchart can be translated into Grafcet and/or into Petri nets. The formal methods for analysis of Petri nets or Grafcet can then be applied.

The development of Grafchart has, as much as possible, followed the Grafcet standard. Grafchart has a syntax similar to Grafcet, i.e., it is based on the concepts of steps, representing the states, and transitions, representing the change of state. Grafchart exists in two versions: one basic version and one high-level version. The basic version is a Grafcet-based model with an extended syntax with improved abstraction facilities. A toolbox implementation exists. When the work presented in this thesis started, the basic version of Grafchart was already available, [Årzén, 1994a]. The high-level version of Grafchart contains additional high level programming language constructs and features inspired by high-level Petri nets. Also for the high-level version of Grafchart, a toolbox has been implemented.

Recipe-based control can with advantage be represented sequentially by a graphical programming language. The language should have an intuitive and clear syntax and must have good abstraction facilities. This is provided by Grafchart. In the thesis it is shown how Grafchart can be used for recipe structuring both at the lowest level of control and for the overall structuring. A recent international standard, IEC 61512 [IEC, 1997], provides a formal definition of terminology, models and functionality of

batch systems. The standard was originally an ISA standard, ISA S88.01, [ISA, 1995], and is therefore often refered to as the ISA S88.01 standard. The standard actually mentions the possibility to use Grafcet to control a batch process, i.e., to structure a recipe and to perform the actions associated with it. However, it is only suggested for the lowest level of control. No suggestions are made for how to do the overall structuring of a recipe.

The main application area of Grafchart and of this thesis is recipe-based control of batch systems. In the thesis it is shown how Grafchart can be used for recipe structuring both at the lowest level of control and for the overall structuring. By using the features of Grafchart in different ways, recipes can be represented in a number of alternative ways. All comply with the ISA S88.01 standard. The different structures are presented and their advantages and drawbacks are discussed. A simulation of a multipurpose, network structured batch plant has served as a test platform. The thesis also shows how the recipes, structured with Grafchart, can be transformed into a Petri net and analyzed with respect to deadlock situations. The results can be used for static deadlock prevention or dynamic deadlock avoidance. Grafchart, the recipe-execution system, and the batch plant are implemented in G2, an object oriented programming environment. G2 is also an industrial environment which makes it possible to directly use the results in industry.

1.1 Contribution of the Thesis

The contributions of this thesis are the following:

- The syntax of Grafchart is formally defined.
- The semantics of Grafchart is defined and the translation between Grafchart and Grafcet and between Grafchart and Petri nets is presented.
- It is shown how Grafchart can be used in recipe-based batch control both at the recipe level and at the equipment control level. A number of alternative ways of representing recipes are presented and discussed.
- It is shown how resource allocation can be integrated with recipe execution using ideas from concurrent programming and Petri nets.
- It is shown how the batch recipes can be analyzed using existing Petri net methods.

The work presented in this thesis is a continuation of the work presented in the licentiate thesis:

* Johnsson C. (1997): "Recipe-Based Batch Control Using High-Level Grafchart", *Licentiate thesis, TFRT – 3217, Dept. of Automatic Control, Lund Institute of Technology, Sweden*, [Johnsson, 1997]

The work has received a lot of interest by the members of the ISA S88.01 standardization committee.

1.2 Published Papers

The work presented in this thesis is primarily based on the following conference presentations and journal articles:

- Johnsson C. and Årzén K.-E. (1994): "High-Level Grafcet and Batch Control", *Presented at ADPM'94 (Automation of Mixed Processes: Dynamical Hybrid Systems), Brussels, Belgium*, [Johnsson and Årzén, 1994]
- Johnsson C. and Årzén K.-E. (1996): "Object-Tokens in High-Level Grafchart", *Presented at CIMAT'96 (Computer Integrated Manufacturing and Automation Technology), Grenoble, France*, [Johnsson and Årzén, 1996b]
- Årzén K.-E. and Johnsson C., and (1996): "Object-oriented SFC and ISA-S88.01 recipes", *Presented at World Batch Forum, Toronto, Canada*, [Årzén and Johnsson, 1996a]
- Johnsson C. and Årzén K.-E. (1996): "Batch Recipe Structuring using High-Level Grafchart", *Presented at IFAC'96 (International Federation of Automatic Control), San Francisco, USA*, [Johnsson and Årzén, 1996a]
- Årzén K.-E. and Johnsson C. (1996): "Object-oriented SFC and ISA-S88.01 recipes", *ISA Transactions, Vol. 35, p. 237-244*, [Årzén and Johnsson, 1996b]
- Årzén K.-E. and Johnsson C. (1997): "Grafchart: A Petri Net/Grafcet Based Graphical Language for Real-time Sequential Control Applications", *Presented at SNART'97 (Swedish Real-Time Systems Conference), Lund, Sweden*, [Årzén and Johnsson, 1997]

- Johnsson C. and Årzén K.-E. (1998): “On recipe-based structures using High-Level Grafchart”, *Presented at ADPM’98 (Automation of Mixed Processes: Dynamical Hybrid Systems), Reims, France*, [Johnsson and Årzén, 1998e]

- Johnsson C. and Årzén K.-E. (1998): “On recipe-based structuring and analysis using Grafchart”, *Accepted for publication in European Journal of Automation*, [Johnsson and Årzén, 1998f]

- Johnsson C. and Årzén K.-E. (1998): “Grafchart and batch recipe structures”, *Presented at WBF’98 (World Batch Forum), Baltimore, USA*, [Johnsson and Årzén, 1998a]

- Johnsson C. and Årzén K.-E. (1998): “Grafchart applications”, *Presented at GUS’98 (Gensym User Society), Newport, USA*, [Johnsson and Årzén, 1998c]

- Johnsson C. and Årzén K.-E. (1998): “Grafchart and its relations to Grafcet and Petri nets”, *Presented at INCOM’98 (Information Control Problems in Manufacturing), Nancy, France*, [Johnsson and Årzén, 1998b]

- Johnsson C. and Årzén K.-E. (1998): “Grafchart for recipe-based control”, *Computers and Chemical Engineering 22:12, 1998*, [Johnsson and Årzén, 1998d]

- Johnsson C. and Årzén K.-E. (1998): “Petri net analysis of batch recipes structured with Grafchart”, *Presented at FOCAPO’98 (Foundations of Computer Aided Process Operation), Snowbird, USA*, [Johnsson and Årzén, 1998g]

- Johnsson C and Årzén K.-E. (1999): “Grafchart and batch-recipe structures”(extended version of the WBF’98 paper), *Accepted for presentation at Interphex’99, New York, USA*, [Johnsson and Årzén, 1999a]

- Johnsson C and Årzén K.-E. (1999): “Grafchart and Grafcet: A comparison between two graphical languages aimed for sequential control applications”, *Accepted for presentation at IFAC’99, Beijing, China*, [Johnsson and Årzén, 1999b]

Grafchart has also been used in other areas of batch control. This work is only presented marginally in this thesis. More information can be found in the following conference presentations and journal articles:

- Viswanathan S., Venkatasubramanian V., Johnsson C., and Årzén K.-E. (1996): "Knowledge Representation and Planning Strategies for Batch Plant Operating Procedure Synthesis", *Presented at AIChE annual meeting, Chicago, USA*, [Viswanathan *et al.*, 1996]
- Simensen J., Johnsson C. and Årzén K.-E. (1996): "A Framework for Batch Plant Information Models", *Internal report, TFRT-7553, Dept. of Automatic Control, Lund Institute of Technology, Sweden*, [Simensen *et al.*, 1996]
- Simensen J., Johnsson C. and Årzén K.-E. (1997): "A Multiple-View Batch Plant Information Model", *Presented at PSE'97/ESCAPE-7, Trondheim, Norway* and *Computers and Chemical Engineering, 21:1001, 1997*, [Simensen *et al.*, 1997a] and [Simensen *et al.*, 1997b]
- Viswanathan S., Johnsson C., Srinivasan R., Venkatasubramanian V. and Årzén K.-E. (1998): "Automating Operating Procedure Synthesis for Batch Processes: Part 1. Knowledge Representation and Planning Framework", *Computers and Chemical Engineering, 22:11, 1998*, [Viswanathan *et al.*, 1998a]
- Viswanathan S., Johnsson C., Srinivasan R., Venkatasubramanian V. and Årzén K.-E. (1998): "Automating Operating Procedure Synthesis for Batch Processes: Part 2. Implementation and Application", *Computers and Chemical Engineering, 22:11, 1998*, [Viswanathan *et al.*, 1998b]

1.3 Outline of the Thesis

The work presented in the thesis is a continuation of the work presented in the licentiate thesis, [Johnsson, 1997]. This thesis consists of two parts. The first part, Chapter 2 – 5, focuses on programming languages and mathematical models for sequential control. It ends with a presentation of Grafchart, which is the language used later in the thesis. Chapter 2, 3 and 4 were, with minor modifications, published also in the licentiate thesis. Chapter 5 is an extended and reworked version of the two corresponding chapters, Chapter 5–6, in the licentiate thesis.

The second part, Chapter 6–9, describes the main application of Grafchart, recipe-based batch control. It describes batch control systems in

general and a recipe management system for batch processes in particular. It also deals with analysis of batch recipes. Parts of Chapter 6, 7 and 8 were published in the licentiate thesis. However, these chapters now contain additional material. Chapter 9 is new and was not included in the licentiate thesis.

The outline of the thesis is the following:

Chapter 2 – 4, present the foundation on which Grafchart is based. In Chapter 2, **Petri nets** are presented. Petri nets are a mathematical and graphical model. One main characteristic of the nets are their possibility to be theoretically analyzed. They can be used for simulation, verification and analysis of discrete systems. **Grafcet** is described in Chapter 3. Grafcet, or Sequential Function Chart (SFC), is a mathematical model aimed at formal specification and realization of programmable logic controllers (PLC). In order to model large systems efficiently, the Petri net model has been enlarged and developed into high-level Petri nets. In Chapter 4, these nets, and other closely related **High-Level Nets** are described.

Chapter 5 describes **Grafchart**. Grafchart is the name of both a mathematical sequential control model and a toolbox implementation. Grafchart combines the graphical features of Grafcet/SFC, with high-level programming languages constructs, object-oriented programming structures and ideas from high-level Petri nets. Grafchart exists in two versions; the basic version and the high-level version. Both versions are presented. Grafchart is aimed, not only at local level sequential control problems like PLC-programming, but also at supervisory sequential control applications. The Grafchart toolbox is implemented in G2.

Chapter 6 – 9 present batch control systems, which is the main application area of Grafchart. Chapter 6 describes **batch control systems** in general terms and gives an overview of ISA S88.01. The description of how to produce a batch is called a recipe. The chapter also includes a presentation of the batch control industry practice of today. Chapter 7 combines **batch control systems and Grafchart**. It is shown how the different features of Grafchart can be used to implement a batch control system and how they fit the ISA S88.01 standard.

In Chapter 8 different recipe structures, all implemented with Grafchart, are presented and discussed. By combining different features of Grafchart, the recipe structures can be largely variated. Advantages and drawbacks of the different structures are given. The chapter also contains a discussion on resource allocation, and a presentation of different possible strategies is given. **Recipe structuring using Grafchart** allows an in-

tegration between Petri nets based analysis methods and Grafcet/SFC based structuring and execution.

In Chapter 9 it is shown how the batch recipes can be analyzed and how possible deadlock situations due to improper resource allocation can be found. The approach taken is to transform the batch recipes that are structured with Grafchart to a Petri net. The size of the Petri net structure can be reduced and then treated with already existing analysis methods. An example is used to show how this can be done. The **recipe analysis** results can be used for static deadlock prevention or dynamic deadlock avoidance. Both alternatives are presented and discussed in the chapter.

The last chapter, Chapter 10, is dedicated to a summary of the thesis. Some **conclusions** are drawn and possible future work is proposed.

Happy Reading!

2

Petri Nets

Petri nets were proposed by the German mathematician Carl Adam Petri in the beginning of the 1960s, [Petri, 1962]. Petri wanted to define a general purpose graphical and mathematical model describing relations between conditions and events. The mathematical modeling ability of Petri nets makes it possible to set up state equations, algebraic equations, and other models governing the behavior of the modeled system. The graphical feature makes Petri nets suitable for visualization and simulation.

The Petri net model has two main interesting characteristics. Firstly, it is possible to visualize behavior like parallelism, concurrency, synchronization and resource sharing. Secondly, there exists a large number of theoretical methods for analysis of these nets. Petri nets can be used at all stages of system development: modeling, mathematical analysis, specification, simulation, visualization, and realization. Petri nets have been use in a wide range of application areas, e.g., performance evaluation, [Molloy, 1982], [Sifakis, 1978], distributed database systems, [Ozsu, 1985], flexible manufacturing systems, [Murata *et al.*, 1986], [Desrochers and Al'Jaar, 1995], [Silva and Teruel, 1996], logic controller design, [Silva and Velilla, 1982], [Valette *et al.*, 1983], multiprocessor memory systems, [Kluge and Lautenbach, 1982], and asynchronous circuits, [Yoeli, 1987].

Petri nets have during the years been developed and extended and several special classes of Petri nets have been defined. These include, e.g.: generalized Petri nets, synchronized Petri nets, timed Petri nets, interpreted Petri nets, stochastic Petri nets, continuous Petri nets, hybrid Petri nets, coloured Petri nets, object-oriented Petri nets, and multidimensional Petri nets.

In this chapter a brief overview of ordinary Petri nets and generalized Petri nets is given as well as a short description of other Petri net classes. More detailed presentations can be found in [David and Alla, 1992], [Murata, 1989], [Peterson, 1981].

2.1 Basic Concepts

A Petri net (PN) has two types of nodes: places and transitions, see Figure 2.1. A place is represented as a circle and a transition is represented as a bar or a small rectangle. Places and transitions are connected by arcs. An arc is directed and connects either a place with a transition or a transition with a place, i.e., a PN is a directed bipartite graph.

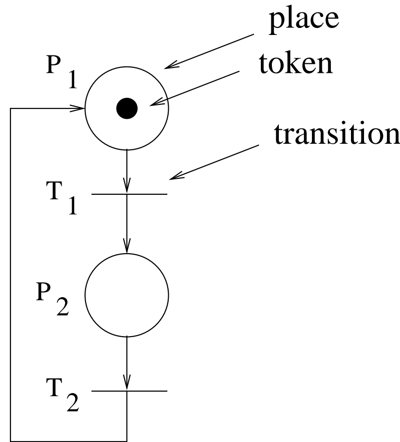


Figure 2.1 A Petri net.

The set of places is called P and the set of transitions T. Each place contains a nonnegative integer of tokens. The number of tokens contained in a placed is denoted $M(P_i)$ or m_i. The marking of the net, M, is defined by a column vector where the elements are the number of tokens contained in the corresponding places. The marking defines the state of the PN or the state of the system described by the PN. For the PN given in Figure 2.1 we have:

$$\begin{aligned} \text{Places:} \quad & P = \{P_1, P_2\} \\ \text{Transitions:} \quad & T = \{T_1, T_2\} \\ \text{Marking:} \quad & m_1 = 1,\ m_2 = 0 \\ & M = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \end{aligned}$$

The set of input (upstream) transitions of a place P_i is denoted ${}^{\circ}P_i$ and the set of output (downstream) transitions is denoted P_i°. Similar notations exist for the input and output places of a transition.

A transition without an input place is called a source transition and a transition without an output place is called a sink transition.

Systems with concurrency can be modeled with Petri nets using the and-divergence and the and-convergence structure, see Figure 2.2 (left). Systems with conflicts or choices can be modeled using the or-divergence and the or-convergence structure, see Figure 2.2 (right).

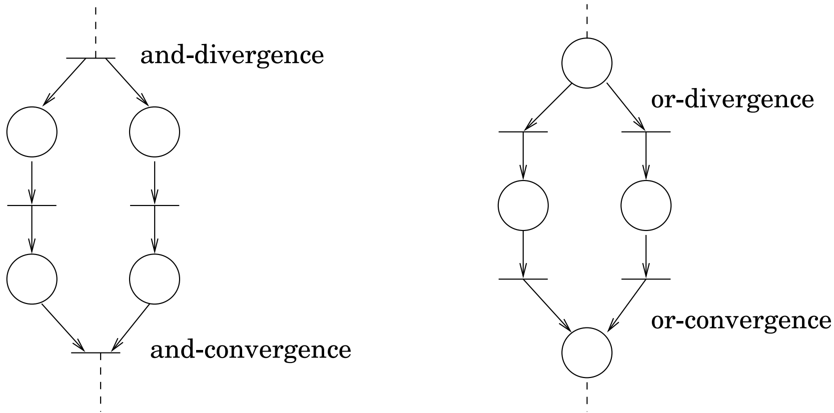


Figure 2.2 Concurrency and conflicts.

State graphs and event graphs are special classes of Petri net structures. An unmarked Petri net is a state graph (state machine) if and only if every transition has exactly one input and one output place. If the Petri net is marked its behavior will be equivalent to a classical state machine if and only if it contains only one token. A Petri net is an event graph (marked graph, transition graph) if and only if every place has exactly one input and one output transition. In a state graph parallelism cannot be modeled whereas alternatives or conflicts cannot be modeled in an event graph.

Some typical interpretations of transitions and places are shown in Table 2.1, [Murata, 1989].

Dynamic Behavior

A transition is enabled if each of its input places contains at least one token. An autonomous PN is a PN where the firing instants are either unknown or not indicated whereas a non-autonomous PN is a PN where the firing of a transition is conditioned by external events and/or time.

An enabled transition in an autonomous PN may or may not fire. An enabled transition in a non-autonomous PN is said to be fireable when the

Input Places	Transition	Output Places
Preconditions	Event	Postconditions
Input data	Computation step	Output data
Input signals	Signal processor	Output signals
Resources needed	Task or job	Resources released
Conditions	Clause in logic	Conditions
Buffers	Processor	Buffers

Table 2.1 Some typical interpretations of transitions and places in Petri nets.

transition condition becomes true. A fireable transition must fire immediately. The firing of a transition consists of removing one token from each input place and adding one token to each output place of the transition. The firing of a transition has zero duration.

2.2 Formal Definition

An unmarked ordinary PN is a 4-tuple $Q = < P, T, Pre, Post >$ *where:*

- $P = \{P_1, P_2, ...P_n\}$ *is a finite, nonempty set of places.*
- $T = \{T_1, T_2, ..., T_m\}$ *is a finite, nonempty set of transitions.*
- $P \cap T = \emptyset$*, i.e., the sets* P *and* T *are disjoint.*
- $Pre : P \times T \rightarrow \{0, 1\}$ *is the input incidence function.*
- $Post : P \times T \rightarrow \{0, 1\}$ *is the output incidence function.*

A marked ordinary PN is a pair $R = < Q, M_0 >$ *in which* Q *is an unmarked ordinary PN and* M_0 *is the initial marking.*

$Pre(P_i, T_j)$ is the weight of the arc connecting place P_i with transition T_j. This weight is 1 if the arc exists and 0 if not. $Post(P_i, T_j)$ is the weight of the arc connecting transition T_j with place P_i. This weight is 1 if the arc exists and 0 if not.

EXAMPLE 2.2.1
Consider the Petri net given in Figure 2.3.

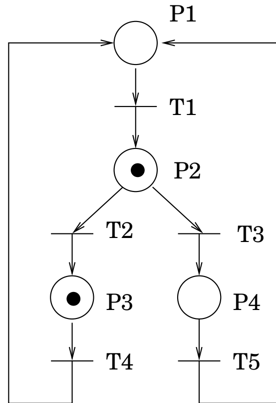


Figure 2.3 A Petri net.

The net is described by the 5-tuple

$$R =< P, T, Pre, Post, M_0 >$$

where

$$\begin{aligned} P &= \{P_1, P_2, P_2, P_4\} \\ T &= \{T_1, T_2, T_3, T_4, T_5\} \end{aligned}$$

$$\begin{aligned} W^- = [Pre(P_i, T_j)] &= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \\ W^+ = [Post(P_i, T_j)] &= \begin{bmatrix} 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \\ M_0 &= \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \end{aligned}$$

#

2.3 Petri Net Properties

The marking of a PN is a column vector whose components are the marking of place P_i. The set of reachable markings from marking M_0 is denoted $*M_0$. The firing of transition T_i from marking M_i will result in marking M_{i+1}, this is denoted:

$$M_i[T_i > M_{i+1}$$

The firing of more than one transition is called a firing sequence and is denoted S.

Home state A PN has a home state M_h for an initial marking M_0 if, for every reachable marking $M_i \in *M_0$, a firing sequence S_i exists such that $M_i[S_i > M_h$.

Reversible A PN is reversible for an initial marking M_0 if M_0 is a home state.

Bounded A place P_i is bounded for an intital marking M_0 if there is a nonnegative integer k such that, for all markings reachable from M_0, the number of tokens in P_i is not greater than k (P_i is said to be k-bounded). A PN is said to be bounded for an initial marking M_0 if all the places are bounded for M_0 (the PN is k-bounded if all the places are k-bounded)

Safe A PN is safe for an initial marking M_0 if, for all reachable markings, each place contains at most one token, i.e., the net is 1-bounded.

Live A transition T_j is live for an initial marking M_0 if, for every reachable marking $M_i \in *M_0$, a firing sequence S from M_i exists, which contains transition T_j. A PN is live for an initial marking M_0 if all its transitions are live from M_0.

Deadlock-free A deadlock is a marking such that no transition is enabled. A PN is said to be deadlock-free for an initial marking M_0 if no reachable marking $M_i \in *M_0$ is a deadlock.

2.4 Petri Net Analysis Methods

There are three main methods for analyzing a PN, i.e., to find out if a certain property holds; (1) the reachability graph and the related coverability tree, (2) linear algebra methods, and (3) reduction techniques.

(1) Reachability graph and Coverability tree The basic and most simple method is to draw the reachability graph. In this graph the nodes correspond to the reachable markings and the arcs correspond to the firing of transitions. In Figure 2.4 a PN and its reachability graph is shown. The graph can be used to find out that the PN is safe, live, reversible and that it has two repetitive components $T_1T_2T_4$ and $T_1T_3T_5$. Another, equivalent name for the reachability graph is the marking graph.

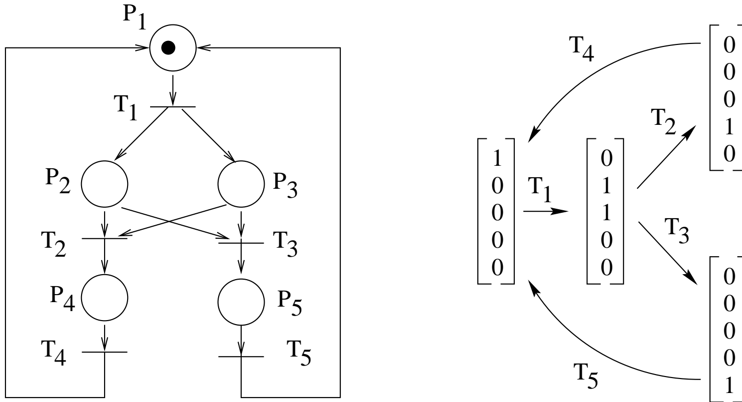


Figure 2.4 A Petri net and its reachability graph.

In an unbounded net, the number of tokens in a place can be infinite, this makes the reachability graph undefined. An unbounded net is instead represented by associating the symbol ω with the places that might have an infinitely large number of tokens. The resulting graph is now called the coverability tree, see Figure 2.5 (upper right). If the nodes that correspond to the same marking are merged together the coverability graph is obtained, see Figure 2.5 (lower right).

(2) Linear algebra The linear algebra methods are mathematical methods used to determine the properties of a net. The fundamental equation of a net is given by:

$$M_k = M_i + W \cdot \underline{S}$$

where $\underline{S}$ is the characteristic vector of the firing sequence S that takes marking M_i to marking M_k. The j'th element in the column vector $\underline{S}$ corresponds to the number of firings of transition j in the sequence S.

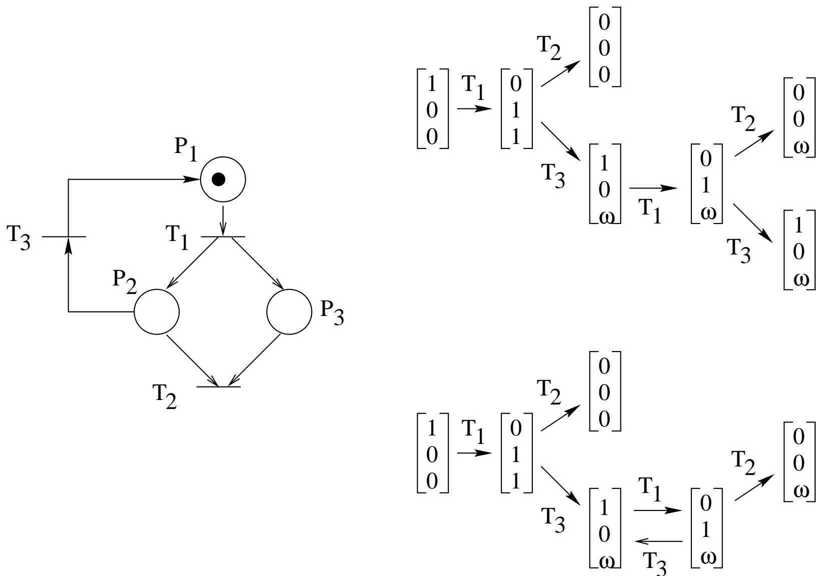


Figure 2.5 A Petri net and its coverability graph (lower) and coverability tree (upper).

The matrix W is the incidence matrix defined as:

$$
\begin{aligned}
W &= W^{+} - W^{-} \\
W^{+} &= [\,Post(P_i, T_j)\,] \\
W^{-} &= [\,Pre(P_i, T_j)\,]
\end{aligned}
$$

The marking of a Petri net can be changed through the firing of its transitions. If a deadlock situation does not occur the number of firings is unlimited. However, not all markings can be reached and not all firing sequences can be carried out. The restrictions are given by the invariants of the net. A marking invariant is obtained if a weighted sum of the marking of a subset of the places in a net is always constant, no matter what the firing might be. The places contained in this subset is a conservative component and the vector containing the weights is the P-invariant. The conservative components of a net often have a physical interpretation. If the firing of a certain sequence of transitions results in the same marking as it was started from, the sequence is called a repetitive component. The characteristic vector of the firing sequence is the T-invariant.

The P-invariants and the T-invariants of a net can be deduced by linear algebra methods, [David and Alla, 1992].

(3) Reduction methods Although the construction of the reachability graph is an efficient way of determining the properties of a small-size PN it is not a suitable method when a net has a large number of reachable markings. However, reduction methods exist that transform a large size PN into a PN of a smaller size. The idea of the reduction methods is to successively apply local transformation rules that transform the net into a simpler net, i.e., into a net with a smaller number of places and transitions, while preserving the properties of the net that one wants to investigate.

Four reduction rules exist that preserves the properties: live, bounded, safe, deadlock-free, with home state and conservative, [Murata and Koh, 1980].

1. Reduction R_1: Substitution of a place
2. Reduction R_2: Implicit place
3. Reduction R_3: Neutral transition
4. Reduction R_4: Identical transitions

Two reduction methods exist that preserves the invariants, [Berthelot, 1986].

1. Reduction R_a: Self loop transitions
2. Reduction R_b: Pure transitions

The reduction rules are developed for autonomous Petri nets. The rules are briefly described in Appendix B.

2.5 Generalized Petri Nets

In a generalized PN weights (strictly positive integers) are associated with the arcs. The weight associated with the arc from place p_j to transition t_i is denoted $w(p_j, t_i)$ and the weight associated with the arc from transition t_i to place p_k is denoted $w(t_i, p_k)$. In a generalized Petri net a transition t_i is enabled if each input place p_j of t_i contains at least $w(p_j, t_i)$ tokens. A firing of an enabled transition t_i consists of removing $w(p_j, t_i)$ tokens from each input place p_j of t_i and adding $w(t_i, p_k)$ tokens to each output place p_k of t_i, see Figure 2.6. If a weight is not explicitly specified it is assumed to be 1.

All generalized PN can be transformed into ordinary PN.

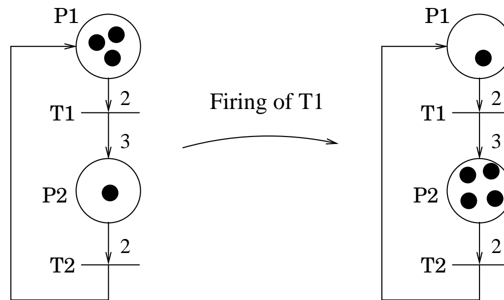


Figure 2.6 A Generalized Petri net.

2.6 Other Petri Net Classes

A number of special classes of Petri nets have been defined. They are briefly described in this section.

Synchronized PN

In ordinary autonomous PN an enabled transition may or may not fire, i.e., the ordinary PN are asynchronous. In synchronized Petri nets external events are associated with the transitions. If a transition is enabled and the associated event occurs, the transition will immediately fire. It is assumed that two external events can never occur simultaneously.

A synchronized PN is a triple $<R, E, Sync>$ *where:*

- R *is a marked PN*
- E *is the set of external events*
- *Sync is a function from* T *to* $E \cup \{e\}$*, where e is the always occurring event, i.e., an event that is always true.*

Timed PN

Using timed PN, systems with time dependent behavior can be described. The timing can either be associated with the places or with the transitions. These nets are called P-timed PN and T-timed PN respectively.

A P-timed PN is a couple $<R, Time>$ *such that:*

- R *is a marked PN*
- *Time is a function from the set* P *of places to the set of positive or zero numbers.* $Time(P_i) = d_i$. d_i *is the timing associated with place* P_i.

When a token arrives in place P_i of a P-timed PN, the token must remain in this place at least d_i time units. During this time the token is unavailable. When time d_i has elapsed, the token becomes available again and can now participate in the enabling of a transition.

A T-timed PN is a couple $<R, Time>$ *such that:*

- *R is a marked PN*
- *Time is a function from the set T of transitions to the set of positive or zero numbers.* $Time(T_j) = d_j$. d_j *is the timing associated with transition* T_j.

When a token arrives in a place of a T-timed PN, it directly enables the output transition of this place. Before the firing of the output transition can take place the token has to be reserved for at least d_j time units. During this time the token is unavailable for any other activity. When d_j time units have elapsed the transition can fire and the token is removed from the preceding place and placed in the succeeding place of the transition.

Interpreted PN

An interpreted PN is a synchronized, P-timed Petri net. Associated with the places are operations $O = \{O_1, O_2, \cdots\}$. The interpreted PN has a data processing part whose state is defined by a set of variables $V = \{V_1, V_2, \cdots\}$. This state is modified by the operations. The variables determine the value of the condition (predicate) C associated with the transition. The operation of an interpreted PN together with its environment is shown in the left part of Figure 2.7. The right part of the same figure shows that the timing d_i and the operation O_i are associated with the place P_i while the event E_j and the condition C_j are associated with transition T_j. The behavior of an interpreted Petri net is strongly related to Grafcet.

Stochastic PN

In a timed PN a fixed duration is associated with each place or with each transition. In stochastic PN a random time is associated with the transition. The most common hypothesis is that the timing is distributed according to an exponential law. The marking at time t, $M(t)$ of a stochastic PN is then an homogeneous Markov process. A Markov chain can thus be associated with every stochastic PN and the probabilities of states in stationary behavior can be calculated.

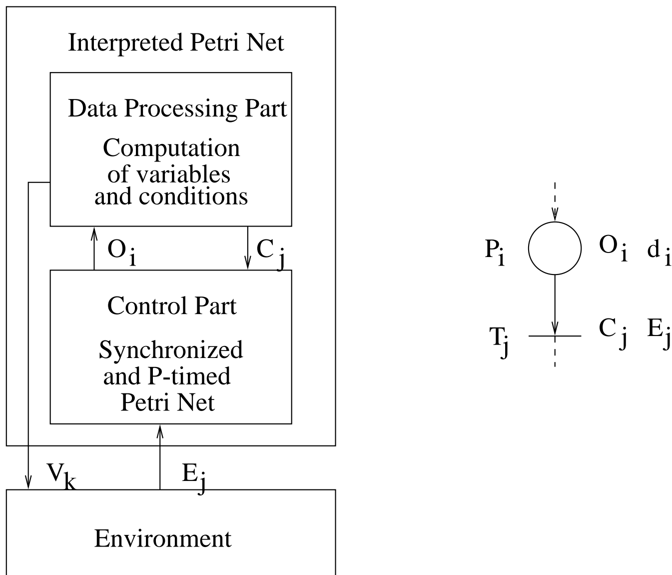


Figure 2.7 An interpreted Petri net.

Continuous PN and Hybrid PN

In a continuous PN the numbers of tokens in a place is given by a real number. It is possible to put weights on the arcs and thereby have a generalized continuous PN, the weights can also be real numbers. A hybrid PN contains one discrete part and one continuous part.

2.7 Summary

Petri nets can conveniently be used to model systems with e.g., concurrency, synchronization, parallelism and resource sharing. The graphical nature of Petri nets also makes them suitable to use for visualization and simulation of systems. In addition to this, the nets can be theoretically analyzed with different analysis methods.

3

Grafcet

Grafcet was proposed in France in 1977 as a formal specification and realization method for logical controllers. The name Grafcet was derived from `graph`, since the model is graphical in nature, and `AFCET` (Association Française pour la Cybernétique Economique et Technique), the scientific association that supported the work.

During several years, Grafcet was tested in French industries. It soon proved to be a convenient tool for representing small and medium scale sequential systems. Grafcet was therefore introduced in the French educational programs and proposed as a standard to the French association AFNOR where it was accepted in 1982. In 1988 Grafcet, with minor changes, was also adopted by the International Electrotechnical Commission (IEC) as an international standard named IEC 848, [IEC, 1988]. In this standard Grafcet goes under the name Sequential Function Chart (SFC). Seven years later, in 1995, the standard IEC 1131-3, with SFC as essential part, arrived, [IEC, 1993]. The standard concerns programming languages used in Programmable Logic Controllers (PLC). It defines four different programming language paradigms together with SFC. No matter which of the four different languages that is used, a PLC program can be structured with SFC.

Because of the two international standards, Grafcet, or SFC, is today widely accepted in industry, where it is used as a representation format for sequential control logic at the local PLC level.

In this chapter a brief overview of Grafcet is given. A more thorough presentation can be found in [David and Alla, 1992].

3.1 Syntax

Grafcet has a graphical syntax. It is built up by steps, drawn as squares, and transitions, represented as bars. The initial step, i.e., the step that

should be active when the system is started, is represented as a double square. Grafcet has support for both alternative and parallel branches, see Figure 3.1.

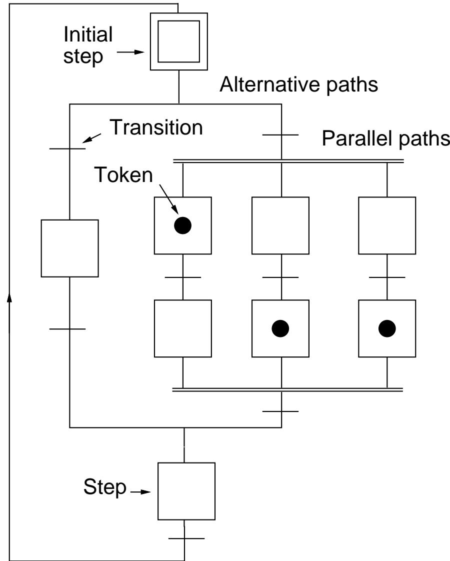


Figure 3.1 Grafcet graphical syntax.

Steps

A step can be active or inactive. An active step is marked with one (and only one) token placed in the step. The steps that are active define the situation or the state of the system. To each step one or several actions can be associated. The actions are performed when the step is active.

Transitions

Transitions are used to connect steps. Each transition has a receptivity. A transition is enabled if all steps preceding the transition are active. When the receptivity of an enabled transition becomes true the transition is fireable. A fireable transition will fire immediately. When a transition fires the steps preceding the transition are deactivated and the steps succeeding the transition are activated, i.e., the tokens in the preceding steps are deleted and new tokens are added to the succeeding steps.

Actions

There are two major categories of actions: level actions and impulse actions. A level action is modeled by a binary variable and has a finite duration. The level action remains set all the time while the step, to which the action is associated, is active. When the step is deactivated, the action is reset. A level action may be conditional or unconditional. An impulse action is responsible for changing the value of a variable. The variable can, but must not, be a binary variable. An impulse action is carried out as soon as the step changes from being inactive to active. A variable representing time may be introduced to create time-delayed actions and time-limited actions. A level action can always be transformed into a set of impulse actions.

In Figure 3.2 (left) two steps, `x1` and `x2`, are shown. A level action, `A`, is associated with the upper step and an impulse action, `B*`, is associated with the lower step. In the same figure (right) an example is given describing the activation and deactivation of the two steps together with the duration of the two actions.

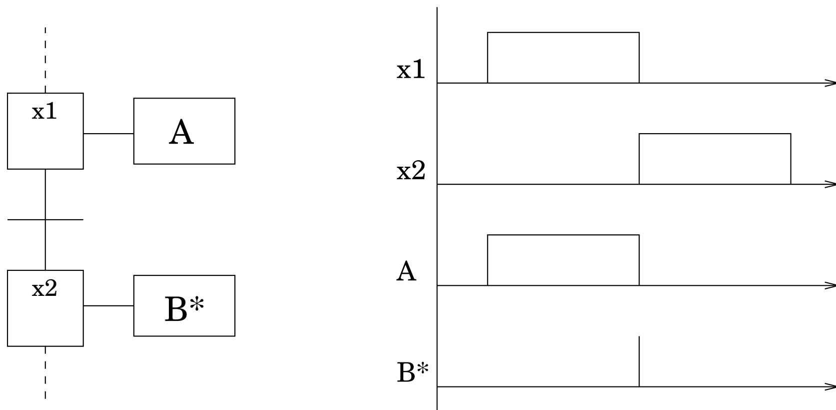


Figure 3.2 Level and impulse actions.

A situation can be stable or unstable. If the transition following a step is immediately fireable when the step becomes active, the situation is said to be unstable. An impulse action is carried out even if the situation is unstable whereas a level action is performed only if the situation is stable. Between two different external events it is assumed that there is always enough time to reach a stable situation, i.e., two external events are assumed never to occur so close in time that the system does not have time to reach a stable situation.

Receptivities

Each transition has a receptivity. A receptivity may either be a logical condition, an event, or an event and a condition. In Figure 3.3 (left) three transitions and their receptivities are shown. The receptivity of the first transition is an event, $\uparrow x$. The receptivity of the second transition is a condition, y, and the receptivity of the last transition is a combination of a condition and an event, $x \cdot \uparrow z$.

In the same figure (right) an example is given of the events and conditions and the corresponding activation and deactivation of the two steps `x1` and `x2`.

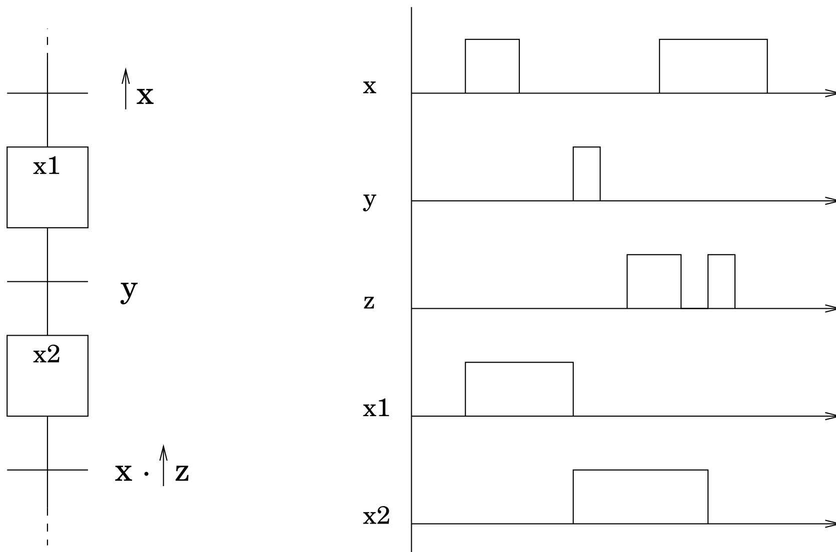


Figure 3.3 Three receptivities.

Macro steps

To facilitate the description of large and complex systems macro steps can be used. A macro step is a step with an internal representation that facilitates the graphical representation and makes it possible to detail certain parts separately. A macro step has one input and one output step. When the transition preceding the macro step fires the input step of the macro step is activated. The transition succeeding the macro step does not become enabled until the execution of the macro step reaches its output step. The macro step concept is shown in Figure 3.4.

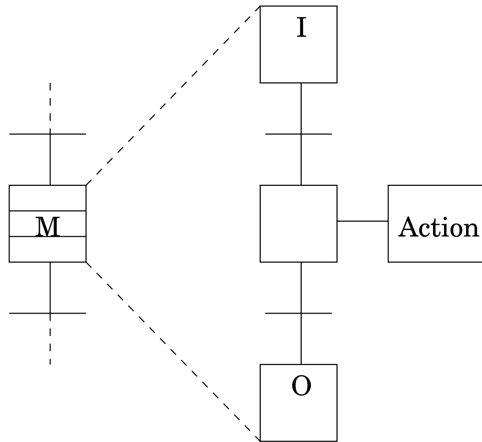


Figure 3.4 A macro step.

3.2 Dynamic Behavior

The dynamic behavior of Grafcet is defined by five rules, [David, 1995].

1. The initial situation of a Grafcet is determined by its initial steps.
2. A transition is enabled if all of its previous steps are active. A enabled transition is fireable if its associated receptivity is true. A fireable transition is immediately fired.
3. Firing of a transition results in deactivation of its previous step and a simultaneous activation of its following steps.
4. Simultaneously fireable transitions are simultaneously fired.
5. If an active step is to be simultaneously deactivated and activated it remains active.

EXAMPLE 3.2.1
In Figure 3.5 (left) a system consisting of a tank is shown. The tank has an inlet valve V_1 and an outlet valve V_2. There are two level sensors L_0 and L_1, one temperature sensor T and one heater Q.

The Grafcet for controlling the tank system is shown in Figure 3.5 (right). When the system is started, valve V_1 should open and the filling should start. When the level in the tank reaches L_0, the heating should start. The tank is now heated and filled in parallel. When the level in the tank reaches L_1 the filling is stopped and when the temperature reaches T the

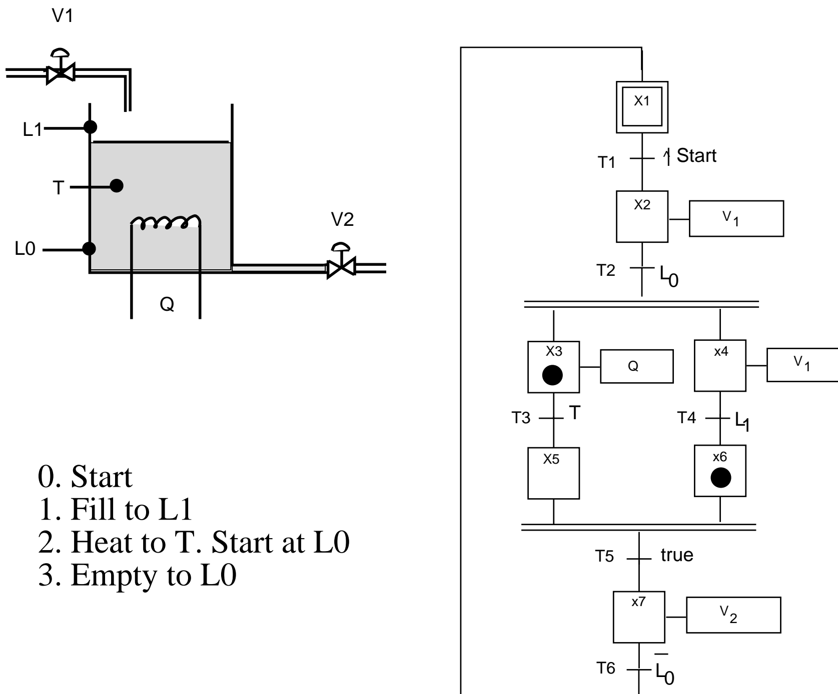


Figure 3.5 A tank example.

heating is stopped. When both the right level and the right temperature are reached, the tank is emptied. This is done by opening the outlet valve V_2. When the system is empty, i.e., when the level in the tank is below L_0, the sequence can be restarted. All the actions shown in Figure 3.5 (right) are level actions, e.g., the valve V_2 is open (boolean variable V2=1) as long as step x7 is active.

Interpretation Algorithm

A Grafcet describes a logic controller, i.e., it specifies the relation between the input sequence and the output sequence. The interpretation of a Grafcet must be without ambiguity, i.e., the same input sequence applied to a Grafcet must result in the same output sequence, independently of the person or system interpreting the Grafcet. The output sequences must be equivalent both with respect to the relative order between the actions and with respect to the timing. The aim of the interpretation algorithm given below, [David and Alla, 1992], is to define exactly how a Grafcet should be interpreted so that there will never be any ambiguities.

The interpretation algorithm is based on two assumptions:

1. The system to be controlled by the Grafcet is assumed to be slower than the logic controller implemented by the Grafcet.
2. Two external events do never occur simultaneously.

Algorithm:

1. Initialization: activate the initial steps and execute the associated impulse actions. Go to Step 5
2. When a new external event occurs, determine the set T_1 of transitions fireable on occurrence of this event. If T_1 is not empty, go to Step 3. If T_1 is empty, modify, if necessary, the state of the conditional actions associated with the active steps. Go to Step 2 and wait for a new external event.
3. Fire all the fireable transitions. Go to Step 6 if the situation remains unchanged after this firing.
4. Execute the impulse actions associated with the steps that became active at Step 3.
5. Determine the set of transitions, T_2, that are fireable on the occurrence of the event e, i.e., the transitions that have a receptivity that is true. Go to Step 3 if T_2 is not empty.
6. A stable situation is reached
   (a) Determine the set A_0 of the level actions which should be deactivated.
   (b) Determine the set A_1 of the level actions that should activated.
   (c) Set all the actions that belong to A_0 but not to A_1 to 0. Set all the actions that belong to A_1 to 1. Go to Step 2.

The algorithm searches for stable situations (step 6). However, the algorithm is based on the assumption that a finite number of iterations always results in a stable situation. Therefore, it might seem as if it is not necessary to do the search. Another simpler algorithm exists where the search is not done. However, the shorter algorithm does not give a correct result if the net contains unstable situations. Even though the shorter algorithm does not always give a correct output, this algorithm is the one used most often in industrial implementations of Grafcet, [Gaffe, 1996].

An interpretation algorithm that works in a synchronous manner has been developed, [Gaffe, 1996]. The output of a Grafcet interpreted with

the synchronous algorithm will in most cases be equivalent to that of the algorithm with search for stability. However, since the output sometimes differ, a Grafcet together with a synchronous interpretation algorithm has been given a special name, SGrafcet.

Implementation Aspects

The interpretation algorithm to choose when implementing a Grafcet is the algorithm where a search for stable situations is done. However, certain practical aspects must be taken into consideration if the real behavior should be consistent with the interpretation (theoretical behavior). The algorithm is based on the assumption that no external events can occur simultaneously. However, when a logic controller, i.e., a Grafcet, is implemented, two events may occur so close in time that there is no technical mean to distinguish which event occurred first. This might cause problems in a conflict situation (alternative path). Situations like this should therefore be avoided. This is done by making the receptivities mutually exclusive.

3.3 Formal Definition

A Grafcet can formally be defined in many ways. The definition does only describe the structure of a Grafcet, its actions and its receptivities. To execute a Grafcet an interpretation algorithm must be applied. In other words, the formal definition defines the syntax whereas the interpretation algorithm defines the semantics.

A Grafcet can be defined as a 5-tuple

$$G = < \mathcal{V}_r, \mathcal{V}_a, \mathcal{X}, \mathcal{T}, I >$$

where:

- $\mathcal{V}_r$ *is the set of variables originating from the plant,* $\mathcal{V}_{ext}$*, or corresponding to the internal states of the Grafcet,* $\mathcal{V}_{int}$*.*

  $$\mathcal{V}_r = \{\mathcal{V}_{ext} \cup \mathcal{V}_{int}\}$$

  $\mathcal{V}_{ext}$ *and* $\mathcal{V}_{int}$ *can either be conditions, i.e., variables that are true or false, or events, i.e., variables that change values.*

- *$\mathcal{V}_a$ is the set of variables issued to the plant. $\mathcal{V}_a$ can either be continuous, i.e., a variable that is set to true or false, or discontinuous, i.e., a variable that is given a new value.*
- *$\mathcal{X}$ is the set of steps, $\mathcal{X} = \{x_1, x_2, x_3, ...\}$.*
  A step $x_i \in \mathcal{X}$ is defined by

  $$\{action(x_i)\}$$

  where

  - *$action(x_i)$ defines the actions associated with the step x_i, $action(x_i) \in \mathcal{V}_a$.*
- *$\mathcal{T}$ is the set of transitions, $\mathcal{T} = \{t_1, t_2, t_3, ...\}$.*
  A transition $t \in \mathcal{T}$ is defined by the 3-tuple

  $$\{X_{PR}(t), X_{FO}(t), \varphi(t)\}$$

  where

  - *$X_{PR}(t)$ is the set of previous steps of t, $X_{PR}(t) \in \mathcal{X}$.*
  - *$X_{FO}(t)$ is the set of the following steps of t, $X_{FO}(t) \in \mathcal{X}$.*
  - *$\varphi(t)$ is the receptivity associated with t, $\varphi(t) \in \{\mathcal{V}_r \cup true \cup false\}$.*
- *I is the set of initial steps, $I \subseteq \mathcal{X}$.*

EXAMPLE 3.3.1
The Grafcet given in Figure 3.5 is represented by the 5-tuple G.

$$G =< \mathcal{V}_r, \mathcal{V}_a, \mathcal{X}, \mathcal{T}, I >$$

where:

$$\mathcal{V}_r = \{\uparrow Start, L_0, T, L_1, \overline{L_0}\}$$

$$\mathcal{V}_a = \{V_1, Q, V_2\}$$

$$\mathcal{X} = \{X1, X2, X3, X4, X5, X6, X7\}$$

$$action(X2) = V_1$$
$$action(X3) = Q$$
$$action(X4) = V_1$$
$$action(X7) = V_2$$

$$\mathcal{T} = \{T1, T2, T3, T4, T5, T6\}$$

$$X_{PR}(T1) = X1$$
$$X_{PR}(T2) = X2$$
$$X_{PR}(T3) = X3$$
$$X_{PR}(T4) = X4$$
$$X_{PR}(T5) = \{X5, X6\}$$
$$X_{PR}(T6) = X7$$

$$X_{FO}(T1) = X2$$
$$X_{FO}(T2) = \{X3, X4\}$$
$$X_{FO}(T3) = X5$$
$$X_{FO}(T4) = X6$$
$$X_{FO}(T5) = X7$$
$$X_{FO}(T6) = X1$$

$$\varphi(T1) = \uparrow Start$$
$$\varphi(T2) = L_0$$
$$\varphi(T3) = T$$
$$\varphi(T4) = L_1$$
$$\varphi(T5) = true$$
$$\varphi(T6) = \overline{L_0}$$

$$I = \{X1\}$$

#

If macro steps are to be considered, the definition will be as follows:

A Grafcet, with macro steps taken into consideration, can be defined as a 6-tuple

$$G = < \mathcal{V}_r, \mathcal{V}_a, \mathcal{X}, \mathcal{T}, \mathcal{M}, I >$$

where:

- $\mathcal{V}_r$ *is defined as before.*
  $\mathcal{V}_r$ *can be divided into two subsets,* $\mathcal{V}_{r\mathcal{X}}$ *and* $\mathcal{V}_{r\mathcal{M}}$. $\mathcal{V}_{r\mathcal{X}}$ *are the variables associated with the transitions not included in the macro steps.* $\mathcal{V}_{r\mathcal{M}}$ *are the variables associated with the transitions included in the macro steps. The two subsets are not necessarily disjoint.*

- $\mathcal{V}_a$ *is defined as before.*
  $\mathcal{V}_a$ *can be divided into two subsets,* $\mathcal{V}_{a\mathcal{X}}$ *and* $\mathcal{V}_{a\mathcal{M}}$. $\mathcal{V}_{a\mathcal{X}}$ *are the variables associated with the steps not included in the macro steps.* $\mathcal{V}_{a\mathcal{M}}$ *are the variables associated with the steps included in the macro steps. The two subsets are not necessarily disjoint.*

- $\mathcal{X}$ *is defined as before.*
  $action(x_i) \in \mathcal{V}_{a\mathcal{X}}$

- $\mathcal{T}$ *is defined as above.*
  $X_{PR}(t) \in \{\mathcal{X} \cup \mathcal{M}\}$
  $X_{FO}(t) \in \{\mathcal{X} \cup \mathcal{M}\}$
  $\phi(t) \in \{\mathcal{V}_{r\mathcal{X}} \cup true \cup false\}$

- $\mathcal{M}$ *is the finite set of macro steps,* $\mathcal{M} = \{M_1, M_2, \cdots\}$.
  A macro step M_i *is defined as a 7-tuple*

  $$M_i = < \mathcal{V}_{ri}, \mathcal{V}_{ai}, \mathcal{X}_i, \mathcal{T}_i, \mathcal{M}_i, In_i, Out_i >$$

  where:

  - $\mathcal{V}_{ri}, \mathcal{V}_{ai}, \mathcal{X}_i, \mathcal{T}_i$ *are defined as* $\mathcal{V}_r, \mathcal{V}_a, \mathcal{X}, \mathcal{T}$ *defined above.*
  - $\mathcal{M}_i$ *is the set of macro steps,* $\mathcal{M}_i = \{M_{i,1}, M_{i,2}, \cdots\}$. *The macro steps are not allowed to be infinitely recursive.*
  - In_i *is the input step of the macro step,* $In_i \subseteq \mathcal{X}_i$.
  - Out_i *is the output step of the macro step,* $Out_i \subseteq \mathcal{X}_i$.

  $\mathcal{V}_{r\mathcal{M}} = \{\mathcal{V}_{r1} \cup \mathcal{V}_{r2} \cup \cdots\}$.
  $\mathcal{V}_{a\mathcal{M}} = \{\mathcal{V}_{a1} \cup \mathcal{V}_{a2} \cup \cdots\}$.

- I *is the set of initial steps,* $I \subseteq \mathcal{X}$.

EXAMPLE 3.3.2
The Grafcet given in Figure 3.6 is represented by the 6-tuple G.

$$G = < V_r, V_a, \mathcal{X}, \mathcal{T}, \mathcal{M}, I >$$

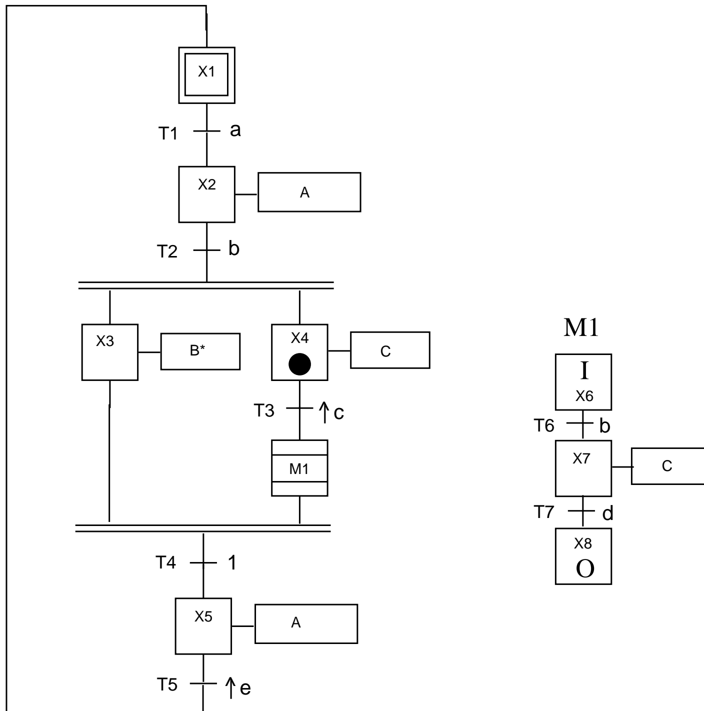


Figure 3.6 A Grafcet with a macro step.

$$\begin{aligned} V_r &= \{V_{rX} \cup V_{r\mathcal{M}}\} \\ V_{rX} &= \{a, b, \uparrow c, \uparrow e\} \end{aligned}$$

$$\begin{aligned} V_a &= \{V_{aX} \cup V_{a\mathcal{M}}\} \\ V_{aX} &= \{A, B*, C\} \end{aligned}$$

$$
\begin{aligned}
\mathcal{X} \;=\;& \{X1, X2, X3, X4, X5\} \\
& action(X2) = A \qquad action(X3) = B* \\
& action(X4) = C \qquad action(X5) = A \\[1ex]
\mathcal{T} \;=\;& \{T1, T2, T3, T4, T5\} \\
& X_{PR}(T1) = X1 \qquad X_{FO}(T1) = X2 \\
& X_{PR}(T2) = X2 \qquad X_{FO}(T2) = \{X3, X4\} \\
& X_{PR}(T3) = X4 \qquad X_{FO}(T3) = M1 \\
& X_{PR}(T4) = \{X3, M1\} \qquad X_{FO}(T4) = X5 \\
& X_{PR}(T5) = X5 \qquad X_{FO}(T5) = X1 \\
& \varphi(T1) = a \qquad \varphi(T2) = b \\
& \varphi(T3) = \uparrow c \qquad \varphi(T4) = 1 \\
& \varphi(T5) = \uparrow e \\[1ex]
\mathcal{M} \;=\;& \{M1\} \\
& V_{r\mathcal{M}} = V_{r1} = \{b, d\} \\
& V_{a\mathcal{M}} = V_{a1} = \{C\} \\
& \mathcal{X}_1 = \{X6, X7, X8\} \\
& \quad action(X7) = C \\
& \mathcal{T}_1 = \{T6, T7\} \\
& \quad X_{PR}(T6) = X6 \qquad X_{FO}(T6) = X7 \\
& \quad X_{PR}(T7) = X7 \qquad X_{FO}(T7) = X8 \\
& \quad \varphi(T6) = b \qquad \varphi(T7) = d \\
& \mathcal{M}_1 = \emptyset \\
& In_1 = X6 \\
& Out_1 = X8 \\[1ex]
I \;=\;& \{X1\}
\end{aligned}
$$

#

3.4 Grafcet vs Petri Nets

Grafcet has many similarities to Petri nets. The Petri net model that is closest in nature to Grafcet is Interpreted Petri nets, see Chapter 2.6.

1. Both models have two types of nodes: steps and transitions for Grafcet, places and transitions for Petri nets.
2. In both models, the net execution is synchronized by external events.

However, there are also differences:

1. The marking of a Grafcet is boolean whereas the marking of a Petri net is given by a nonnegative integer.
2. All simultaneously fireable transitions will be simultaneously fired in a Grafcet whereas in an Interpreted Petri nets the transitions will be fired in a sequence, called the complete firing sequence. If the sequence is not maximal all the fireable transitions will not be fired. This difference means that, for a Grafcet, in an or-divergence situation, with all receptivities being true, the transitions in both branches will be fired and both the "alternative" branches will be executed. This is often not the designers intention and it is recommended that these transitions are made mutually exclusive. An Interpreted Petri net treats the situation by nondeterministically choosing one of the branches.
3. The conditions used in Grafcet can depend on the state of the marking, this is not the case in an Interpreted Petri net.

Despite the differences that exist, an Interpreted Petri net can be made equivalent to a Grafcet and a Grafcet can be made equivalent to an Interpreted Petri net, see Figure 3.7.

Interpreted PN → Grafcet If the PN is safe, see Chapter 2.3, the first difference does not exists. If the Interpreted PN is such that no transitions contained in a conflict situation can be fired at the same time the behavior of the Petri net is deterministic and the second difference is removed. The third difference can be avoided by rewriting the Interpreted Petri net.

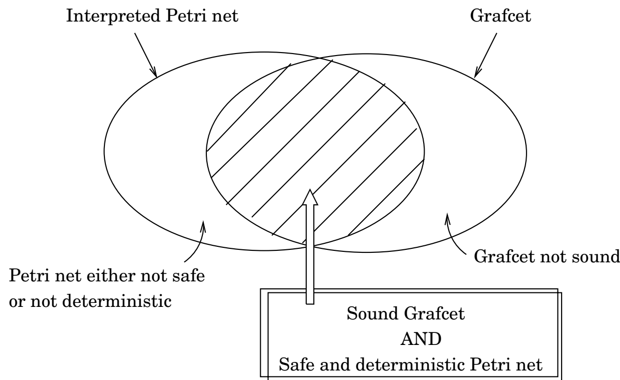


Figure 3.7 Comparison between Grafcet and Interpreted Petri nets.

Grafcet → Interpreted PN If the Grafcet is sound, see below, the Grafcet is equivalent to an Interpreted Petri net.

A sound Grafcet has the following properties, [David, 1995]

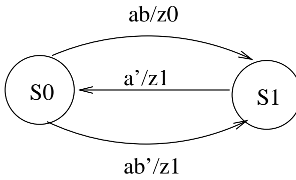
- An active step cannot have a fireable input transition without having an output transition that is simultaneously fireable.
- For any pair of simultaneously fireable transitions, they have no input step in common.
- No step can have more than one simultaneously fireable input transition.

3.5 Grafcet vs Finite State Machines

The classical models for describing sequential systems are state machines. Two different types exist; Mealy machines and Moore machines, [Mealy, 1955], [Moore, 1956]. The output from a Mealy-machine depends on the internal state and the input whereas the output from a Moore machine depends only on the internal state.

In Figure 3.8 a Mealy-machine (left) and the corresponding Moore-machine (right) are shown. The two state machines have two inputs, a and b, and two outputs, z0 and z1.

A Mealy or a Moore machine can directly be transformed into a Grafcet. Moreover, it can be shown that the number of steps (and transitions)



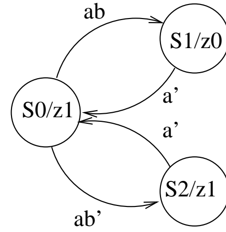
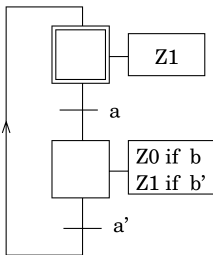


Figure 3.8 A Mealy-machine (left) and a Moore-machine (right).

required in a Grafcet is at most equal to the number of states (and transitions) required for the description by the corresponding machine. For a Moore machine the Grafcet actions become unconditional and for a Mealy machine the Grafcet actions become conditional. The Grafcets corresponding to the Mealy machine and the Moore machine in Figure 3.8 are shown in Figure 3.9.

A Grafcet may be transformed into a state machine but the number of states required in the corresponding state machine might be greater than the number of steps in the Grafcet.



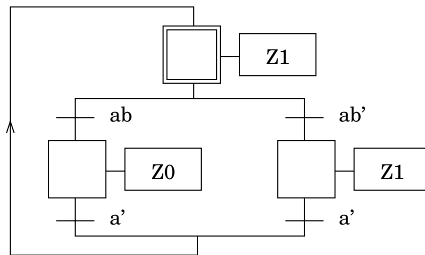


Figure 3.9 The Grafcet corresponding to the Mealy machine (left) and to the Moore machine (right) in Figure 3.8.

A state machine can be either synchronous or asynchronous. Both can be modeled by a Grafcet. However, Grafcets exist that cannot be transformed into an asynchronous nor a synchronous state machine, [David, 1995].

3.6 Summary

Grafcet was developed in France in the mid seventies. It was proposed as a formal specification and realization method for logical controllers. Grafcet has since become well known through the two international standards, IEC 848 from 1988 and IEC 1131-3 from 1993. In these standards Grafcet is refered to as Sequential Function Charts (SFC). The aim of Grafcet, or SFC, in the standards has gradually shifted from being a representation format for logical controllers towards being a graphical programming language for sequential control problems at the local level. One main advantage of Grafcet, or SFC, is its simple and intuitive graphical syntax. Today Grafcet is widely used and well accepted in industry.

4

High-Level Nets

If a system has several parts that are identical these parts should, using ordinary Petri nets or Grafcet, be modeled by as many identical nets as there are parts. This kind of problem is of no importance for small systems but it might be catastrophic for the description of a large system. Real-world systems are often large and contain many parts which are similar but not identical [Jensen, 1992]. Using ordinary nets, these parts must be represented by disjoint subnets with a nearly identical structure. This means that the total net becomes very large and it becomes difficult to see the similarities between the individual subnets representing the similar parts. To make the representation more compact, efficient, and manageable the different identical parts could be modeled by one net where each part is represented by an individual token. In order to distinguish the different tokens, i.e., the different parts, an identifier is associated with each token. This is the main idea behind all high-level nets, [Jensen and Rozenberg, 1991].

In this chapter some different high-level nets are described. The different models presented are not supposed to give a complete overview of the field but a rough idea of what has been done in the area.

4.1 Coloured Petri Nets

Coloured Petri nets exist in two main different versions. The first version was developed in 1981, [Jensen, 1981], and the second version, which is richer and more advanced, was developed in 1986, [Jensen, 1992]. In both versions, the identifier, associated with each token is called the token colour. Petri net models where the tokens can be identified, like coloured Petri nets, are also called high-level Petri nets.

Jensen compares the step from ordinary low-level Petri nets to high-level nets with the step from assembly languages to modern programming lan-

guages. In low level nets there is only one type of token which means that the state of the place is described by an integer (or by a boolean). In high-level nets, each token can carry complex information or data. The state of the system can therefore be more precise [Jensen, 1992].

EXAMPLE 4.1.1
An example taken from [David and Alla, 1992] illustrates the idea of coloured Petri nets. Figure 4.1 represents two identical systems. In each system, the truck can move either to the left or to the right. As soon as the truck reaches one end, it changes direction and sets of again towards the other end.

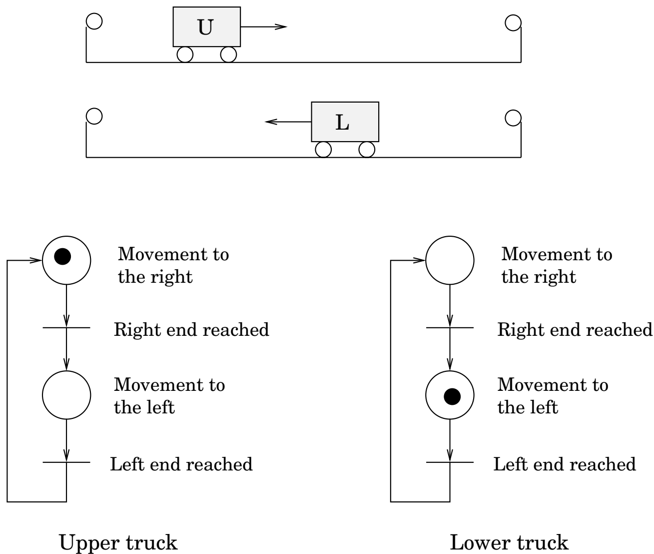


Figure 4.1 Two identical systems and their associated Petri nets.

The two systems can be modeled by two separate nets, as indicated in Figure 4.1. Each net contains one token. The two systems can also be modeled in one single net as shown in Figure 4.2. In order to distinguish the two trucks from each other, the tokens are given individual names or colours. The token corresponding to the upper truck is labeled $<U>$ and the truck corresponding to the lower truck is labeled $<L>$.

#

The operation performed when transforming an ordinary Petri net to a Coloured Petri net is called folding. The reverse operation is called unfolding.

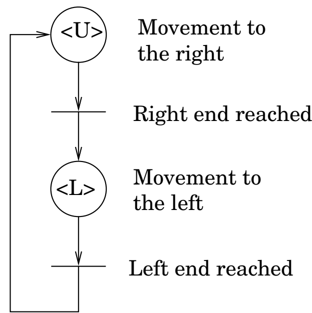


Figure 4.2 One coloured Petri net representing two identical systems.

The properties of a coloured Petri net are the same as those for an ordinary Petri net, i.e., firing sequences, bounded, live, deadlock-free, etc.

Coloured Petri Nets Version 1981

To each transition a set of firing colours is associated. These represent the different firing possibilities. A transformation of colours may occur during the firing of a transition. To be able to represent colour transformations the concept of arc inscriptions is introduced. The arc inscriptions are functions associated with the input and the output arc of a transition. The functions on the input arcs determine the numbers and the colours of the tokens that will be removed from the input places when the transition fires with respect to a certain firing colour. Similarly, the functions on the output arcs determine the colours and the number of the tokens that will be added in the output places of the transition. The identity function is associated to arcs without colour transformation. The identity function is however, very often not explicitly written out.

EXAMPLE 4.1.2
An example of a small coloured PN is given in Figure 4.3. In the figure it is shown how the colour functions, associated with the arcs, can be used.

Transition $T1$ is enabled with respect to the firing colour m since the input places contains at least $f(m)$ and $g(m)$ tokens, respectively. The transition is not enabled with respect to the firing colour b. This is due to the left input place that does not fulfill the condition of containing $f(b)$ tokens. When the transition fires with respect to the firing colour m, $f(m)$ and $g(m)$ tokens will be removed from the respective input place and $f(m)$ and $Id(m) = 1\ m$ tokens will be added to the respective output place.

#

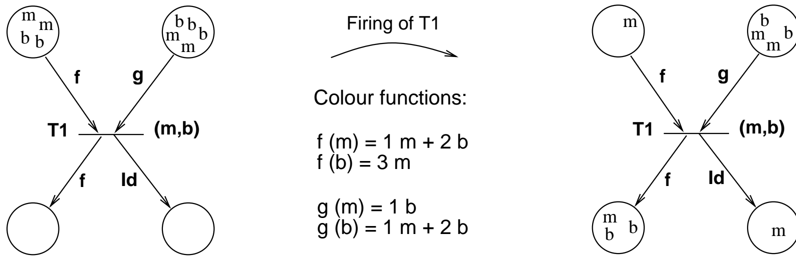


Figure 4.3 A simple coloured PN.

Complex colours consisting of two or more subcolours can be defined. This can further reduce the size of the model of a system. However, if the system is too much compactified, it becomes hard to read and understand.

EXAMPLE 4.1.3
In Figure 4.2 a single colour is used to identify each truck. The system can however be even more compactly described using complex colours. The upper and the lower truck can both move either to the left or to the right. In Figure 4.2 the two directions are represented by two separate places. If each truck instead is represented by a complex colour where the first subcolour indicates if it is the upper, U, or the lower, L, truck and the second subcolour indicates if the truck moves to the right, r, or to the left, l, the system can be described by only one place and one transition, see Figure 4.4.

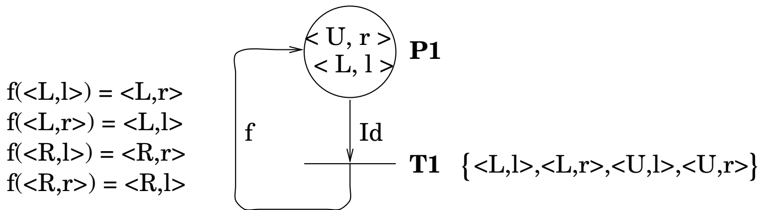


Figure 4.4 A coloured Petri net with complex colours.

#

Formal Definition

A coloured PN is a 6-tuple

$$R = < P, T, Pre, Post, M_0, C >$$

where:

- *P is the set of places.*
- *T is the set of transitions.*
- *Pre and Post are functions relating the firing colours to colours of the tokens.*
- *M_0 is the initial marking.*
- *$C = \{C_1, C_2, \dots\}$ is the set of colours.*

EXAMPLE 4.1.4
The coloured Petri net shown in Figure 4.4 can be described by the 6-tuple R.

$$R = \{P, T, Pre, Post, M_0, C\}$$

where

$$\begin{aligned} P &= \{P_1\} \\ T &= \{T_1\} \\ Pre &= [Id] \\ Post &= [f] \\ M_0 &= [<U, r> \quad <L, l>]^T \\ C &= \{<U, r>, <U, l>, <L, r>, <L, l>\} \end{aligned}$$

#

Coloured Petri Nets Version 1986

The difference between the second version of Coloured Petri nets and the first one is the representation of the arc inscriptions. Coloured Petri nets version 1981, relies on a function representation. The arc inscriptions are functions that are associated with the input and output arcs of a transition. Associated with a transition is a set of firing colors that indicate the firing possibilities of a transition. Coloured Petri nets version 1986 relies on an expression representation where arc expressions are used in

combination with guards. It has been shown that the two representations can be transformed into each other.

The token colour is a data value, that might be of arbitrarily complex type, e.g., a record where the first place is an integer and the second place a boolean variable. Each place in the net have a colour set that specifies the possible colours of tokens that might reside in this place.

The concept of guards is introduced. A guard is a boolean expression restricting the conditions under which a transition can fire. The guard must be fulfilled before the transition can fire. Each arc has an associated arc expression that determines which and how many tokens that will be affected by a transition firing. A declaration is associated with each net specifying the different colour sets and variables.

Presentations of practical applications with Coloured Petri nets can be found in [Jensen, 1997]. A formal definition of Coloured Petri nets version 1986 can be found in [Jensen, 1995].

EXAMPLE 4.1.5
A example, inspired by [Jensen, 1995], illustrates how this type of net works. In Figure 4.5 there are two types of processes, called p and q. Three q-processes start in place $p1$ and cycle through the places $(p1, p2, p3)$. Two p-processes start in place $p2$ and cycle through the places $(p2, p3)$. Each of the five processes is represented by a token, where the token colour is a pair such that the first element tells whether the token represents a p-process or a q-process and the second element is an integer telling how many full cycles that process has completed. In the initial marking there are three $(q,0)$-tokens at place $p1$ and two $(p,0)$-tokens at place $p2$. There are two different types of resources, one r-resource and three s-resources.

In the arc expressions and in the guards different variables are used. The variables are specified in the declaration that is associated with the net. The variable i is allowed to be of type I where I is a colour declared as an integer. The variable x is allowed to be of type U where U is a colour either equal to p or q. A guard might restrict the possible types of variables that can enable a transition. This is, e.g., the case for transition $T1$ where the variable x is only allowed to be of type q. The colour set, P, associated with the places, indicates the possible token colours allowed in a place. The colour P is composed of one part of type U and one part of type I. The resources are of token type E.

Transition $T1$ is enabled if there is a token of type (q,i) in place $P1$, a token of type e in place R and a token of type e in place S. When the transition fires the tokens that enabled the transition are removed and a token of type (q,i) is added in place $P2$. Transition $T3$ does not have a

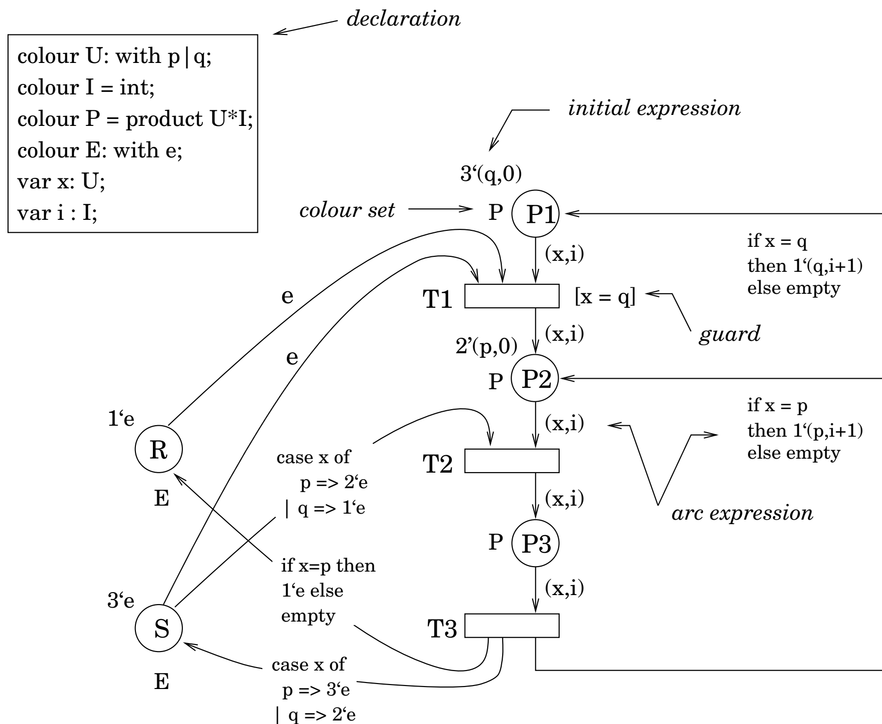


Figure 4.5 A coloured Petri net describing a resource allocation system.

guard, this means that the transition can be enabled by a token of type (p, i) or a token of type (q, i) in place $P3$. If the transition is enabled with respect to a token of type (p, i), the firing of the transition will remove a token of type (p, i) from place $P3$ and add one token of type e in place R, three tokens of type e in place S and one token of type $(p, i + 1)$ in place $P2$. If the transition instead is enabled with respect to a token of type (q, i), the firing of the transition will remove a token of type (q, i) from place $P3$ and add two tokens of type e in place S and one token of type $(q, i + 1)$ in place $P1$.

#

A toolbox, called Design/CPN, has been developed for Coloured Petri nets, [Met, 1993]. The user graphically draws the net and textually specifies the colour sets, arc expressions and guards. The inscription language of the toolbox is Standard ML, [Harper, 1986].

4.2 Coloured Grafcet

Coloured Grafcet is a graphical model similar to Grafcet. Coloured Grafcet was introduced by Agaoua in a PhD-thesis in 1987 [Agaoua, 1987]. It has also been used to realize a model for simulation and real-time control, [Suau, 1989].

Grafcet is used to control a system. In order to be able to control large systems composed of several identical, or similar, subsystems, the concept of Coloured Grafcet is introduced, i.e., the main reason for introducing Coloured Grafcet is the same as that for the introduction of Coloured Petri nets.

A Coloured Grafcet is built up by steps and transitions in the same way as Grafcet. Associated with the arcs are functions that specify the numbers and the colours of the tokens that should be removed or added to a step, the functions are called *Pre* and *Post*, respectively. A step may contain at most one token of each colour. The set of different token types is denoted C. All possible combinations of token types that may reside in a step is called C_S. Each action associated with a step is bound to a colour. When a token enters the step, the action, if any, associated with the step and with the colour of the token is effectuated. The action types are the same as those for Grafcet, see Chapter 3.1 (Actions). A transition can be enabled with respect to different colours. A transition can have several receptivities. Each receptivity is bound to a colour. The set of different receptivity types is called C_T and is a product of C and $\mathcal{R}$ where $\mathcal{R}$ is the set of conditions and events that enable a change of state. The colour-function is denoted C and is defined as $C = \{C_S \cup C_T\}$.

The new concepts in Coloured Grafcet, compared to Grafcet, are the introduction of colours and the colour-functions.

Example 4.2.1
Consider the case were there are two machines, M_1 and M_2. Each machine can be in either of two states: idle or busy. If the machine is in the state idle and the button start is pushed the machine switches state and starts to work. If the machine is in state busy and the button stop is pushed the machine stops working and becomes idle. The two machines can be modeled by two separate Grafcets, see Figure 4.6 (left), or by one Coloured Grafcet, see Figure 4.6 (right).

#

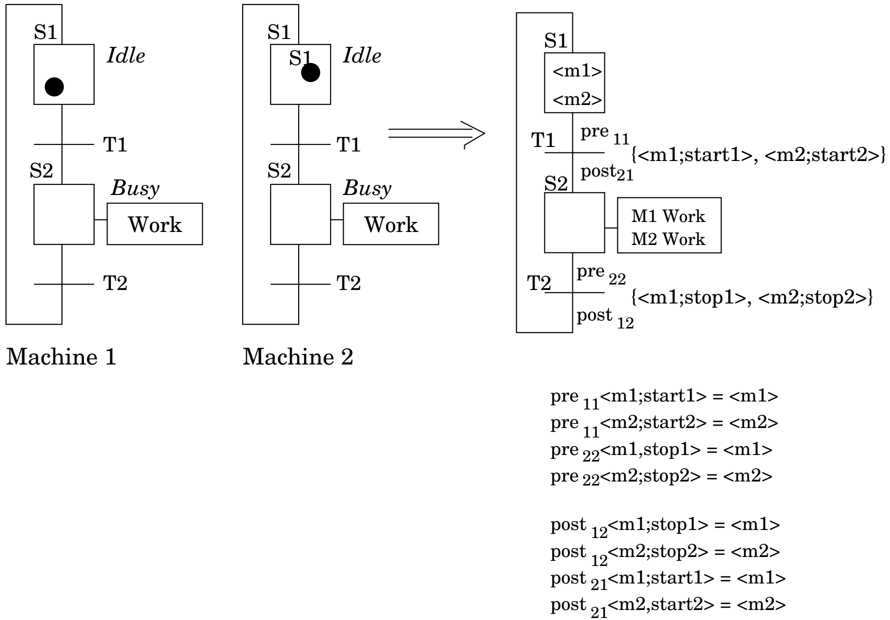


Figure 4.6 Two Grafcets transformed into a Coloured Grafcet.

Formal Definition

A Coloured Grafcet, [Agaoua, 1987], is a 6-tuple

$$CG = < S, T, C, Pre, Post, M_0 >$$

where:

- $S = \{S_1, S_2, ...S_n\}$ *is a finite, nonempty, set of steps.*
- $T = \{T_1, T_2, ..., T_m\}$ *is a finite, nonempty, set of transitions.*
- $S \cap T = \emptyset$*, i.e., the sets* S *and* T *are disjoint.*

- *C is the colour-function defined for $S \cup T$ in the non-empty set $\{CS_i \cup CT_i\}$ such that $CS_i \in C_S$ and $CT_j \in C_T$. The elements of $\{CS_i \cup CT_j\}$ are called colours.*
- *Pre is the input incidence function defined for $S \times T$ such that pre_{ij} is a function $CS_i \rightarrow CT_j$.*
- *Post is the output incidence function defined for $S \times T$ such that $post_{ij}$ is a function $CT_j \rightarrow CS_i$.*
- *M_0 is the initial marking defined for S such that $\forall S_i \in S$, $M_0(S_i) \in C_S$.*

EXAMPLE 4.2.2
The Coloured Grafcet in Figure 4.6 (right) is described by the sextuple:

$$CG = <S, T, C, Pre, Post, M_0>$$

where

$$\begin{aligned}
S &= \{S_1, S_2\} \\
T &= \{T_1, T_2\} \\
C &= \{C_S \cup C_T\} \\
C &= \{<m1>, <m2>\} \\
C_S &= \{<m1>, <m2>, <m1> + <m2>\} \\
\mathcal{R} &= \{start1, start2, stop1, stop2\} \\
C_T &= \{<m1; start1>, <m1; start2>, <m1; stop1>, <m1; stop2>, \\
&\quad <m2; start1>, <m2; start2>, <m2; stop1>, <m2; stop2>\} \\
Pre &= \begin{bmatrix} pre_{11} & 0 \\ 0 & pre_{22} \end{bmatrix} \\
Post &= \begin{bmatrix} 0 & post_{12} \\ post_{21} & 0 \end{bmatrix} \\
M_0 &= [<m1> + <m2> \quad 0]^T
\end{aligned}$$

#

4.3 Object Petri Nets and LOOPN

The difference between ordinary Petri nets (PN) and Coloured Petri nets (CPN) is the expressive comfort. A Coloured Petri net can be transformed into an ordinary PN, but the structure of the CPN is more compact and most often easier to read and understand. CPN therefore constitute a significant advance over PN, but the absence of powerful structuring primitives is still a weakness, [Jensen, 1990].

Object Petri nets are a class of Petri nets where object-oriented ideas are applied to Coloured Petri nets, [Lakos, 1994]. Two class hierarchies exists: one for tokens and one for subnets or modules. So far, the object Petri nets have only focused on object-oriented structuring of the token types. The components of a class are drawn within a frame marked with the class name. A place is a data field which can supply values. All data fields are drawn as circles. Transitions and functions are drawn as rectangles. The export of fields and functions from a class is indicated by an undirected arc from the object to the class boundary. A class can be made a subclass of another class and thereby inheriting from it. The Object Petri nets have retained the traditional Petri net style with tokens as passive data items and their life cycles specified by the global control structure of the net. This means that, unlike ordinary object-oriented programming, it is not possible to let the call to a function, or method, depend on the type of data within the token and it is neither possible to affect the response of an operation. The Object Petri nets can be proven to be behaviorally equivalent to Coloured Petri nets and thus also to ordinary Petri nets. This means that the analysis techniques developed for PN can be applied.

Object Petri nets provides a simple way to model multi-level systems, i.e., systems where the components that move through the system have their own internal life cycles. Real-life situations generally have a number of layers of data activity, [Lakos, 1994]. For example, in modeling a traffic intersection, the cars which move through the intersection can be considered as data objects. The cars also have internal activities such as consumption of petrol, mechanical failure, etc, which may be of interest in the simulation.

EXAMPLE 4.3.1
A small example will illustrate how the OBJSA nets work. Figure 4.7 illustrates a truck that can move either to the left or to the right. The truck is driven by driver. Inside the truck there is food. As soon as the driver becomes hungry, he stops the truck and starts to eat, when he is no longer hungry he continues to drive.

The token type for a driver is called Driver and is shown in Figure 4.8

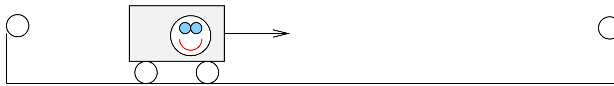


Figure 4.7 A truck and a driver.

(left). The driver contains data, e.g., an identity number given by an integer. The state of the driver is described by the net named DriverAction shown in Figure 4.8 (middle). The net has two functions, Drive and Eat, that are exported. The two functions depend on the marking of place $P1$ and $P2$ respectively, as shown by a special arc, known as the compound arc. This arc has the effect of binding the variable y to the marking of $P1$ or $P2$ and then the function returns the boolean value $y = 1$. The token moving around in this net is of the type Driver.

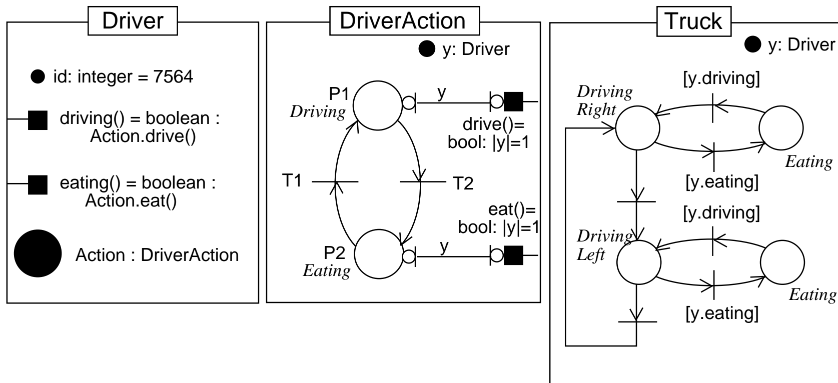


Figure 4.8 Class definitions for Driver, DriverAction and Truck.

The net describing the truck is shown in Figure 4.8 (right). The token moving around in this net is of type Driver. Associated with the transitions are receptivities from which it is possible to check the data of the token.

#

Modeling a multi-level system with CPN will often result in a large and complicated net whereas the modeling with OPN will be easier and the resulting structure of the net becomes easy to read and understand.

The formal definition of OPN as well as the relation to CPN are given in [Lakos, 1994]. LOOPN++, a textual language for Object Petri nets is defined in [Lakos and Keen, 1994].

4.4 OBJSA Nets

High-level nets all have individual tokens. In the OBJSA nets this is combined with algebraic specification techniques, [Battiston *et al.*, 1988]. OBJSA net systems is a class of high level Petri nets. The OBJSA nets can be decomposed into state-machine components. The individual tokens are defined as abstract data types. To modify the data a language called OBJ2 or OBJ3 is used.

EXAMPLE 4.4.1
A small example will demonstrate the idea behind the OBJSA nets. The example is a modified version of an example first published in [Battiston *et al.*, 1988].

Consider a system that consists of one sender (S) and one receiver (R). The receiver has a mailbox (M) consisting of a one-cell buffer where the sender asynchronously put messages. The receiver asynchronously reads the messages and removes them from the mailbox. The system can be modeled by an ordinary PN or by a coloured PN (CPN). In the CPN the tokens carry an identifier permitting the tokens to be distinguished from each other. The messages from a sender can be represented by the token $<msg>$ where *msg* is the message to be sent. The tokens representing the messages in the mailbox is either of the type empty $<->$ or of the type message $<msg>$.

However, usually mailboxes are not one-cell buffers but list of messages. This can conveniently be modeled using algebraic nets, i.e., nets where the information associated with the token is algebraically specified and algebraically modified. Figure 4.9 shows the sender and the mailbox represented by an algebraic net. The token representing the mailbox is now a list, either empty $<eL>$ or nonempty $<L>$. When a new message comes the mailbox modifies its list of messages by placing the new message in the end of the old list. This operation is performed by the @-operator in the OBJ3 language. There exists a large number of operators defined in the OBJ3 language, e.g., $tail(L)$, an operator that returns all elements in the list L but the first.

#

A formal definition a more thoroughly presentation of the OBJSA nets and the OBJ2 and OBJ3 languages can be found in [Battiston *et al.*, 1988].

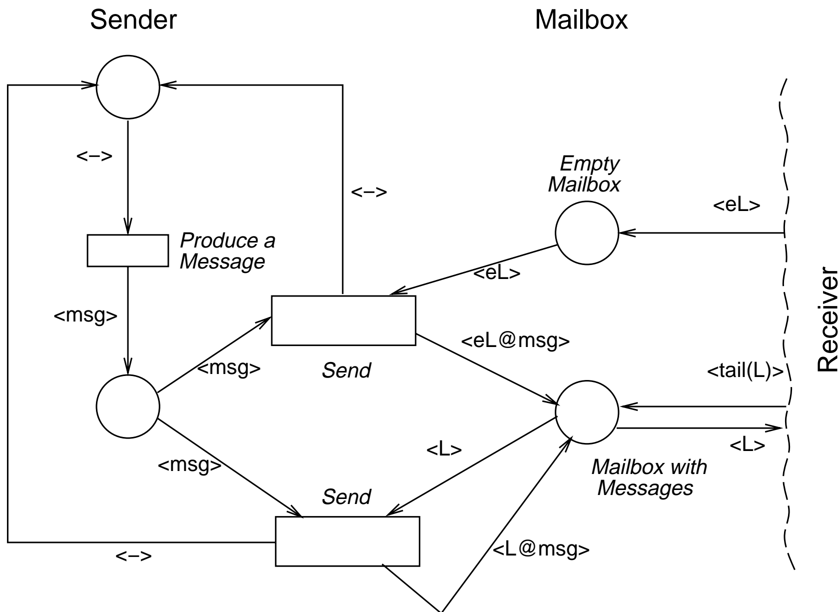


Figure 4.9 A sender and a receiver represented by an algebraic net.

4.5 Other High-Level Languages Influenced by Petri Nets

Petri nets have also influenced some other high-level programming languages.

Cooperative Objects

Cooperative Objects is an object-oriented language influenced by the language Eiffel and by high-level Petri nets, [Bastide *et al.*, 1993]. Cooperative Objects aims at modeling a system at various abstraction levels as a collection of concurrent objects. The objects cooperate in a well defined way to provide the services expected from the system.

In this language concurrency, object behavior and inter-object communication are described in terms of high-level Petri nets. The language relies on a client-server organization of objects and it retains the most important features in object-oriented languages: classification, encapsulation, inheritance, instantiation and dynamic use relationship.

In Eiffel, the contract that the server object should fulfill for its client is expressed by a set of preconditions, postconditions and invariants. In Cooperative Objects, the contract is expressed by a high-level Petri net.

Also the implementation of a class is given by a high-level Petri net as well as the semantics of a service invocation.

The language can be used both for specification of a concurrent class and for implementation. Results from the PN theory can be used to ensure the compatibility between a specification and its implementation.

Moby

MOBY (MOdellierung von BürosYstemen) is a tool which supports modeling and analysis of systems by hierarchical timed high level Petri nets with objects, [Fleischhack and Lichtblau, 1993]. The tool should assist in the modeling, analysis and simulation of processes in various application areas. The tool consists of editors for the specification of the net part, and of a simulator for the validation of constructed models. The simulator contains a conflict resolution component, which reduces the combinatorial complexity of possible behaviors of a model by using specific knowledge about the system, [Fleischhack and Lichtblau, 1993].

The nets considered in MOBY combines features from algebraic high level nets and coloured Petri nets with concepts of time and hierarchy. The tokens are structured in an object oriented manner. A transition may be refined and the can be seen as a kind of macro expansion. From a transitions subnets can be called.

To each place in an object net one of the four types: multiset, stack, queue or priority queue, is assigned. This means that the objects in a place are handled according to the rules of the corresponding data type. Multiset places may contain tokens belonging to different types whereas other places are bound to one token type.

Arcs are either of standard type or of inhibitor type. In order to activate a transition, inhibitor arcs require that its input place does not contain certain objects. The arcs can be either activating or consuming. Activating arc behave like standard ones as far as the activation is concerned, however, the firing of the transition does not remove objects from the place. Consuming arcs enable a transition to empty a place.

GINA

Gina is an object-oriented language concept for parallel, hierarchically structured, data driven programs. It is also a Petri net language based on dynamically modified, interpreted nets, [Sonnenschein, 1993].

4.6 Summary

Real-world systems are often very large and do often contain many parts which are similar but not identical. Using ordinary low level nets, these parts must be represented by disjoint subnets with nearly identical structures. This means that the total net becomes very large and it becomes difficult to see the similarities between the individual subnets representing similar parts. To make the representation more compact, efficient and manageable the different identical parts could be modeled by one net where each part is represented by an individual token. To be able to distinguish the different tokens, i.e., the different parts, an identifier is associated with each token.

5

Grafchart

Grafcet, or Sequential Function Charts (SFC), has been widely accepted in industry as a representation format for sequential control logic at the local level through the standards IEC 848 and IEC 1131-3, as described in Chapter 3. There is, however, also a need for a common representation format for the sequential elements at the supervisory control level. Supervisory control applications receive increasing attention both from the academic control community and the industry. The reasons for this are the increasing demands on performance, flexibility, and safety caused by increased quality awareness, environmental regulations and customer-driven production.

Sequential elements show up in two different situations in supervisory control. The first situation arises due to the fact that the processes in the process industry are typically of a combined continuous and sequential nature. All processes have different operation modes. In the simplest case these can consist of start-up, production and shut-down. The second situation concerns the case when the problem that the supervisory system should solve itself can be decomposed into sequential steps.

Grafcet was developed for logic controllers. It is a mathematical model whose semantics is defined by an interpretation algorithm. Since Grafcet was developed for sequential control logic at the local level it lacks many features needed to structure and implement the complex applications that are found on the supervisory control level.

Grafchart is the name of a mathematical sequential control model. It is based on Grafcet, Petri nets and concepts from object-oriented programming. Grafchart is aimed, not only at local level applications, but also at supervisory level applications. It has been developed at Lund Institute of Technology, Sweden, since 1991, [Årzén, 1991], [Årzén, 1994b], [Årzén, 1993]. Grafchart exists in two different versions. The first and basic version of Grafchart is mainly based on Grafcet whereas the second and

high-level version also incorporates ideas from high-level Petri nets.

Grafchart is also the name of an implementation of the Grafchart model in G2, an object-oriented graphical programming environment. G2 is briefly described in [Gensym Coorporation, 1995], see Appendix E. With Grafchart the same language can be used both on the local control level and on the supervisory control level, see Figure 5.1.

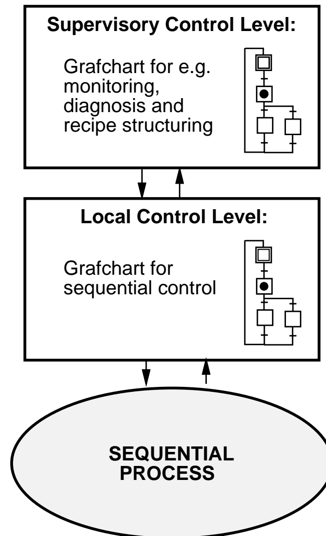


Figure 5.1 Supervision of sequential processes.

The language of Grafchart is, like any other programming language, fully defined by its syntax, semantics and pragmatics, [Turbak *et al.*, 1995]. The syntax defines the notations used, the semantics defines the meaning of the syntax, and the pragmatics focuses upon how the semantics is implemented. If the pragmatics is omitted the language is reduced to a model.

This chapter starts with a presentation of the syntax of Grafchart, i.e., its graphical elements and its step actions and transition receptivities. Then follows a presentation of the basic version (Grafchart-BV) and the high-level version (Grafchart-HLV) of the language. The presentation covers, among other things, the dynamic behavior, i.e., the semantics, of Grafchart-BV and Grafchart-HLV. A section in the chapter covers the error handling aspect and another section describes the G2 implementation, i.e, the pragmatics of Grafchart. The chapter ends with a short presentation of some applications of Grafchart.

5.1 Graphical Language Elements

The graphical syntax of Grafchart is similar to that of Grafcet. It supports alternative branches and parallel branches. The graphical language elements of Grafchart are steps, transitions, macro steps, procedure steps, process steps, Grafchart procedures and Grafchart processes. Each element is represented by an object. The icon of this object defines the graphical presentation of the element. The elements are interconnected using graphical connections.

Grafchart processes

An entire function chart can be represented as a Grafchart process object, see Figure 5.2. The function chart is encapsulated by the Grafchart process object. In the G2 implementation the function chart is placed on the subworkspace, see Chapter 5.9, of the Grafchart process.

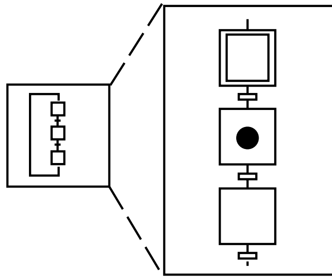


Figure 5.2 A Grafchart process.

A Grafchart can be closed or open, as shown in Figure 5.3. However, the function chart always starts with one or several initial steps.

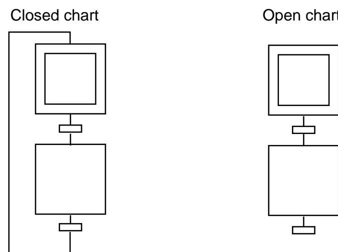


Figure 5.3 One closed and one open function chart.

Steps

A step is an object that is represented as a square, see Figure 5.4. Actions, i.e., can be associated with a step. A step can be active or inactive.

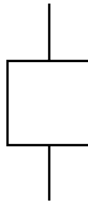


Figure 5.4 A step.

Initial steps

The graphical representation of an initial step, i.e. a step that should be active when the function chart is started, is a double square, see Figure 5.5.

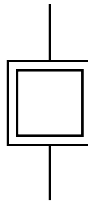


Figure 5.5 An initial step.

Transitions

A transition is represented by an object. The graphical representation of a transition is shown in Figure 5.6. A transition can be enabled or disenabled.

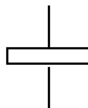


Figure 5.6 A transition.

Parallel bars

A parallel bar is used to indicate the beginning and the end of a parallel branch. To indicate the beginning of such a branch a parallel bar of type

and-divergence (parallel split) is used and to indicate the end a parallel bar of type and-convergence (parallel join) is used, see Figure 5.7.

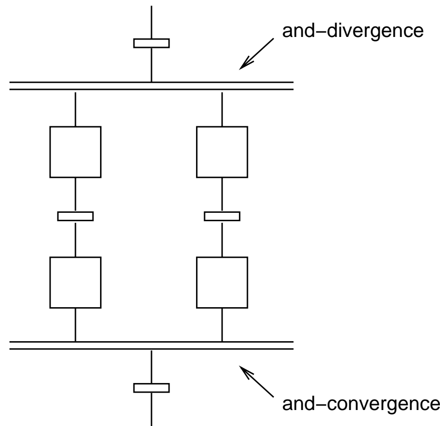


Figure 5.7 A parallel branch.

Macro steps

Macro steps are used to represent steps that have an internal structure of (sub)steps, transitions and macro steps. In the G2 implementation a macro step is represented by an object that has a subworkspace. The internal structure is placed on the subworkspace. In Figure 5.8, a macro step and its internal structure are shown.

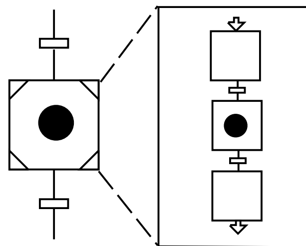


Figure 5.8 A macro step.

Enter steps and Exit steps

Special step objects, called enter-steps and exit-steps, are used to indicate the first and the last sub-step of a macro step. When the transition preceding a macro step becomes true, the enter-step upon the subworkspace of the macro step and all the transitions following that enter-step are

enabled. The transitions following a macro step will not become enabled until the execution of the macro step has reached its exit-step. An enter step and an exit step are shown in Figure 5.9.

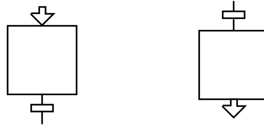


Figure 5.9 An enter step (left) and an exit step (right).

Grafchart procedures

Sequences that are executed in more than one place in a function chart can be represented as Grafchart procedures, see Figure 5.10. The Grafchart procedure object has a subworkspace on which the procedure body is placed. The enter-step and exit-step objects are used to indicate the first and the last sub-step of a procedure. Only one enter step is allowed in each Grafchart procedure.

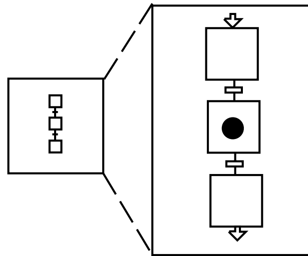


Figure 5.10 A Grafchart procedure and its subworkspace.

The Grafchart procedures are reentrant. Each procedure invocation executes in its own local copy of the procedure body. This makes recursive procedure calls possible.

Procedure steps

A Grafchart procedure is called from a procedure step. The procedure step has a `procedure` attribute in which the name of the Grafchart procedure that should be called is specified. Immediately when a procedure step is activated, the Grafchart procedure is called and a new token is placed in the enter step of the Grafchart procedure. The transitions following a procedure step do not become enabled until the execution of the Grafchart procedure has reached its exit step. In Figure 5.11 a procedure step is shown.

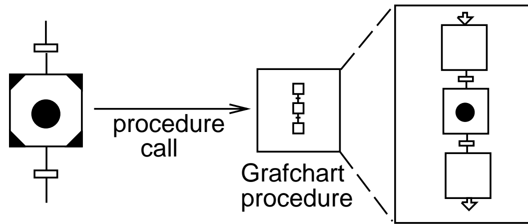


Figure 5.11 A Procedure step.

Process steps

A procedure step is the equivalent of a procedure call in an ordinary programming language. Sometimes it is useful to start a procedure as a separate execution thread, i.e., to start the procedure as a separate process. This is possible with the process step, see Figure 5.12. The transitions after a process step become enabled as soon as the execution has started in the Grafchart procedure. An outlined circle token is shown in the process step as long as the process is executing. It is possible to have more than one process executing at the same time from the same process step.

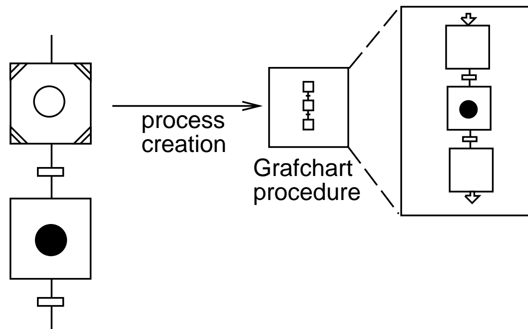


Figure 5.12 A Process step.

Tokens

Tokens are represented as filled circles. In the basic version of Grafchart, the tokens are boolean indicators indicating if a step is active or not, whereas the tokens in the high-level version of Grafchart are objects that have an identity and may contain information.

5.2 Actions and Receptivities

Actions are associated with steps and receptivities are associated with transitions. The two versions of Grafchart differs with respect to when a step is considered active and when a transition is considered enabled, see Chapter 5.4 and Chapter 5.5 respectively.

Actions

Four different types of actions exist: always, initially, finally and abortive. An initially action is executed once immediately when the step becomes active. A finally action is executed once immediately before the step is deactivated. An always action is executed periodically while the step is active. An abortive action is executed once immediately before the step is aborted. All actions can be conditional or unconditional. The actions that may be performed in a step action depends on the underlying implementation language. The minimum requirement is that it should be possible to assign values to variables and that it should be possible to execute macro actions, see Chapter 5.9.

In Figure 5.13 a step, x1, is shown. Three actions are associated with the step. The figure (right) shows the execution of the actions.

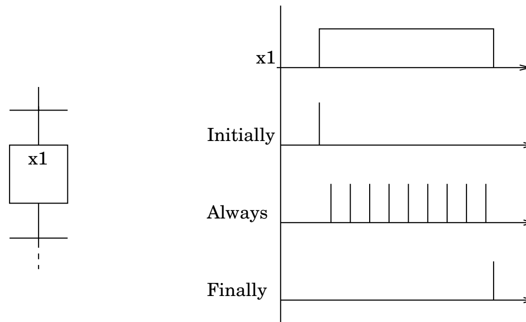


Figure 5.13 A step with three actions.

Actions can also be associated to macro steps and Grafchart procedures. In this case, the initially actions are executed before the actions of the enter step of the macro step or the Grafchart procedure and the finally actions are executed after the actions of the exit step of the macro step or the Grafchart procedure. Always actions are executed periodically all the time while the macro step or Grafchart procedure is active. Abortive actions that are associated with a macro step, are executed if the execution of the macro step is aborted. Abortive actions can also be associated with

a Grafchart procedure and they are executed if the Grafchart procedure is aborted.

Receptivities

A receptivity is used for specifying when an active step should fire. Each receptivity contains two attributes, `condition` and `event`. These are used to enter the event expression and/or the logical condition telling when the transition should fire. The event expression and the logical condition are expressed in the underlying implementation language.

In Figure 5.14 two examples are given of how the event or condition of the transition receptivity effects the activation and deactivation of the two steps `x1` and `x2`.

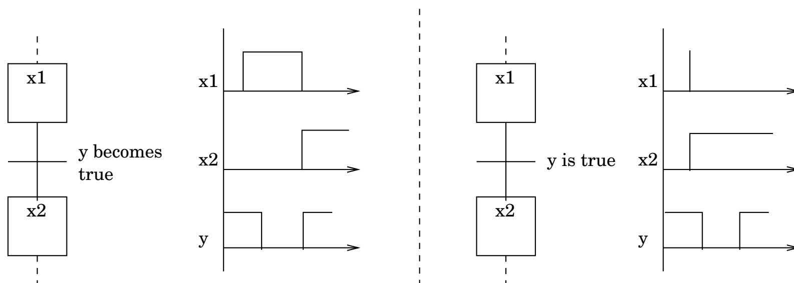


Figure 5.14 A transition with a receptivity.

5.3 Parameterization and Methods

Grafchart includes two advanced features: parameterization and methods and message passing.

Parameterization

Parameterization denotes the possibility for a graphical element to have attributes acting as parameters. Grafchart processes (i.e., entire function charts), Grafchart procedures, macro steps, and steps can be parameterized. The parameterization feature makes it easier to reuse function charts from one application to another.

A step is allowed to have none, one or several attributes. These attributes act as parameters that can be referenced and modified from within the step actions associated to this step.

The attributes of a Grafchart process, Grafchart procedure, or a macro step can be referenced from within all steps and all transitions placed

on the subworkspace of the graphical element or from within a step or transition placed deeper in the subworkspace-hierarchy.

To refer to a parameter, a Pascal-like dot notation is used.

- `sup.attribute1`
  refers to the `attribute1` attribute visible in the current context.
- `sup.attribute1^`
  refers to the object named by the `attribute1` attribute visible in the current context. This notation is the G2 equivalent of a pointer attribute in a conventional programming language.
- `sup.attribute1^.attribute2`
  refers to the `attribute2` attribute of the object given by the `attribute1` attribute visible in the current context.

Lexical scoping is used when searching for the attribute of an object replacing the `sup` notation. `Sup` is short for superior.

Methods and Message Passing

Methods and message passing are supported by allowing Grafchart procedures to be methods of general G2 objects. For example, an object representing a batch reactor can have Grafchart methods for charging, discharging, agitating, heating etc. Inside the method body, it is possible to reference the object itself and the attributes of this object using a Smalltalk influenced notation.

- `self`
  refers to the object that the method belongs to, i.e., this object.
- `self.attribute1`
  refers to the `attribute1` attribute of this object.
- `self.attribute1^`
  refers to the object named by `attribute1` of this object.
- `self.attribute1^.attribute2`
  refers to the `attribute2` attribute of the object named by the `attribute1` attribute of this object.

References to attributes of the object that the method belongs to can be combined with parameter references using the `sup.attribute` notation.

5.4 Grafchart – *the basic version*

The basic version of Grafchart (Grafchart-BV) is mainly influenced by Grafcet, both what concerns the syntax and the semantics. In this version of Grafchart the tokens are simple boolean indicators indicating whether a step is active or not. An active step is indicated by the presence of a token in the step. In Figure 5.15 an active (left) and an inactive step (right) are shown. Each Grafchart function chart can contain at most one token, except in an and-divergence situation where there is one token in each branch.

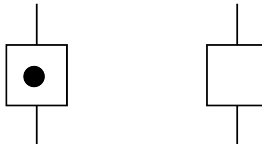


Figure 5.15 An active step (left) and an inactive step (right).

Associated with a step is none, one or several actions. In Figure 5.16 an active step is shown, associated with the step are two actions, one `initially` action and one `always` action.

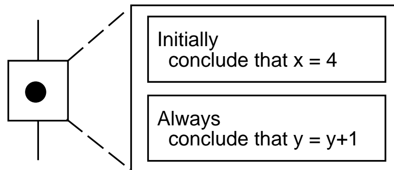


Figure 5.16 A step with an action.

Associated with a transition is exactly one receptivity, see Figure 5.17.

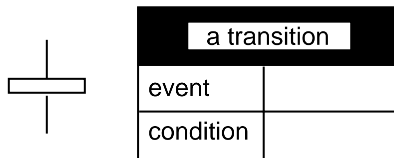


Figure 5.17 A transition with a receptivity.

Associated with the procedure steps and the process steps is a `procedure` attribute that is used to specify the name of the Grafchart procedure that should be called. When the procedure step or the process step becomes

active the corresponding Grafchart procedure is started. In Figure 5.18 a procedure step (left) and a process step (right) with their `procedure` attributes are shown.

Figure 5.18 A transition with a receptivity.

An exception transition is a special type of transition that may only be connected to macro steps and procedure steps. An ordinary transition connected after a macro step or procedure step will not become enabled until the execution has reached an exit step. An exception transition is enabled all the time that the macro step or procedure step is active. If the exception transition condition becomes true or the exception transition event occurs, the exception transition will fire, abortive actions, if any, will be executed, and the step following the exception transition will become active. Exception transitions cannot be connected to a process step. The reason for this is that a Grafchart procedure can be started more than once from such a step and it is therefore not clear which one of the procedures that should be aborted.

Macro steps and procedure steps remember their execution state from the time they were aborted and it is possible to resume them in that state. This is done using the resume macro-action, see Chapter 5.6. A macro step with an exception transition is shown in Figure 5.19. Exception transitions were first proposed in [Årzén, 1991]. They have proved to be very useful when implementing error handling.

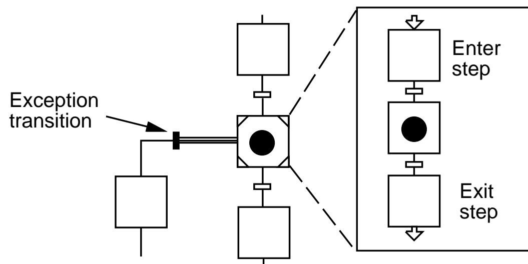


Figure 5.19 Macro step with exception transition.

Parameterization

Consider the example shown in Figure 5.20. In the figure a macro step with two attributes, `tank` and `limit`, is shown. The macro step is named FT1 and it contains the logic for the control and monitoring of the filling of a tank. The `tank` attribute contains the name of the tank that should be filled, i.e., the value of this attribute acts as a pointer. The `limit` attribute contains the limit up to which the tank should be filled. The macro step has an enter-step that contains an action that initiates the filling. In this action a reference is made to the value of the tank attribute, i.e., the tank-12 object , using the notation `sup.tank^`. Similarly the transition condition refers to the level of the tank referenced by the tank attribute and to the value of the limit attribute. The `sup.attribute` notation is translated and replaced by the corresponding G2 expression during compilation as described in Chapter 5.9.

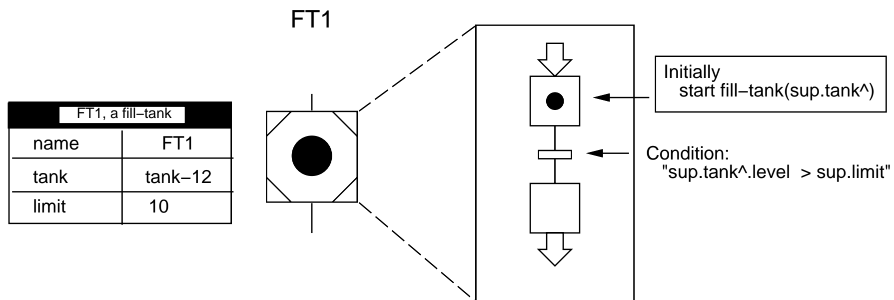


Figure 5.20 Parameterization of a macro step.

Lexical scoping is used when searching for the attribute of an object replacing the `sup` notation. Consider the example shown in Figure 5.21. The macro step M1 has an attribute named `x` with value 156. The substructure of the macro step contains among other a transition and another macro step, M2. The macro step M2 has an attribute named `x` with value 12. The substructure of M2 contains a transition with a reference to an *x*-attribute using the `sup.X` notation. This reference will be to the *x*-attribute of the macro step M2. The transition placed on the subworkspace of the macro step M1 also refers to an `x`-attribute using the `sup.X` notation. In this case the reference will be to the `x`-attribute of M1, see Figure 5.21. If, however, the macro step M2 would not have had an attribute named `x`, the `sup.X` reference of the transition in M2 would instead have refered to the `x`-attribute of M1.

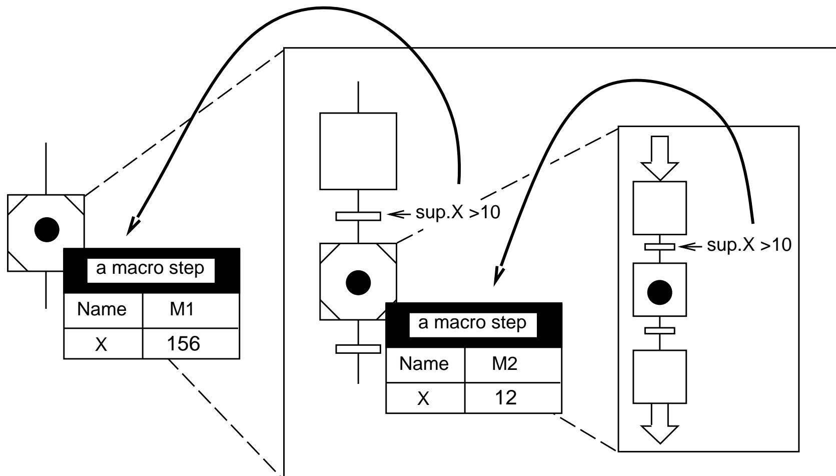


Figure 5.21 Lexical scoping.

A Grafchart procedure is started from a process step or a procedure step. The actual values of the parameters of a Grafchart procedure are set in the procedure step or the process step. In addition to the `procedure` attribute, the procedure steps and process steps also have an attribute named `parameters` which contains a list of assignments to the formal parameters of the Grafchart procedures being called, see Figure 5.22.

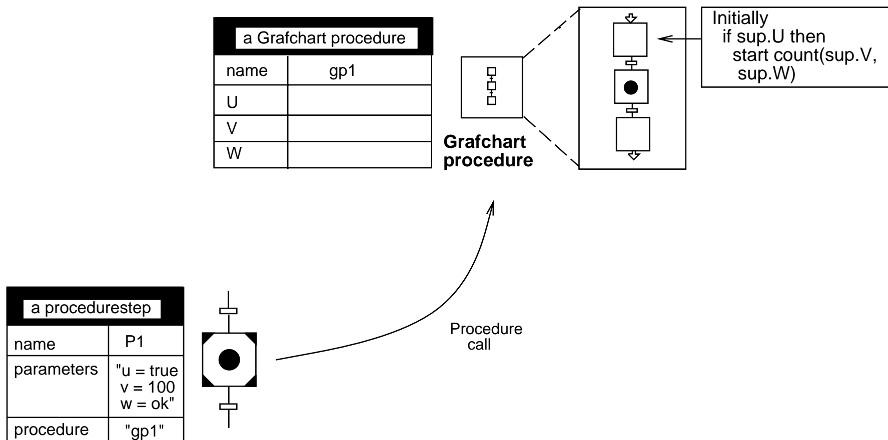


Figure 5.22 Parameterization of a Grafchart procedure.

The assigned values can either be constants (numbers, strings or symbols), as in Figure 5.22 or the value of a parameter that is visible in the process or procedure step context, as in Figure 5.23. Using the latter form, it is also possible for a Grafchart procedure to return values to the process or procedure step. In order to specify the direction in which the parameter is passed, one of the keywords `IN`, `OUT` or `INOUT` is added after the value of the parameter. It is also possible to determine which Grafchart procedure that should be called from the value of a parameter. In Figure 5.23 the parameters `val`, `v` and `q` are all visible in the context of the procedure step. The parameter `proc` is also a parameter visible in the context of the procedure step, this parameter determines the Grafchart procedure that should be called. When the Grafchart procedure is called `U` is assigned the value of `val` and `W` is assigned the value of `q`. When the execution of the Grafchart procedure ends, `v` is assigned the value of `V`, and `q`, is assigned the value of W.

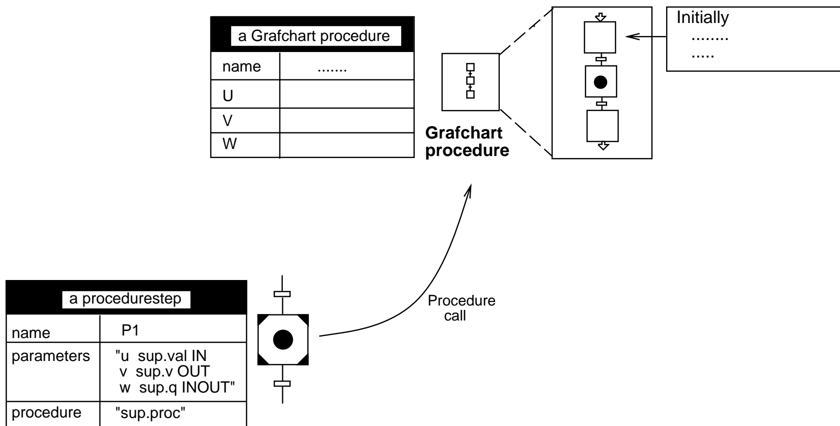


Figure 5.23 Parameterization of a Grafchart procedure.

Methods and Message Passing

The method of an object is called through a procedure or process step in the same way as if the procedure was stand-alone. Instead of giving a procedure reference, the procedure that will be called is determined by an object reference and a method reference. An example of a Grafchart method is shown in Figure 5.24. The reactor object `R1` contains the method `charge`. The method is implemented by the Grafchart procedure `reactor-charge`. The procedure step invokes the `charge` method of the `R1` object. The object and method references can also be determined by parameters.

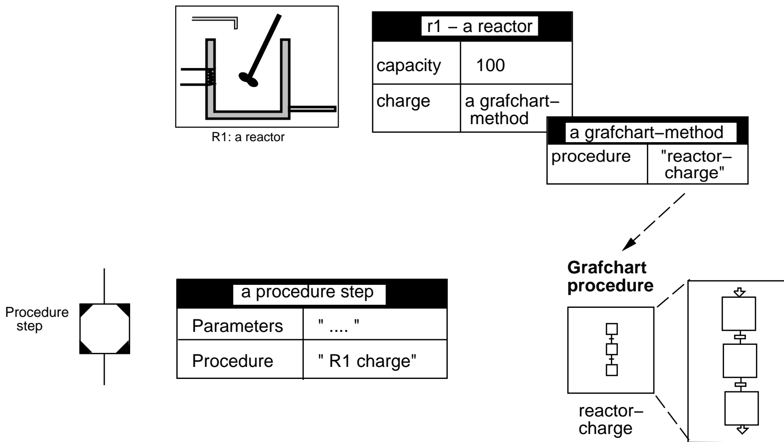


Figure 5.24 Grafchart methods.

EXAMPLE 5.4.1
Consider the tank example in given in Figure 5.25. This is the same tank as the one in Chapter 3.2 Example 3.2.1.

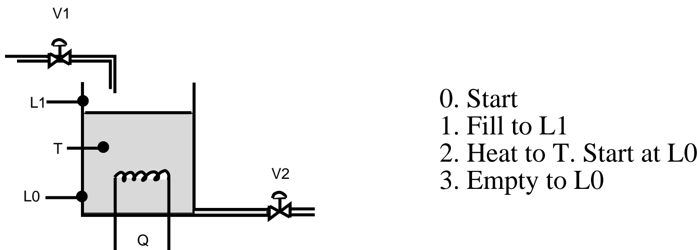


Figure 5.25 A tank example.

The Grafchart for controlling the tank is represented by a Grafchart process with attributes, see Figure 5.26. Each transition in the function chart is given a name. When clicking on the transition, the corresponding receptivity shows up. In Figure 5.26 all receptivities are placed on the right side of the chart. The structure of the function chart is similar to the Grafcet in Example 3.2.1, the difference being the syntax of the actions and the receptivities.

The behavior of the system controlled by the Grafchart can easily be modified. By, e.g., changing the value of the T attribute of the Grafchart process, the temperature to which the tank should be heated is changed.

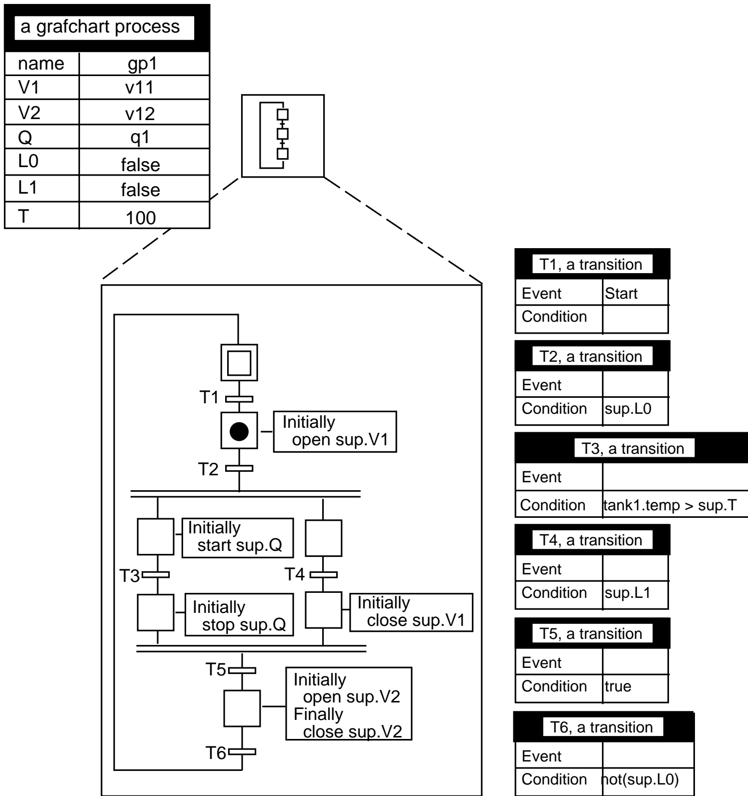


Figure 5.26 A tank example.

Dynamic Behavior

The dynamic behavior of Grafchart is given by the following rules.

Enabled and fireable transitions

A transition is enabled if and only if

- All the steps preceding the transition are `deactiveable`.

An transition is fireable if and only if

- The transition is enabled.
- The receptivity of the transition is true.

A step, initial step, enter step, exit step, or process step is deactiveable if and only if it is active.

A macro step is deactiveable if and only if the exit step upon the subworkspace of the macro step is active.

A procedure step is deactiveable if and only if the exit step in the Grafchart procedure started by the procedure step is active.

An exception transition is fireable if and only if

- The macro step or the procedure step preceding the exception transition is active.
- The receptivity of the exception transition is true.

A macro step is active if at least one of the steps upon the subworkspace of the macro step is active.

A procedure step is active if at least one of the step in the Grafchart procedure started by the procedure step is active.

A process step is active if at least one of the step in the Grafchart procedure started by the procedure step is active.

Firing of a transition The firing of a transition consists of deactivating the steps preceding the transition, de-enabling all transitions following the preceding steps, activating the steps succeeding the transition and enabling the transitions following the succeeding steps.

Firing rules

- Rule 1: All fireable transitions are immediately fired.
- Rule 2: Several simultaneously fireable transitions are simultaneously fired.
  The rule applies in all situations except for the or-divergence situation where Grafchart treats the situation by nondeterministically choosing one of the branches. In order to have an interpretation algorithm without ambiguities, situations like this must be avoided. It is therefore required, for Grafchart as in the IEC 1131-3 standard, that the user makes all branches of an or-divergence situation mutually exclusive.
- Rule 3: When a step must be simultaneously activated and deactivated, it is first deactivated and then activated again.

Interpretation Algorithm

The interpretation algorithm of Grafchart describes the theoretical understanding of a Grafchart and guarantees that everyone faced with the same Grafchart understands it in the same way.

As in the case of Grafcet, multiple interpretation algorithms or semantics are possible for Grafchart. The following algorithm corresponds to the search for stability algorithm for Grafcet, see Chapter 3.2 (Interpretation Algorithm). Step 4 - 18 are assumed to take zero time. In order for the algorithm to be deterministic, all or-divergence situations must be made mutually exclusive. The algorithm presented below is not deterministic unless this condition is fulfilled.

Algorithm

1. Initialization. Activate the initial steps and execute the associated initially actions. Go to Step 18.
2. Wait for a new event to occur or a condition to be true. It might be necessary to start or stop the execution of the always actions associated with the active steps since the conditional part of the always action might change.
3. When a new event occurs or a condition becomes true, determine the set of fireable transitions T. If T is empty go to Step 2.
4. Determine the set T2, which is the set of exception transitions contained in the set T, $T2 \subseteq T$. Let T1 be the set of all transitions contained in the set T, except the exception transitions, i.e. $T1=T\setminus T2$.
5. If T1 is empty and T2 is non-empty, go to Step 6.
   If T2 is empty and T1 is non-empty, go to Step 12.
   If both T1 and T2 are non-empty, go to Step 6 or Step 12.
6. Make sure that T2 contains no transitions that are in conflict with each other, i.e., transitions participating in the same or-divergence situation. If so, the number of transitions contained in T2 must be reduced. This is done by excluding exceptions transitions participating in the conflict until T2 no longer contains any exception transitions that are in conflict with each other. The exception transitions to exclude are chosen randomly.
7. Stop executing the always actions that are still executing and that belong to the steps (or their activated substeps) preceding the transitions in T2.

8. Carry out the abortive actions associated with the steps (and their substeps) preceding the transitions in T2.
9. Fire the transitions in T2.
10. Carry out the initially actions of the steps (and their activated substeps) succeeding the transitions in T2.
11. Let T1 be the set of transitions contained in T1 that are still enabled. If T1 is empty, go to Step 18.
12. Make sure that T1 contains no transitions that are in conflict with each other, i.e., transitions participating in the same or-divergence situation. If so, the number of transitions contained in T1 must be reduced. This is done by excluding transitions participating in the conflict until T1 no longer contains any transitions that are in conflict with each other. The transitions to exclude are chosen randomly.
13. Stop executing the always actions that are still executing and that belong to the steps (or their activated substeps) preceding the transitions in T1.
14. Carry out the finally actions of the steps (and their activated substeps) preceding the transitions in T1.
15. Fire the transitions in T1.
16. Carry out the initially actions of the steps (and their activated substeps) succeeding the transitions in T1.
17. Let T2 be the set of transitions contained in T2 and that are still enabled. If T2 is non-empty, go to Step 6.
18. Determine the set of fireable transitions and exception transitions T. If T is non empty go to Step 4.
19. A stable situation is reached. Start executing the always actions associated with the active steps (and their activated substeps). Go to step 2.

Formal Definition

The formal definition of the basic version of Grafchart is presented together with an illustrative example in Appendix C.

5.5 Grafchart – *the high-level version*

The high-level version of Grafchart is an extended version of Grafchart-BV that incorporates ideas from Petri nets, coloured Petri nets and object-oriented programming. In the basic version of Grafchart, a token is simply a boolean indicator indicating whether a step is active or not. Only one token is allowed to reside in the chart (except in parallel situations). In the high-level version of Grafchart the tokens are objects with an identity. The tokens are allowed to have attributes and methods. More than one token is allowed in a function chart and in a step. The tokens may be of the same or of different type. To indicate if a token is present in a step or not, a grafchart-marker is used. The grafchart marker appears as a black or coloured filled circle and it contains a reference to the 'real' object token. The reason for introducing a grafchart marker that refers to the ' real' object token containing the attributes and methods and not letting the grafchart marker itself contain the information has to do with the way parallel structures are handled, see Chapter 5.5 (parallelism).

In Figure 5.27 a step containing a grafchart marker of type A is shown. The grafchart marker contains a reference to an object token. The object token contains attributes. The attributes of an object token can be referenced from the actions of the step that the corresponding grafchart marker is placed in and from the receptivities of the transitions that the grafchart marker is currently enabling.

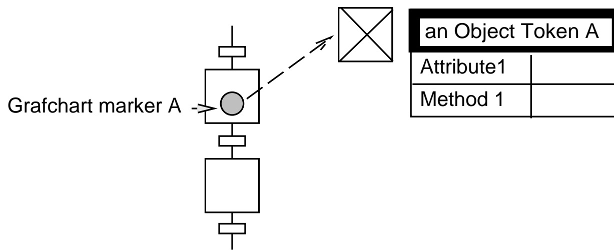


Figure 5.27 A grafchart-marker and an object-token.

An active step is indicated by the presence of a grafchart marker in the step. Figure 5.28 (left) shows a step that is active with respect to tokens of type A, Figure 5.28 (middle) shows a step that is active with respect to tokens of type A and tokens of type B. The step shown in Figure 5.28 (right) is inactive.

One of the ideas of introducing object tokens is that the description of a system can be made more compact. Imagine a system with two identical tanks which both should be filled with water, heated and then emptied,

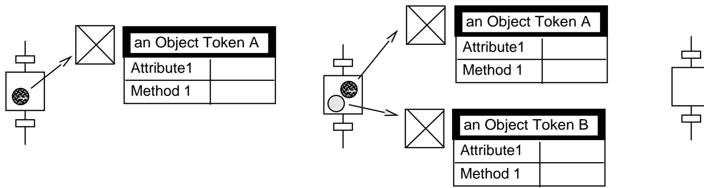


Figure 5.28 Two active steps (left and middle), and an inactive step (right).

compare Example 5.4.1. The tanks operate independently of each other. In the basic version of Grafchart, each tank system is controlled by a separate Grafchart and information about e.g., the temperature, the level etc., is stored as attributes of the entire chart, see Figure 5.29.

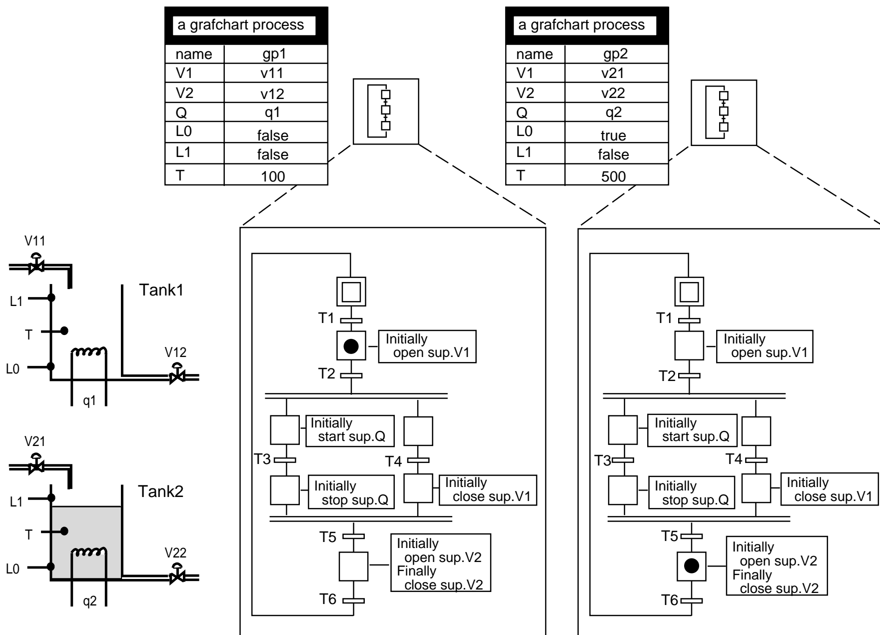


Figure 5.29 A system composed of two identical tanks, each controlled by a Grafchart.

If, however, the high-level version of Grafchart is used, the two tanks can be controlled by one chart in which there are two object tokens, one for each tank. The information about, e.g., the temperature, the level etc., can no longer be stored in the attributes of the chart since they do not have the same values for the two tank systems. Instead the information

is stored as attributes of the object tokens, see Figure 5.30.

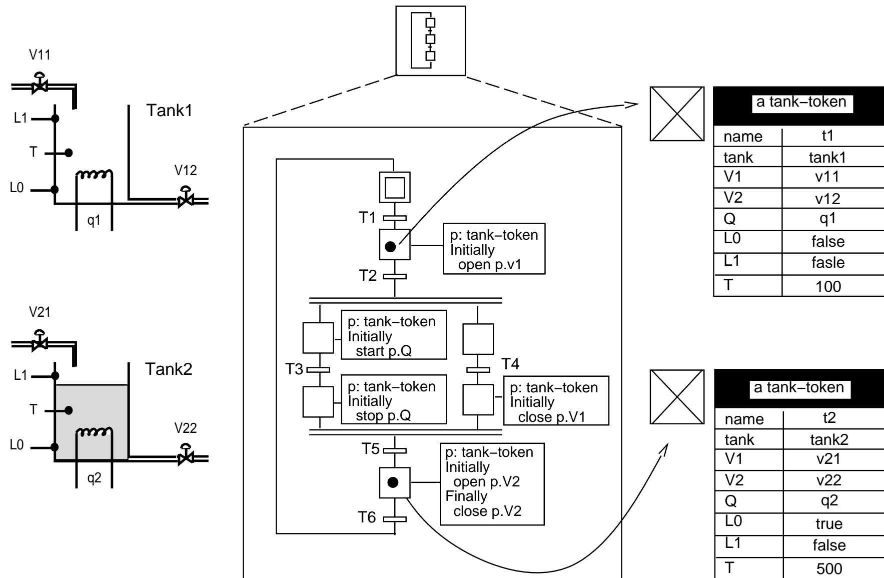


Figure 5.30 A system composed of two identical tanks, controlled by one Grafchart.

The corresponding grafchart markers can move around in the function chart independently of each other since the behavior of one tank does not depend on that of the other tank. However, applications also exist where the object tokens are not independent of each other, e.g., a production line where the different parts move between different stations but where it is impossible for one part to pass another. In these applications the grafchart markers cannot pass each other and a grafchart marker may therefore have to wait for another grafchart marker to leave a step. Another example with dependencies between the object tokens is batch production. The batches in a batch production cell are effected by different recipe operations and they share the same equipment. One batch might, e.g., have to wait for an other batch to release a resource. The high-level version of Grafchart can be used to model structures with dependencies between the object tokens.

A grafchart marker, or a grafchart marker together with its object token, will most often be refered to by the shorter term `token`. Only when we want to stress the fact that the grafchart marker and the object token are two different objects their correct names will be used.

Steps and Actions

A step may contain several tokens of the same or of different classes. To each step, actions can be associated. The action types are the same as in the basic version of Grafchart, i.e., initially, finally, always and abortive. The difference is that in Grafchart-HLV each action is associated with a token class. An initially (finally) action is executed when an instance of its token class enters (leaves) the step. An always action is executed periodically when an instance of its token class is present in the step. The actions may contain conditions that depend on the presence of tokens of other types or on the values of their attributes.

In Figure 5.31 a step and its actions are shown. Two tokens are placed in the step, one of class `P-token` and one of class `Q-token`. One `initially` and one `always` action are associated with the token class `P-token` and one `finally` action is associated with the token class `Q-token`.

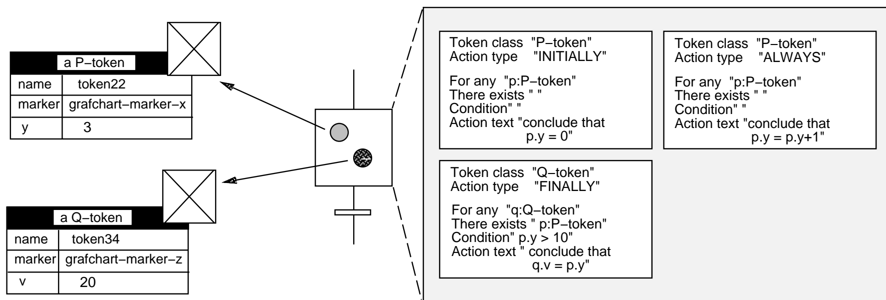


Figure 5.31 Step actions.

The actions-templates are built up by six different parts. In the `token class` part, the class of the token to which the action applies is specified. The `action type` indicates if the action is an initially, finally, always or abortive action. In the `for any` part, the token to which the rule applies is given a `temporary-token-name`, this name can be used in the action-template to refer to the token. The `there exists` part can be used to check the presence of other tokens and the `condition` part can used to check if certain conditions apply. The last part of the action-template is the `action text` where the action that should be performed is specified.

When a token of type `P-token` enters the step in Figure 5.31 its y attribute is assigned the value zero, as specified by the initially action-template. If the token does not directly leave the step, the always action-template will assure that the y attribute is incremented periodically. The finally action-template associated with the token class `Q-token` specifies what should happen with a token of type `Q-token` when it leaves the step. If a token of

type P-token exists in the step, and, if the value of the y attribute of the P-token is greater than ten, then, when a token of type Q-token leaves the step, the value of its v attribute will be assigned the same value as the value of the y attribute of the P-token.

Transitions and Receptivities

Each transition has a receptivity for each token class that the transition can be enabled by. The condition and/or the event of the receptivity may refer to the attributes of the token that enables the transition. It may also refer to the presence of other tokens in the input step and the value of their attributes.

A transition can also have a receptivity that applies to a token superclass. This means that the transition is enabled by any token belonging to a subclass of this superclass.

In Figure 5.32 a transition and its receptivities are shown. The transition has two receptivities, one associated with the token class P-token and one associated with the token class Q-token.

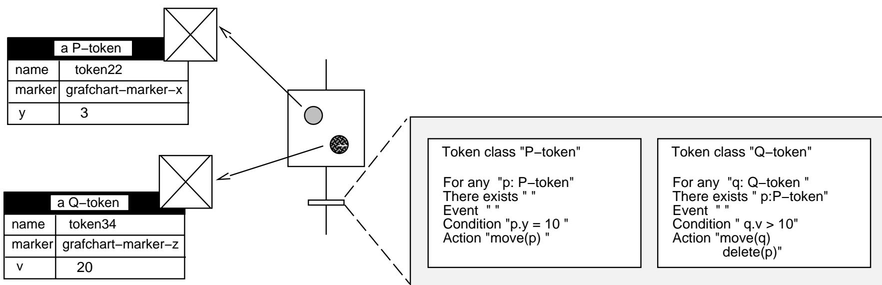


Figure 5.32 Transition with multiple receptivities.

The receptivity is built up by six parts. The token class, for any and there exists parts are used in the same way as those of a step action. The event and condition parts are used to specify the event and conditions of the receptivity. When a transition becomes fireable an operation is performed on the tokens that are referenced by the receptivity. The operation is specified in the action part of the receptivity.

In the standard case the operation would be to move the token from the input step to the output step. However, it is also possible to delete and create tokens and to change the value of the attributes of the tokens. The latter is useful primarily for initializing the values of a newly created token. The operations can also be more complex, e.g., an attribute of the token can be changed at the same time as the token is moved from the

input step to the output step, another token placed in the input step can be deleted or a new token can be created and placed in the output step. It is also possible to move, not only the token that enables the transition but also one or several of the tokens refered to in the condition or event part of the receptivity.

The action `move(p)` implies that the grafchart marker that is placed in the input step of the transition and that points on the object token named `p` is moved from the input step of the transition to the output step. The action `create(P-token)` implies that an instance of a P-token is created. A grafchart marker, of the class that animates a P-token is also created and placed at the output step of the transition. The pointer from the grafchart marker to the P-token is created. The action `delete(p)` deletes the grafchart marker that is placed in the input step of the transition and that points to the object token named `p`, and deletes the object token named `p`.

In Figure 5.32, two tokens are placed in the step preceding the transition, one token of class `P-token` and one token of class `Q-token`. The transition is enabled with respect to both token classes but it is only fireable with respect to the token of class `Q-token`. When the transition fires the token of class `Q-token` will be moved from the input step to the output step of the transition and the token of class `P-token` will be deleted.

Parallelism

The reason for having a grafchart marker that acts as a pointer to the corresponding object token is the way parallel structures are handled.

When the transition preceding a parallel-split (and-divergence) is fired, the grafchart marker is moved from the input step of the transition to the first step in one parallel branch, and copies of the grafchart-marker is added in the first step in each of the other parallel branches. The grafchart markers in all the parallel branches will therefore point to the same object token according to Figure 5.33. If the value of an attribute is changed in one of the branches it will directly be visible in all branches.

The transition after a parallel-join (and-convergence) is only enabled with respect to a token class if all the input steps of the transition contain grafchart markers that point to the same object token. When the transition is fired, one grafchart marker from one of the preceding steps in the parallel branch is moved to the output step, the rest of the grafchart markers are deleted. Since all grafchart markers, in the parallel branches, point to the same object token it does not matter which one that is moved and which are deleted. The parallel-join (and-convergence) situation is shown in Figure 5.34.

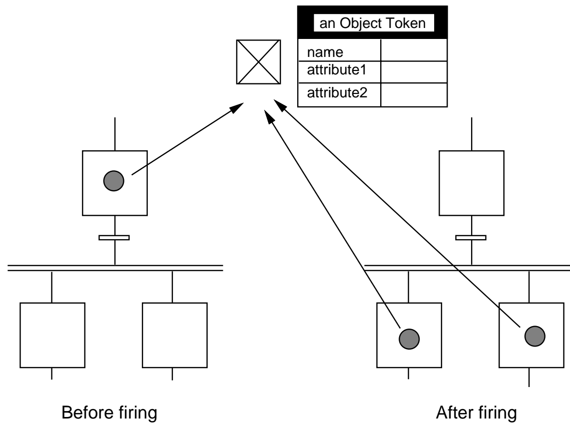


Figure 5.33 An and-divergence situation.

The reason for letting the grafchart marker point to an object token, containing the attributes, and not letting the grafchart marker itself contain the attributes, is the parallel-join structure. If the grafchart marker would contain the attributes and if the value of an attribute of one of the grafchart markers, in one of the parallel-branches, is modified, it would be unclear how the parallel-join (and-convergence) should be treated. Which value of the attribute should be kept and which should be ignored? By

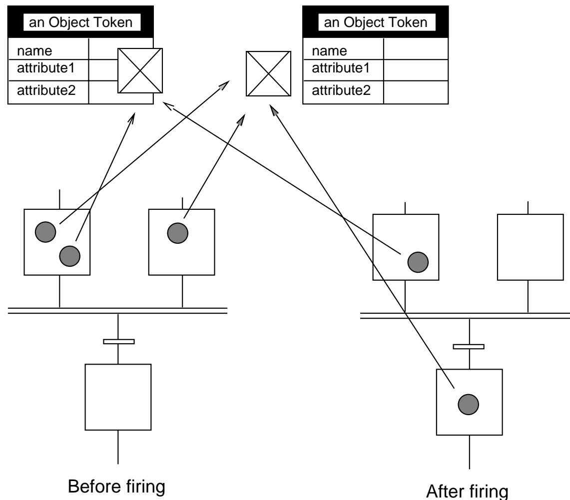


Figure 5.34 An and-convergence situation.

letting the grafchart markers be pointers to an object token containing the attributes, problems like this are avoided. This is also a natural way to model a system since a grafchart marker in a chart is a marker visualizing the state of only one object. For example, the grafchart markers in Figure 5.30 each represent the state of one tank. Even if the state is visualized by a parallel-branch (the tank is heated and filled at the same time) the markers still represent one tank.

PN-transition

A special kind of transition has been introduced in Grafchart-HLV, the transition is called PN-transition (Petri net transition). A PN-transition can, unlike a normal transition, have more than one incoming arc and more than one outgoing arc. The arcs are numbered and the numbers can be used to specify from which step and to which step the token should be moved.

A PN-transition is shown in Figure 5.35. The PN-transition is enabled with respect to the color `P` since there is a token of class `P` in the preceding step connected to arc number 1. The PN-transition is also fireable since there exists a `Q` token in the preceding step connected to arc number 2. When the transition is fired, one token of color `P` is moved from the left input step to the output step connected to arc number 1, and one `Q` token is moved from the right input step to the output place connected to arc number 2, as specified in the `action` part of the receptivity.

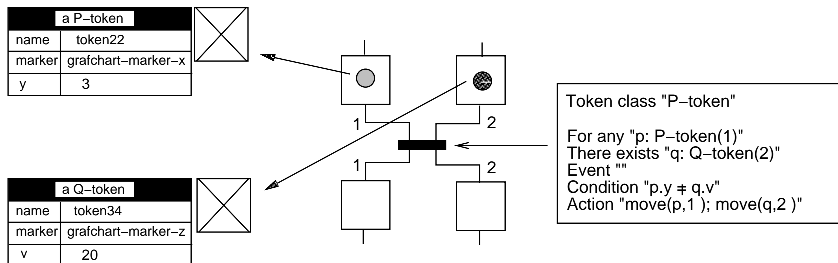


Figure 5.35 An and-divergence situation using a PN-transition.

Macro steps

As soon as a grafchart marker enters a macro step a copy of the grafchart marker is placed in the enter-step of the macro step, i.e., the grafchart marker placed in the macro step and the grafchart marker placed in the enter-step points at the same object token. Several tokens can be in a macro step at the same time.

Procedure- and Process steps

In the basic version of Grafchart a procedure or process step has an associated attribute that specifies the name of the Grafchart procedure that should be called. In Grafchart-HLV the call to a Grafchart procedure is more flexible.

Associated with each procedure or process step is a procedure-call-template. A procedure-call-template is built up by three parts. The `token class` part specifies the class of the token to which the template apply. The `parameter` part specifies the attributes to the procedure, if any, and the `procedure` part determines the name of the Grafchart procedure to be called. The name of the Grafchart procedure can either be given explicitly or implicitly through a reference.

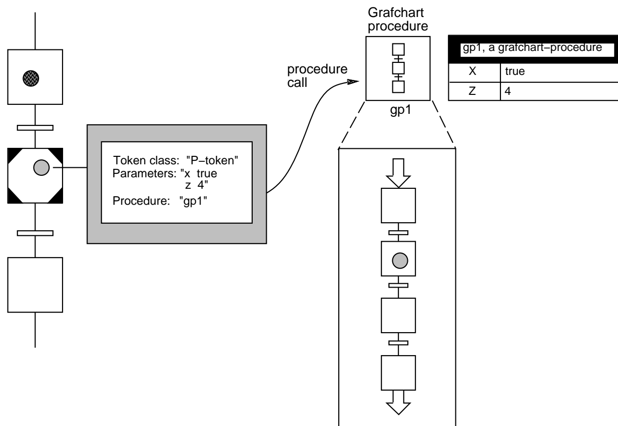


Figure 5.36 Two tokens placed in a procedure step.

In Figure 5.36 a procedure step is shown. A token of class `P-token` is placed within the procedure step. The token causes a call to a Grafchart procedure named `gp1`. The Grafchart procedure `gp1` has two attributes, `x` and `z`. When the call is performed the `x` attribute is given the value true and the `z` attribute is given the value 4. It is also possible to give the `x` attribute and the `z` attribute the values of an attribute of the token. To do this, a special `inv` notation is used, see Chapter 5.5 (Object Tokens).

Each call is executed in its own copy, i.e., when a call is done to a Grafchart procedure a copy of the Grafchart procedure body is created and this copy is executed, when the execution reaches its end the copy is deleted. This means that there can never be more than one token in each Grafchart procedure and therefore the Grafchart procedures of Grafchart-BV can be

used also in Grafchart-HLV. The attributes of the token are, if necessary, transformed into attributes of the Grafchart procedure called by the token.

Parameterization

The parameterization feature is included in the high-level version of Grafchart. The `sup` notation can be used to reference attributes of an object visible in the current context in the same way as presented earlier, see Chapter 5.3.

Methods and Message Passing

The method and message passing feature is included in the high-level version of Grafchart. The `self` notation can be used to reference attributes belonging to the object of the method in the same way as presented earlier, see Chapter 5.3

Object Tokens

In the high-level version of Grafchart, the tokens are objects with an identity. The tokens can, unlike the tokens in the basic version of Grafchart, carry information. This feature is sometimes refered to as the object token feature of Grafchart.

The attributes of an object token placed in a step or macro-step are referenced from step actions and transition receptivities using the `temporary-token-name` notation, see Chapter 5.5 (Steps and Actions) and Chapter 5.5 (Transitions and Receptivities).

In procedure or process steps, the attributes of an object token are referenced using a special `inv` notation.

- `inv.attribute1`
  refers to the `attribute1` attribute of the token

In Figure 5.37 a procedure step is shown. Two tokens are placed within the procedure step, one token of class `P-token` and one token of class `Q-token`. Tokens of class `P-token` cause calls to a Grafchart procedure named `gp1` whereas tokens of class `Q-token` cause calls to a Grafchart procedure named by the `proc` attribute of the token itself. The grafchart-procedure `gp1` has two attributes, `x` and `z`, compare figure 5.36. When the call to the Grafchart procedure is performed, the `x` attribute is given the value of the `x` attribute of the `P-token` giving rise to the call, and the `z` attribute is given the value 4.

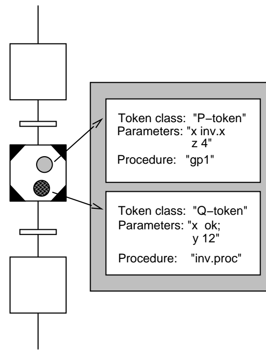


Figure 5.37 Two tokens placed in a procedure step.

Multi-dimensional Charts

Since a token is an object and since objects are allowed to have methods also tokens may have methods. This will give a multi-dimensional structure where a token moving around in a chart may itself contain one or more charts. This feature is sometimes refered to as the multi-dimensional chart feature of Grafchart. The feature gives several interesting structuring possibilities.

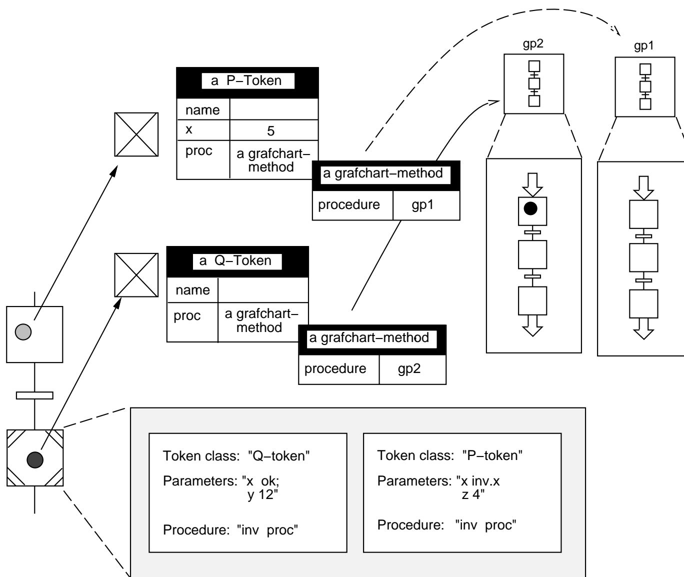


Figure 5.38 A multi-dimensional chart.

A part of a multi-dimensional function chart is shown in Figure 5.38. It consists of one step and one process step. A token of class `P-token` is placed in the step and a token of type `Q-token` is placed in the process step. The name of the Grafchart procedure to be called from the process step is given by the reference `inv proc`, i.e., the Grafchart procedure to be called is the method named `proc` of the token.

The `Q-token`, placed in the process step, in Figure 5.38 has caused a call to the Grafchart procedure gp2. The `Q-token` can however, continue its execution independently of the execution of gp2. When the `P-token` enters the process step a call to the Grafchart process gp1 will be effected.

The reference to the attributes of an object token is done in different ways depending from where the reference is done, see Figure 5.39. If the attribute of an object token should be referenced from within the chart where the corresponding grafchart marker is placed, the `temporary-token-name.attribute` notation is used, and, if the attribute should be referenced from within a method that belongs to the object token, the `self.attribute` notation is used.

In Figure 5.39 a `P-token` is placed in a step. The `P-token` has an `x` attribute and a method called `proc`. The method is implemented by the graf-

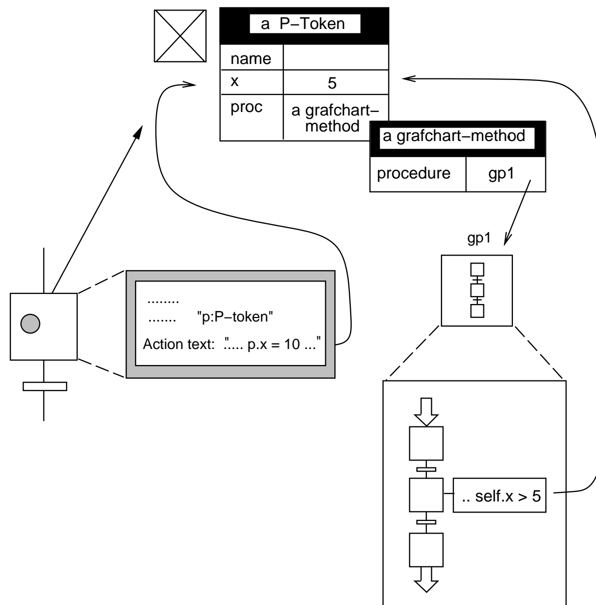


Figure 5.39 Different references to an attribute of an object token.

chart method gp1. Associated with the step where the token is placed, is an action. In the action the P-token is given the temporary name p and the x attribute is refered to as p.x. The method has three steps. Associated with the middle step is an action. When refering to the token attribute x from within this action the self.x notation is used.

The different levels in a multi-dimensional function chart can communicate with each other through the different dot notations that exist, see Figure 5.40.

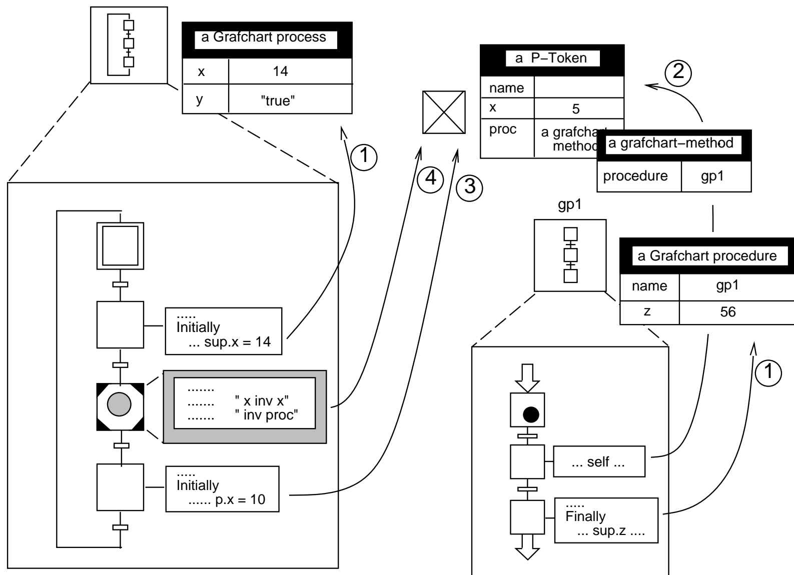


Figure 5.40 The communication between the different levels in a multidimensional function chart.

Four different dot notations exist:

1. The sup notation, described in Chapter 5.3 (Parameterization)
2. The self notation, described in Chapter 5.3 (Methods and Message Passing)
3. The temporary-token-name notation, described in Chapter 5.5 (Steps and Actions)
4. The inv notation, described in Chapter 5.5 (Object Tokens).

Formal Definition

The formal definition of the high-level version of Grafchart is presented in Appendix C.

Conceptual Ideas

The ideas introduced in Grafchart-HLV can be used in many types of applications. The general idea is that there is a system where one or several "parts" reside. The "parts" may be e.g., cars, people, or chemical batches. The "parts" are allowed to have individual information, e.g., production-descriptions, personal thoughts or customer demands.

The concept of Grafchart is the following: information that applies to all parts in the system is stored either in the base-level function chart or as global attributes whereas "part" specific information is stored either as methods belonging to the token representing the part or as local attributes, i.e., attributes of the token representing the part.

Consider the example of the Russian dining philosophers, [Lakos, 1994]. Four philosophers are sitting around a circular table. In front of each philosopher there is a plate. Between each plate there is a chopstick, i.e., on the table there are four plates and four chopsticks. Each philosopher is thinking of a problem. Depending on the state of the problem the philosopher would either like to eat or think. If the philosopher would like to eat, he or she can only do so if the two chopsticks on each side of his plate are not used by any of his adjacent philosopher. In this application each philosopher corresponds to a token. The token resides in a chart corresponding to the table and its setting. The problem that the philosopher is thinking on is represented by another chart stored as a method of the token.

Another type of application suitable for Grafchart is production lines. In a production line entities are produced. The entities-under-production are moving along a conveyor belt. During their transport they pass one or several machines. In this application each entity could be represented by a token. If all entities should be produced in exactly the same way, the information about how this should be done can be represented by the base-level function chart. If the entities differs only very little, the entity-specific information can be stored as attributes of the corresponding token. From the base-level function chart it is possible to refer to the entity-specific information when needed. Production-lines do also exist that are customer driven, i.e., the machines that produce the entities perform different tasks depending on the entity. In this case the base-level chart can be made very general and information about what a machine should do with a certain entity can be stored as methods of the token. When a token enters a step that corresponds to a certain machine the correct method is called. In this application several entities may be under production at the same time, however, the entities that are under production can not pass each other on the conveyor belt. This means that the relative order among the tokens cannot change.

5.6 Error Handling

Grafchart is, as mentioned earlier, based on Grafcet. A common problem with Grafcet is how the logic for the normal operating sequence best should be separated from the error detection and error recovery logic. A separation is necessary, otherwise, the function chart will rapidly become very large, its clear structure will disappear, and the function chart will be hard to read and understand.

Grafchart contains a number of assisting features for separation of error handling logic and normal operating logic.

Exception transitions The exception transition, see Chapter 5.4, is a special type of transition. The exception transition can be connected only to macro steps and procedure steps. The exception transition is, unlike normal transitions, constantly enabled while the macro step or process step is active. If the macro step or procedure step is used in such a way that it corresponds to a certain state in the application, it is, by using exception transitions, possible to collect all error conditions, applying to this state, in a compact way. This clarifies the implementations and it has therefore proved to be very useful.

The exception transition has, so far, only been implemented in the basic version of Grafchart.

Connection posts A connection post is a special G2 item used to break a graphical connection, e.g., a connection between a step and a transition. This enhances the readability of the chart and can be used to separate error handling from normal operating logic. The use of connection posts is shown in Figure 5.41. The normal operating sequence has been separated from the error handling logic. Connection posts with the same name establish a logical connection.

Macro actions Grafchart contains a number of macro actions. The macro actions affect the operation of an entire function chart. The macro actions can be called from step actions. There exist macro actions that cause a step to be deactivated, force a step to be active, force a transition to fire, abort an entire chart, freeze a transition or a whole chart, release a transition or a whole chart, and resume a macro step. Similar macro actions exist in Grafcet, [David and Alla, 1992]. The macro actions can be used to split the error handling logic and the normal operation logic into separate Grafcharts without loosing the interactions between the two parts.

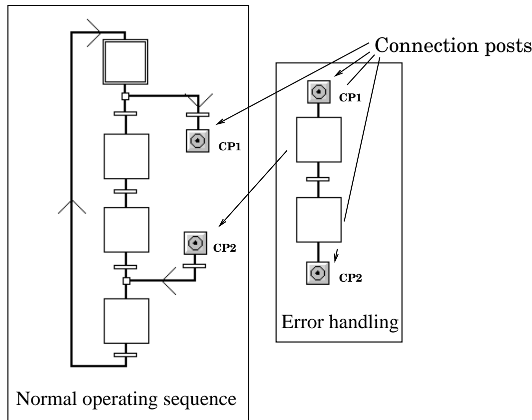


Figure 5.41 Two nets connected by connection posts.

Step fusion sets Grafchart-BV also supports step fusion sets, [Jensen and Rozenberg, 1991]. Each step in a fusion set represents one view of the same conceptual step. Using fusion sets a step can have multiple graphical representations. When one of the steps in the fusion set becomes active (inactive) all the steps in the set will become activated (inactivated). Fusion sets can conveniently be used to separate normal operation logic from error handling logic.

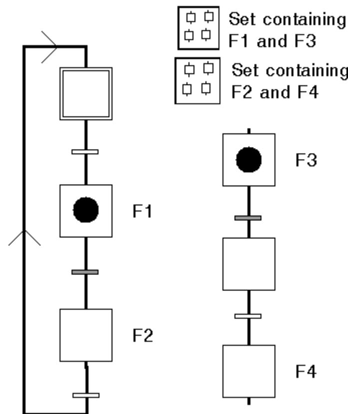


Figure 5.42 A step fusion set.

A small example of a step fusion set is shown in Figure 5.42. In this example step F1 and step F3 belong to one fusion set and step F2 and step F4 belong to another fusion set. In Figure 5.42 step F1 is active, and since step F3 belongs to the same fusion set also step F3 is active. Both of the transitions following step F1 and step F3 are enabled. The normal execution order is represented by the closed function chart: initial step, F1, F3. If something goes wrong while being in step F1, the transition following step F3 will fire, the error will be repaired, represented by the intermediate step, and step F4 becomes active. At the same time as step F4 is activated also step F2 is activated and the normal execution can continue.

EXAMPLE 5.6.1
Consider the tank example presented in Chapter 5.4, Example 5.4.1. The normal operation logic of this system is presented in Figure 5.26. Assume that the normal operation can be disturbed by two faults, called `X` and `Y`, [David and Alla, 1992]. When one of the faults occur, all actions should be reset, i.e., all valves should be closed and all pumps and heaters switched off. In addition to this an acoustic alarm should be turned on. When the fault has been repaired the normal operation starts again and the acoustic alarm is turned off. If the fault was of type `X` the restart should be done in the initial state, if the fault was of type `Y` the restart should be done in the state normally reached when indicator `L0` becomes true. The fault is either restored by itself or repaired by a service man. The service man can turn off the acoustic alarm before it has been repaired. This is done by pressing a button called `service arrived`. If the fault `X` occurs twice in less than a minute, the fact that the fault is restored by itself will not be enough for the system to return to its normal operation, a manual inspection is also required. To indicate that the inspection has taken place, the service man should press the button `inspection done`.

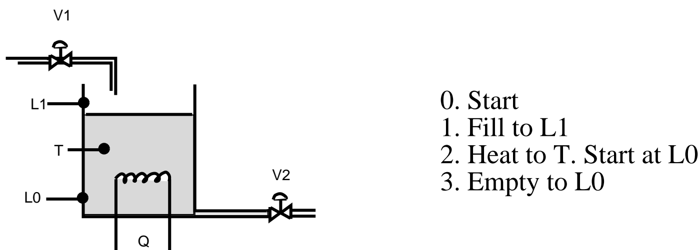


Figure 5.43 A tank example.

The Grafchart for the system is presented in Figure 5.44. In this Grafchart, the normal execution logic and the error logic are combined into

one Grafchart. As can be seen the Grafchart becomes very large and complex. The transition receptivities for transition `T1 - T6` are the same as the ones presented in Figure 5.26, transition receptivities `T8 - T15` are presented under the function chart.

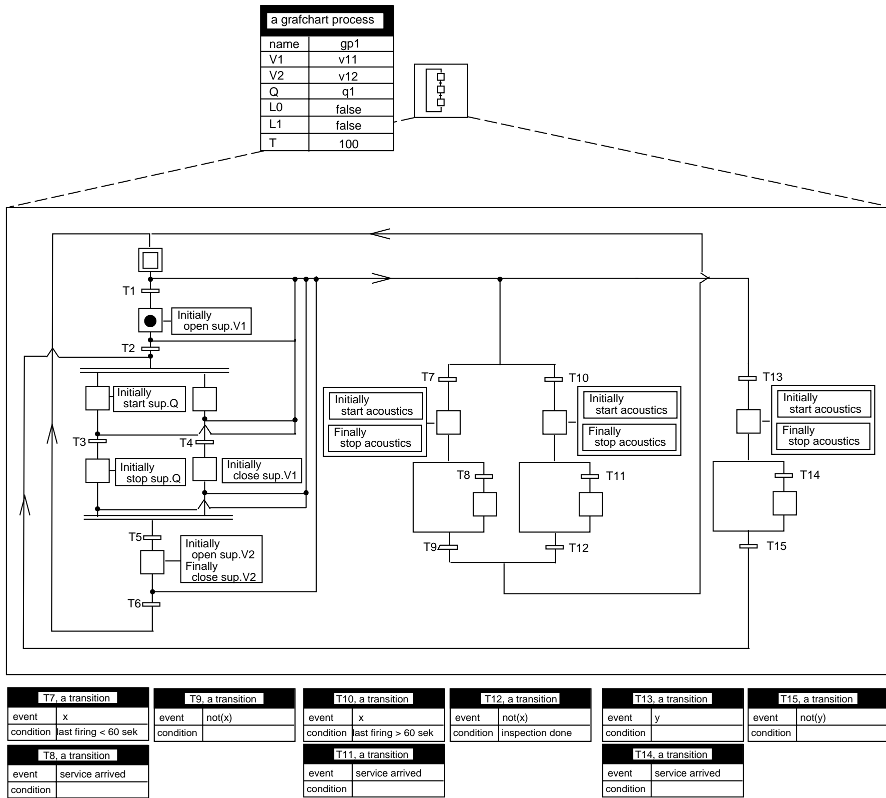


Figure 5.44 The tank example extended with error logic.

To make the implementation easier to grasp, exception transitions and connection posts can be used. This solution is indicated in Figure 5.45. The normal execution is encapsulated within a macro step to which two exception transitions, one for each type of fault, are connected. The error logic is stored on two separate workspaces which are connected to the main workspace by connection posts. The transition receptivities that differs from the once shown in Figure 5.26 and Figure 5.44 are shown in the figure.

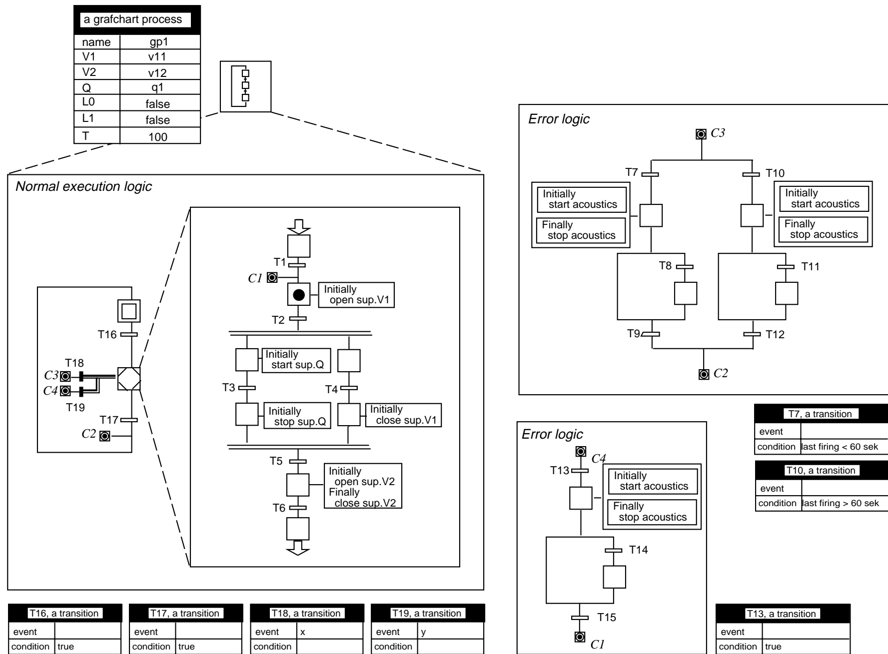


Figure 5.45 The tank example extended with error logic.

Easy-to-grasp-solutions do also exist where macro actions or step fusion sets are used.

#

5.7 Foundations of Grafchart

Grafchart is, as we have seen, a graphical programming language for sequential control. The language is suitable both for applications on the local level and on the supervisory level. The language is easy to use and understand and it has potential for formal descriptions, validation and analysis as shown in Chapter 9. Furthermore, the language is well structured and easy to use. In order to achieve this, Grafchart has been influenced by work in several different areas. The main foundations of Grafchart are: Grafcet, Petri nets, high-level Petri nets and object-oriented programming, see Figure 5.46.

When reading through the foundations of Grafchart and their impact on the language it is important to keep in mind that the main objective when

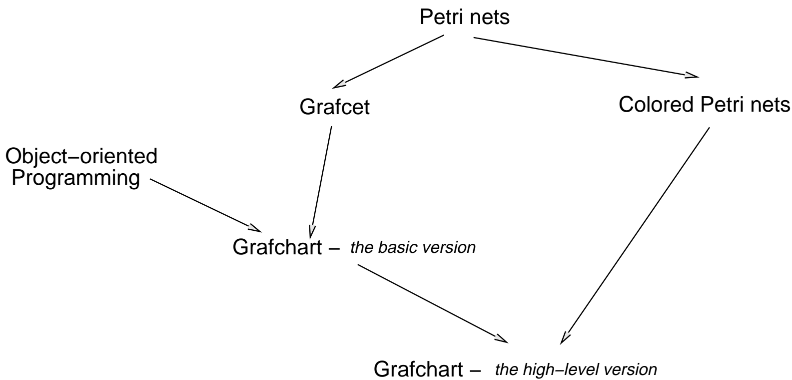


Figure 5.46 The foundation of Grafchart.

developing the language has been to provide the user with a user-friendly and graphical high-level language for sequential control. As a side effect, caused by the connections to Grafcet and Petri nets, the language could also, to some extent, be used for analysis and validation of systems.

Grafcet

Grafcet, [David and Alla, 1992], has a clear and intuitive syntax. Grafchart is based on the same syntax even though it has been enlarged by some extra graphical elements. The syntax of Grafcet has also been very well accepted in industry. Therefore we believe that it is a good idea to keep this syntax.

Petri Nets

It is very difficult, probably impossible, to make an informal explanation of a function chart which is complete and without ambiguities. It is thus extremely important that the intuition is completed by a formal definition, [Jensen, 1992]. Both for Petri nets and Grafcet a formal description exists, [David and Alla, 1992] and [Jensen, 1992]. A similar description language has been worked out for Grafchart.

In order to detect e.g., deadlock situations, formal analysis methods must exist. Since Grafchart is based upon Petri nets and since formal analysis methods exist for Petri nets, it is sufficient to show that a Grafchart function chart can be transformed into a Petri net and then applied to the already existing methods. Alternatively, the Grafchart function chart can be transformed into a Grafcet and the formal methods that have been developed for Grafcet can be used, [Roussel and Lesage, 1996] and [De Loor *et al.*, 1997].

High-Level Petri Nets

High-level Petri nets, [Jensen, 1992], allow a compact graphical description of systems with several similar parts. This can be very useful in some applications that otherwise would have to be modeled by a very large Petri net. A more compact description of a large system facilitates the readability of a net, however, the readability decreases again if the description is made to compact, compare Figure 4.1, Figure 4.2 and Figure 4.4.

Object-Oriented Programming

Object-oriented programming incorporates powerful abstraction facilities and supports well structured applications. The parameterization and method and message passing features in Grafchart are abstraction facilities inspired from similar features found in most object-oriented languages. When implementing large systems it is very important to have good and logical structuring possibilities. The object token concept is a step in this direction.

5.8 Grafchart vs Grafcet/Petri Nets

The language of Grafcet and Petri net and the language of Grafchart have many similarities. This is natural since Grafchart is based upon Grafcet and Petri nets. However, there are also some important differences, these will be pointed out in this section.

General Aim

Grafcet was developed with the focus on logical controllers (PLCs). It is most often used only as a specification language, i.e., as a graphical description of how the controller is intended to work. The implementation is then done either in Grafcet or in another language suitable for PLC implementations such as, e.g., relay (ladder) logic diagrams or instruction lists. Grafchart has been developed without focus on any particular application. Grafchart is a general graphical language aimed at sequential control. Thanks to the advanced features that are included in Grafchart, the language is suitable to use both at the local control level and at the supervisory control level. Grafchart is also intended to be a language used, not only for specifications, but also for the actual implementation. A Grafchart function chart is interpreted on-line.

Petri net was developed with the aim to be used for analysis, simulation and visualization of a process and not for control.

Graphical Interface

The graphical interfaces of Grafcet and Grafchart have many similarities, e.g., the steps , the transitions, the parallel structures and the alternative structures. These similarities are intentional. However, the Grafchart language contains more graphical elements (building blocks) than Grafcet. The new building blocks introduced in Grafchart are: Grafchart processes, Grafchart procedures, process steps, procedure steps and exception transitions. The new building blocks do not extend the functionality of Grafchart compared to Grafcet, but they facilitate the implementation of more advanced sequence structures. The development of Grafchart from Grafcet can be compared to the evolution from assembler languages to high-level programming languages like Java.

The steps in Grafchart-HLV and the places in Colored Petri nets, have many similarities. They both represent the states of the system and they may contain several tokens. The fact that the steps in Grafchart have actions is a major difference from Petri nets. The actions make Grafchart resemble Grafcet. However, places with actions can be found in a version of Petri nets called Interpreted Petri nets, [Moalla, 1985], but here the actions are only expressed with boolean variables.

The structure of the receptivities in Grafchart-HLV differs from the one used in Petri nets. The receptivity in Grafchart has not only a conditional part where it is possible to check if a variable or expression is true or false, but also an event part where it can be detected when a variable receives a new value. The event part in the receptivity is a trace from the Grafcet influence. A Grafchart-HLV structure with a transition containing two receptivities and the corresponding Colored Petri net structure is shown in Figure 5.47.

External Interface

Grafcet and Grafchart have different external interfaces for actions and receptivities. The external interface of Grafcet consists primarily of boolean variables. Two action types are supported: impulse actions and level actions. The level actions are only of interest for binary variables (high/low, true/false, etc).

The external interface of Grafchart consists of the data types that are supported by the underlying implementation language. The current toolbox of Grafchart is implemented in G2 and it supports a lot different variable types, e.g., booleans, numbers, symbols, strings, arrays, structures, etc. Since the boolean variables are not as important in Grafchart as in Grafcet, only impulse actions are supported.

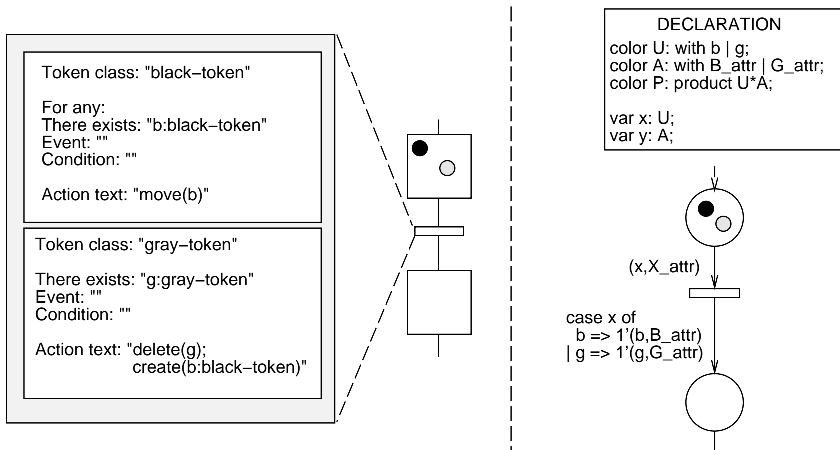


Figure 5.47 A Grafchart structure (left) and an identical Petri net structure (right)

Execution

The firing rules that apply to Grafchart are similar, but not identical, to those of Grafcet.

- Rule 1: All fireable transitions are immediately fired – this rule applies both to Grafcet and to Grafchart.
- Rule 2: Several simultaneously fireable transitions are simultaneously fired – for Grafcet this is true in all situations, for Grafchart this is true for all situations except in an or-divergence situation. Grafchart treats this situation by nondeterministically choosing one of the branches. In order to have an interpretation algorithm without ambiguities, situations like this must be avoided. It is therefore required that the user makes all branches in an or-divergence situation mutually exclusive.

  If the Grafchart is made deterministic and the Grafcet is sound, see Chapter 3.4, this difference does no longer exist.
- Rule 3: When a step must be simultaneously deactivated and activated, it remains active – this rule applies to Grafcet but not to Grafchart. In Grafchart, the step will first be deactivated and then activated again.

The interpretation algorithm of Grafchart describes the theoretical understanding of a Grafchart and guarantees that everyone faced with the same

Grafchart understands it in the same way. For Grafcet, many different interpretation algorithms exists. A complete and unambiguous algorithm is given by [David, 1995]. This algorithm is global, see Chapter 3.2 (Interpretation Algorithm). A global interpretation algorithm does also exist for Grafchart. However, the algorithm currently implemented is local. When a new external event occurs, the global algorithm searches through all receptivities to see which transitions that have become fireable. Since the implementation of Grafchart is based on activatable subworkspaces such a search is not performed. When a new external event occurs or a condition becomes true and the corresponding receptivities becomes true, a procedure called `fire` is automatically started. This procedure takes care of the deactivation and activation of steps and the enabling and de-enabling of transitions. The local algorithm is more efficient and less time demanding than the global algorithm. The local algorithm is also more suitable than the global algorithm for applications where the execution is distributed.

EXAMPLE 5.8.1
Imagine a scenario where the execution of a system is implemented by a Grafchart. The system could, e.g., be a production line where soda bottles are being produced. Three machines are involved in the soda bottling process. The first machine, called machine A, pours the soda into a bottle. The second and third machine, called machine B and machine C respectively, work in parallel. Machine B washes the bottles and puts a label on them, at the same time as machine C wipes off the top of the bottle, seals it with a cap and specifies its best-before-date. The Grafchart for controlling the process is shown in Figure 5.48 (upper part).

The Grafchart is configured in the supervisory node but its execution is distributed on three nodes, one for each machine. Each node contains the relevant parts of the Grafchart. The production of a new soda bottle is started from Node A, a token is then placed within the initial step in the Grafchart stored in Node A. The Grafchart is executing within Node A until the execution reaches the transition preceding the parallel split (and-divergence). The token, with possible attributes and methods, is then sent to Node B and Node C. These nodes know in which step the token should be placed. Since the interpretation algorithm of Grafchart is local, Node B and Node C can execute independently of each other. When a new external event occurs or when a condition becomes true the transitions concerned are fired. If the interpretation algorithm would have been global, all transitions in the Grafchart, independently if they were stored in Node A, Node B or Node C, would have had to been scanned in order to see which transitions that should fire. The fact that all transitions should be scanned through each time a new event occurs or a condition

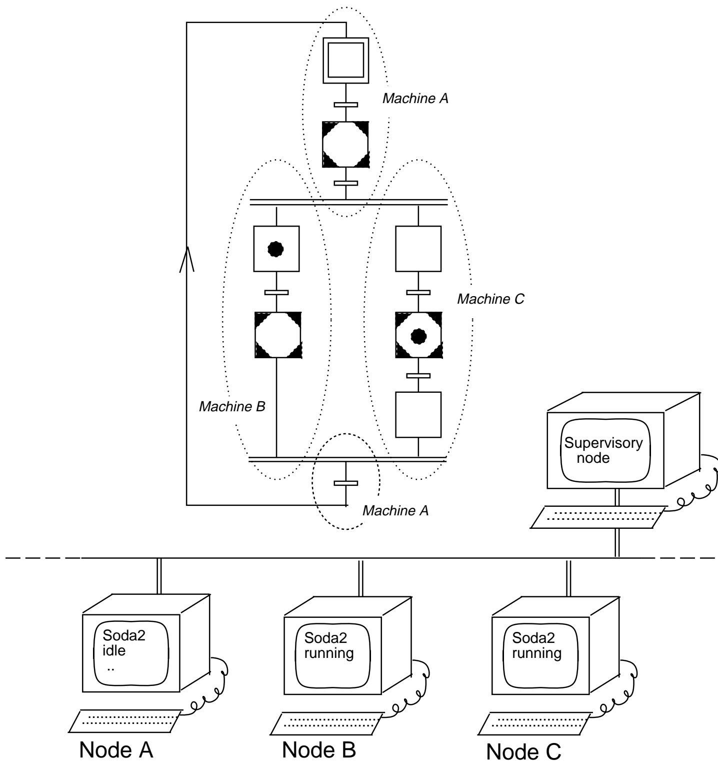


Figure 5.48 A distributed system.

becomes true would require that one of the nodes acted as a supervisory node taking care of the scanning task. In the standard for Sequential Function Charts, IEC 1131-3, [IEC, 1993], distributed systems are not dealt with. In the standard it is assumed that an SFC is always executed within the same node. However, ongoing standardization efforts are also concerning distribution.

#

The firing of a transition is viewed differently in the Grafchart-HLV and the Petri net model. When a transition fires in a Grafchart function chart, the tokens are moved from the input places to the output places. This means that it is the same tokens carrying the same attributes that are moving around in the net. In the Petri net formalism, the tokens are not being moved from input places to output places. Instead the tokens are removed from the input places and completely new tokens are added to

the output places. It is, however, possible to obtain the same firing view, in Grafchart as in Petri nets, by using the `delete` and `create` functions in the receptivities but it is not the intention that these functions should be used for this purpose.

Arc Inscriptions

In the Petri net formalism, arc inscriptions are needed to specify how the different input places should contribute to the enabling of a transition and how the different output places should be affected when the transition is fired. Assume a Petri net with a place followed by an and-divergence situation with two paths. The transition following the place is enabled if there is a token of class `P-token` and a token of class `Q-token` in the place. When the transition fires the two tokens should be removed from the input place and a `P-token` should be added in the left path and a `Q-token` should be added in the right path. In High-Level Grafchart, situations like this can be handled in two different ways. The first solution involves a structure transformation. The and-divergence case is transformed into an or-divergence case and an extra empty dummy step is introduced, see Figure 5.49.

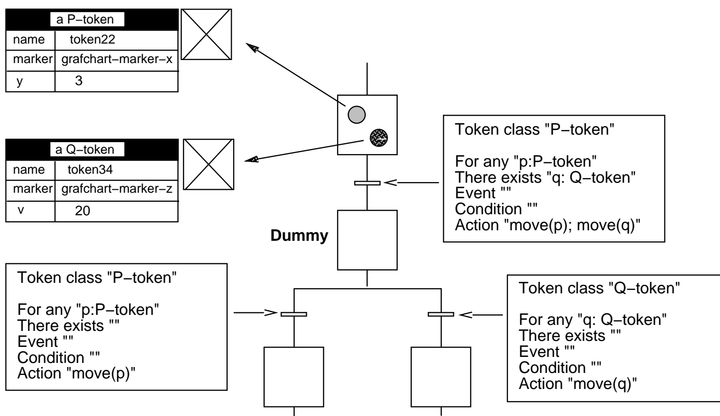


Figure 5.49 An and-divergence situation transformed into an or-divergence situation.

In most applications, however, this structure transformation is undesirable. Therefore a special kind of transition, named PN-transition, was introduced, see Chapter 5.5 (PN-transition). A PN-transition can have more than one incoming arc and more than one outgoing arc. The arcs are numbered and the numbers can be used to specify from which step and to which step the token should be moved. The arc numbers of PN-transitions is the only case where arc inscriptions are used in Grafchart-HLV.

Grafchart - *the basic version*: Transformation to Grafcet

Since the new building blocks introduced in Grafchart do not extend the functionality, a Grafchart can be transformed into an equivalent Grafcet. A Grafchart and a Grafcet are identical if, applied to the same input sequence, their output sequences are identical.

The transformation of a Grafchart to a Grafcet consists of three steps. First, the graphical elements must be changed to those of Grafcet. Secondly, the action types of Grafchart must be changed to those of Grafcet. Special care has to be taken to the firing rules that are interpreted differently in the two models.

Graphical elements: If the Grafchart contains only transitions and steps without any actions, the structure of the corresponding Grafcet will be identical to that of the Grafchart. If the Grafchart contains macro steps, the Grafchart macro step icon must be changed to that of Grafcet. The enter step and the exit step of the Grafchart macro step must be replaced by an input step and output step of Grafcet. Alternatively, the macro step icon can be replaced by its internal structure in which the enter step and exit steps are replaced by ordinary steps.

If the Grafchart contains procedure steps these should be replaced by a Grafcet macro step. The internal structure of the macro step should be that of the Grafchart procedure called by the procedure step, see Figure 5.50.

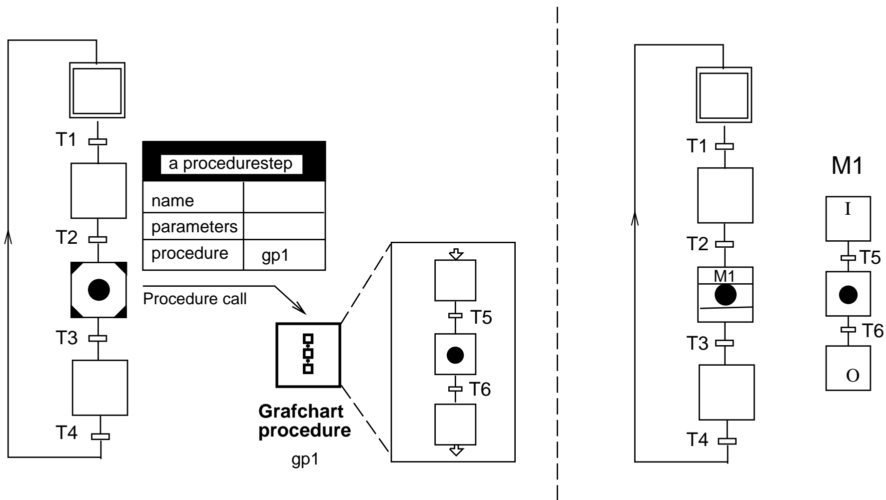


Figure 5.50 A Grafchart (left) and an identical Grafcet (right).

If the Grafchart contains process steps, these should be replaced by a parallel construct (and-divergence) with two branches. One branch contains an empty step followed by the transition succeeding the process step. The second branch contains an empty step followed by an alternative structure (or-divergence) where each path contains a macro step followed by a sink transition. The macro steps are all identical and their internal structure is that of the Grafchart procedure called by the process step. The transition preceding a macro step contains a receptivity that assures that the token does not enter a macro step that is already active. The number of paths in the alternative structure are equal to the number of processes that can run at the same time. This number has to be finite. The transformation of a process step is shown in Figure 5.51.

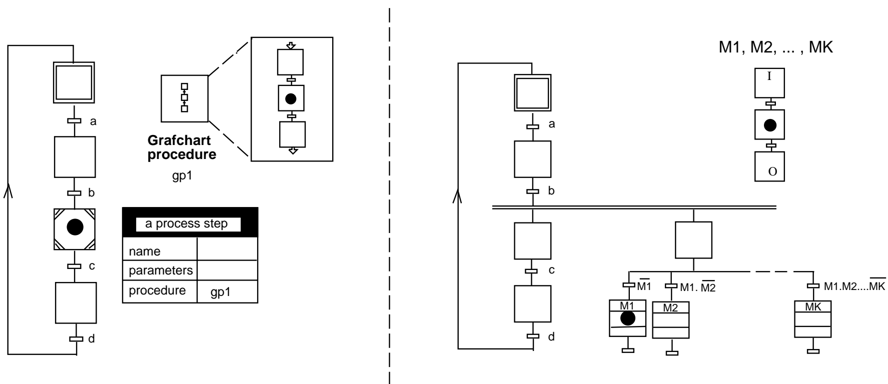


Figure 5.51 A Grafchart(left) and an identical Grafcet (right).

If the macro step or procedure step has an exception transition connected to it, the macro step icon or procedure step icon has to be replaced by its internal structure. To each of the internal steps an extra transition has to be added containing the events and conditions of the exception transition. The transformation of an exception transition is shown in Figure 5.52.

As can be seen in the figure each step contained in the internal structure of the macro step or procedure step is, in the Grafcet representation, followed by an or-divergence situation. In order to have a deterministic interpretation algorithm it is required that all branches in an or-divergence situations are made mutually exclusive, see Chapter 5.8 (Execution). When a procedure step or macro step have an exception transition connected to it, it is most often the users intention that this transition should have a higher priority than the transitions within the internal structure. In Figure 5.52 this means that the receptivities `b, d, e, c` should be extended with the receptivity `not(a)`. Another possibility is to solve the problem

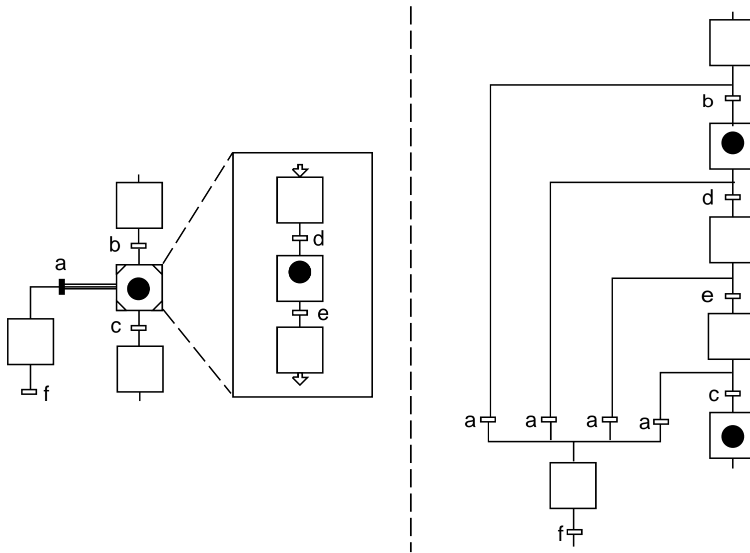


Figure 5.52 A Grafchart(left) and an identical Grafcet (right).

by letting an exception transition have a higher priority than an ordinary transition. This solution is however not ideal since it implies that the corresponding Petri net, used for analysis purposes, is a Petri net with priority. A Petri net with priority is, concerning complexity, comparable with a Turing machine and it can therefore not be analyzed.

Action types: Each step in Grafchart, that contains actions can be replaced by one or several steps with actions in Grafcet. Since Grafcet does not include the concept of impulse actions executed at other time instances than at the activation of the step, the Grafcet structure, corresponding to a Grafchart step with several actions, must include more than one step. The Grafcet structure must assure that the Initially action occurs first, this is guaranteed by starting the Grafcet structure with an impulse action in a separate step. After this step follows a loop taking care of the periodically execution of the Always action. The execution period of this action should be the same as the scan interval of the Always action implemented in Grafchart. Finally a separate step ends the Grafcet structure. This step contains an impulse action corresponding to the Finally action in Grafchart. The transformation is shown in Figure 5.53.

The transformation of a macro step with actions, or procedure step or process step for which the corresponding Grafchart procedure has actions, is possible but results in a large and complex Grafcet.

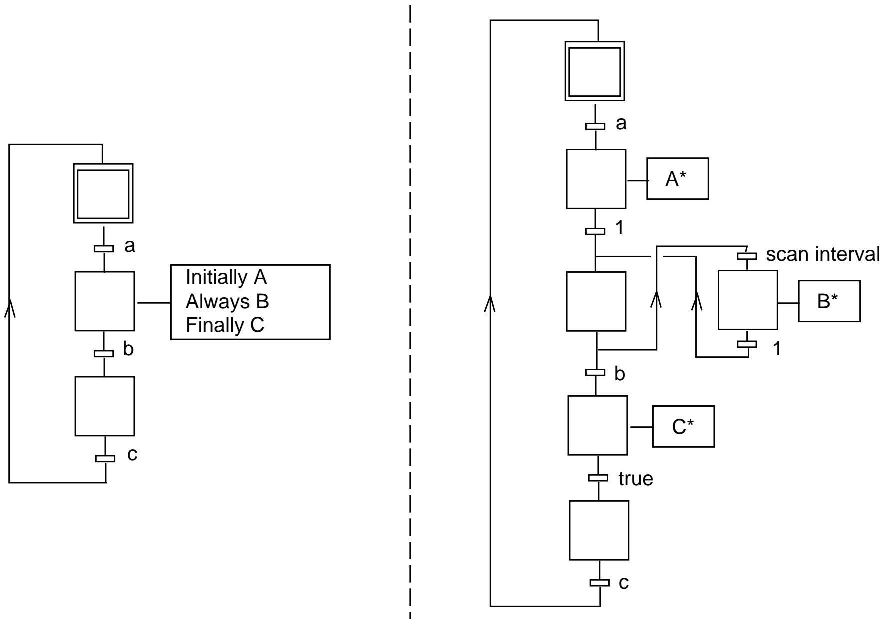


Figure 5.53 A Grafchart(left) and an identical Grafcet (right).

Firing rules: The firing rules are not exactly identical in Grafchart and in Grafcet. A step that has to be simultaneously deactivated and activated remains active in Grafcet. This means that an impulse action associated with such a step in Grafcet will not be executed whereas an initially action associated with such a step in Grafchart will be executed. Some Grafchart structures therefore has to be extended with an extra step in the corresponding Grafcet, see Figure 5.54.

The activation of the extra step assures that the step containing the impulse action is deactivated. A transition with a receptivity that is always true follows the extra step. When this transition fires the step with the impulse action is activated again and its impulse action is executed.

Grafchart - *the high-level version*: Transformation to Petri nets

The main reason for transforming a Grafchart into a Grafcet is to show that a system can equally well be controlled by a Grafchart as a Grafcet. However, the Grafchart structure is, especially for larger and more complex systems, much clearer and better structured than the corresponding Grafcet. A Grafchart structure can also be transformed into a Petri net structure. The main reason for this transformation is analysis. The Graf-

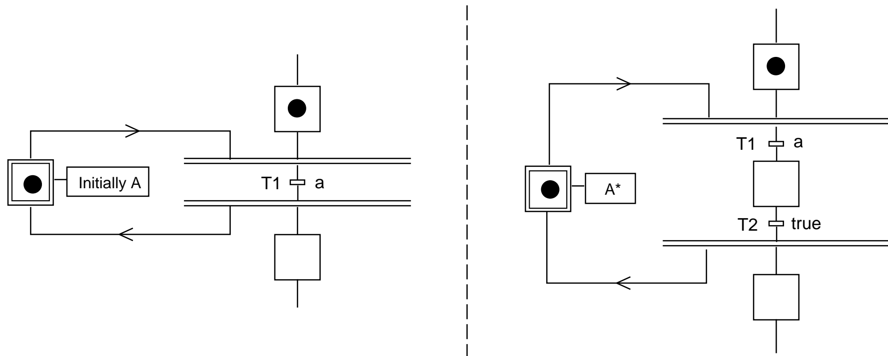


Figure 5.54 A Grafchart(left) and an identical Grafcet (right).

chart to Grafcet transformation consists of three steps; graphical element, action type and firing rule transformation. In the Grafchart to Petri net transformation only the graphical element transformation has to be considered. The action type transformation does not have to be considered. Since a Petri net does not have any actions, the actions are simply neglected. The firing rule transformation has to do with the firing of transitions. Since most analysis methods of Petri nets assumes autonomous Petri nets i.e., Petri nets without transition receptivities, this transformation step can also be left out.

Graphical elements: In order to transform a Grafchart into a Petri net, possibly non-autonomous, the following rules apply:

1. A step is replaced with a Petri net place.
2. A transition remains unchanged.
3. A Grafchart parallel structure or alternative structure are replaced with the syntax of a parallel or alternative structure of Petri net, see Chapter 2.2.
4. The actions are neglected.
5. The receptivities are rewritten from the language of Grafchart to a language used for Petri nets.
6. Macro-steps are replaced with their internal structure.
7. Procedure-steps are replaced with the internal structure of the corresponding Grafchart procedure.
8. Process-steps are replaced with a parallel branch in which the internal structure of the corresponding Grafchart procedure is placed.

9. Parameterization of actions are of no importance because of (4).
10. If parameterization is used in a transition receptivity, this transition must be replaced by an or-divergence situation where there is one branch for each receptivity possibility.
11. If the call to a Grafchart procedure, performed from a procedure-step or process-step is parameterized, each possibility must be analyzed.
12. Method calls are treated in the same way as procedure calls.

Advanced Features

The two features in Grafchart, parameterization and methods, cannot conveniently be reconstructed in Grafcet or Petri nets. The parameterization feature can only be reconstructed by having one Grafcet/Petri net for each parameter setup, which is of course a bad replacement. The method feature can be reconstructed by having ordinary macro steps as replacements for the methods. This will give the same functionality but it will not allow the application to be as well structured and organized.

Analysis

A Grafchart can be transformed into a Petri net and the analysis methods that exist for Petri nets can be applied. This means that a Grafchart can be analyzed with respect to, e.g., deadlock situations, see Chapter 9. The transformation procedure should turn the Grafchart into an autonomous Petri net, i.e., a Petri net without any actions or receptivities. This means that it is only the structure of the Grafchart that can be analyzed see Chapter 9.3.

True formal analysis requires that the step actions and the transition receptivities are taken into consideration. This is not the case in the analysis examples presented in this thesis, see Chapter 9. However, true formal verification support exists for Grafcet, see [Roussel and Lesage, 1996] and [De Loor *et al.*, 1997], and might be of interest also for Grafchart.

There is always a trade off between the complexity of the analysis methods and the completeness of the generated results. The more complex methods, the more complete are the results. Complete results are, of course, always prefered to narrow ones. However, when the analysis methods become too complex, they will be too time-demanding or too complicated to use, which will result in no result at all, and a narrow result is, of course, always prefered to no result at all.

The analysis methods used in the thesis, see Chapter 9 are simple and the generated result does therefore not answer all analysis related questions but the generated result can still be of large interest in many applications.

5.9 Implementation

An implementation of Grafchart has been made in G2, an object oriented programming language, [Gensym Coorporation, 1995]. G2 is briefly described in Appendix E. The Grafchart objects are implemented as G2 objects and the connections are represented by G2 connections.

The implementation of Grafchart is based on the G2 concept of activatable subworkspaces. A workspace is a virtual, rectangular window upon which various G2 items such as rules, procedures, objects, displays, and interaction buttons can be placed. A workspace can also be attached to an object, see Figure 5.55. In this case the workspace is called a subworkspace of that object. When a subworkspace is deactivated, all the items on the workspace are inactive and invisible to the G2 run-time system.

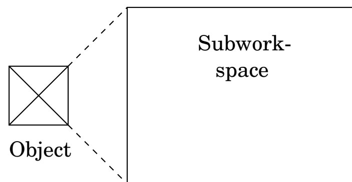


Figure 5.55 An object and its subworkspace.

Grafchart - *the basic version*

Grafchart exists in two different versions, each having its own implementation. Grafchart-BV is the oldest version and was implemented first.

Steps and actions The actions associated with a step are placed on the subworkspace of the step. The actions are represented by G2 rules. The subworkspace of the step is only active when the step itself is active, this means that the rules, placed on the subworkspace, are only executed when the step is active. The actions that may be performed are the action types provided by G2. For example, it is possible to assign values to variables, to start procedures, to create and delete objects, hide and show information, perform animation actions, etc.

The actions are written in the G2's built in rule language. Sometimes, however, the syntax of G2 can be somewhat annoying. To facilitate for the user, the actions are therefore entered as action templates. These are text strings that during compilation are translated into the corresponding G2 rules. The entire syntax of G2 can be used in the action templates together with additional constructs of a syntactical sugar nature. For example, the sometimes long and complicated syntax of certain G2

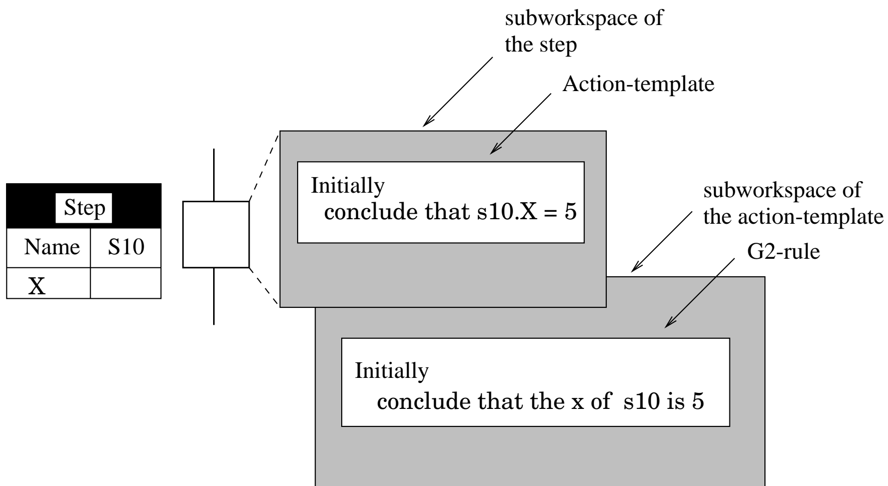


Figure 5.56 A step, an action template and the corresponding G2 rule.

expressions can be replaced by a shorter and simpler expression using a Pascal-like dot notation, e.g. instead of referring to an attribute of an object using the standard syntax of G2 as `the attribute1 of object1` the shorter notation `object1.attribute1` can be used. In Figure 5.56 a step and its subworkspace is shown. On the subworkspace an initially action template is placed. During compilation a subworkspace of the action template is created where the corresponding G2 rule is placed. In Figure 5.56, the subworkspace of the action template and the corresponding G2 rule are shown under the subworkspace of the step.

Transitions and receptivities The condition and event of a transition receptivity are entered as attributes of the transition. During compilation the attributes are automatically translated into an appropriate G2 rule (compare step actions), which is placed on the subworkspace of the transition. The subworkspace is only active when the transition is enabled. A transition that only contains a logical condition is translated into a scanned `when` rule with the shortest scan interval possible. A transition that contains only an event expression and/or an event expression and a logical condition is translated into a `whenever` rule that will fire asynchronously when the event occurs.

When the transition fires, i.e., when the G2 rule condition becomes true and/or the event occurs, the G2 rule starts a procedure that takes care of the activation and deactivation of steps and transitions.

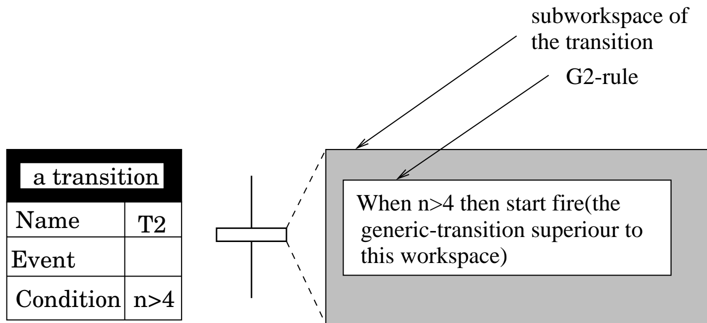


Figure 5.57 A transition, its attributes and the corresponding G2 rule.

In Figure 5.57 a transition is shown together with its `event` and `condition` attributes. The corresponding G2 rule is placed on the subworkspace of the transition as shown in the same figure.

Class hierarchy The objects in Grafchart are defined in classes. The relation between different classes is described by a class hierarchy. Attributes and methods can be associated with a class. A class defined above another class is called a superclass and a class defined below another class is called a subclass. G2 supports multiple inheritance and polymorphism. The graphical language elements in Grafchart are defined in the class hierarchy shown in Figure 5.58.

During compilation the action templates defining the actions of a step and the attributes defining the receptivity of a transition are translated into the corresponding G2 rules. Each class has a method, called compile, that describes how this class should be translated. The translation of a function chart is initialized by a call to a procedure called `compile`. This procedure traverses the function chart and calls the `compile`-method of each object in the function chart.

Interpretation algorithm The interpretation algorithm used in the implementation of Grafchart is based on the one given in Chapter 5.4.1. However, the algorithm given in Chapter 5.4.1 is a global algorithm whereas the one implemented in G2 is local. When a new external event occurs, the global algorithm searches through all receptivities to see which transitions that have become fireable. Since the implementation is based on activatable subworkspaces and G2 rules such a search is not performed. When a new event occurs or a condition becomes true the G2 rule, that corresponds to a receptivity, immediately becomes true and a G2 procedure, called `fire` is automatically started, see Figure 5.57. This procedure

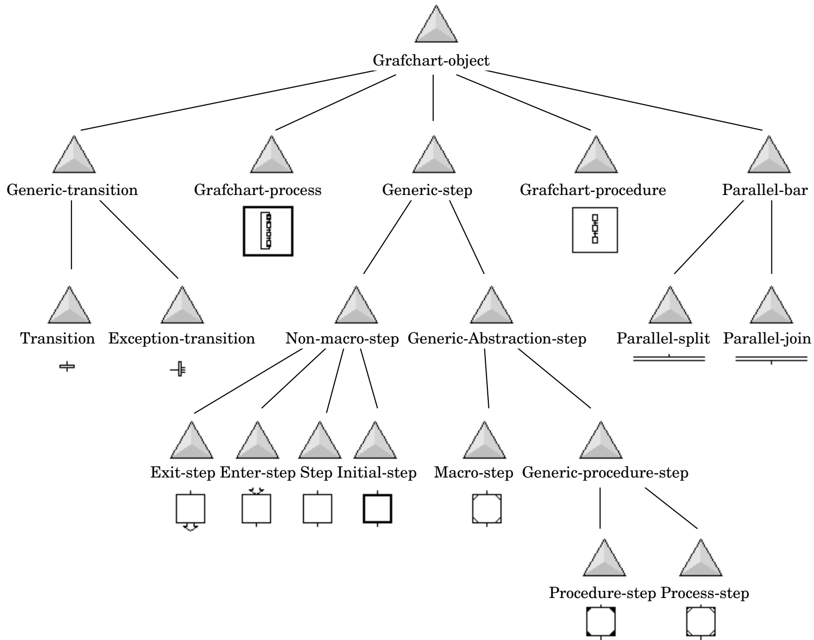


Figure 5.58 The class hierarchy in Grafchart.

takes care of the deactivation and activation of steps and the enabling and de-enabling of transitions. The local algorithm that is used in the implementation is more efficient and less time demanding than the global algorithm. In almost all cases the behavior is equivalent to the behavior of the global algorithm.

EXAMPLE 5.9.1
Figure 5.59 shows a part of a function chart. The function chart has two steps, named $S1$ and $S2$, and two transitions, named $T1$ and $T2$. Step $S1$ is active and step $S2$ is inactive.

When the receptivity of transition $T1$ becomes true the procedure `fire` is started. The procedure calls, in the given order, the following methods: `deactivate-step`, `activate-step`, `de-enable` and `enable`. In Figure 5.59 this corresponds to deactivation of step $S1$, activation of step $S2$, de-enabling of transition $T1$ and enabling of transition $T2$.

#

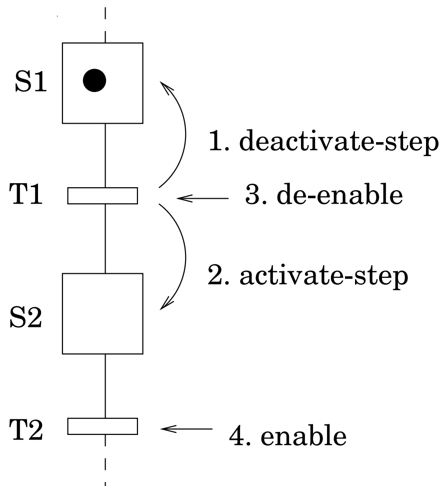


Figure 5.59 Firing of transition $T1$.

Firing of a transition The deactivation and activation of steps as well as the de-enabling and enabling of transitions can be more complicated if there are more than one input or output step or transition and/or if a parallel path or an alternative path is affected by the firing.

When a transition is fired, the following will happen (action execution not taken into account).

1. Deactivate-step
   (a) If the Grafchart object preceding the transition is a parallel-join the deactivation will be done for all Grafchart objects preceding the parallel-join.
   (b) If the Grafchart object preceding the transition is a macro step the deactivation will done for the macro step and the exit step of the macro step.
   (c) If the Grafchart object preceding the transition is a procedure step the deactivation will be done for the procedure step and for the exit step of the corresponding Grafchart procedure.
   (d) If the Grafchart object preceding the transition is a process-step, it will not be deactivated.
   (e) If the Grafchart object preceding the transition is a step, the step will be deactivated.

2. Activate-step

    (a) If the Grafchart object succeeding the transition is a parallel-split the activation will be done for all Grafchart objects succeeding the parallel-split.

    (b) If the Grafchart object succeeding the transition is a macro step the activation will be done for the macro step and for the enter step of the macro step.

    (c) If the Grafchart object succeeding the transition is a procedure step the activation will be done for the procedure step and for the enter step of the corresponding Grafchart procedure.

    (d) If the Grafchart object succeeding the transition is a process-step the activation will be done for the process step, if it is not already active, and for the enter step of the corresponding Grafchart procedure.

    (e) If the Grafchart object succeeding the transition is a step, the step will be activated.

3. De-enable

    (a) The de-enable message is sent to the Grafchart object preceding the transition.

        i. If, however, this Grafchart object is a parallel-join the de-enable message will be sent to each Grafchart object preceding the parallel-join.

    (b) The objects receiving the de-enable message transmit the message to each succeeding Grafchart object.

        i. If, however, this Grafchart object is a parallel-join the de-enable message will be sent to each Grafchart object succeeding the parallel-join.

    (c) The Grafchart object is now a transition, this transition is de-enabled.

4. Enable

    (a) The enable message is sent to the Grafchart object succeeding the transition.

        i. If, however, this Grafchart object is a parallel-split the enable message will be sent to each Grafchart object succeeding the parallel-split.

(b) The objects receiving the enable message transmit the message to each succeeding Grafchart object.

i. If, however, the Grafchart object is a parallel-join the enable call will be sent to each Grafchart object succeeding the parallel-join.

(c) The Grafchart object is a transition, this transition is enabled.

i. If, however, this Grafchart object preceding the transition is a parallel-join the transition will only be enabled if all input places to the parallel-join are active.

EXAMPLE 5.9.2
Consider the function chart in Figure 5.60. Transition $T1$ and $T2$ are enabled. If transition $T1$ fires, the following will happen.

1. Deactivate $S1$.
2. Activate the parallel bar $PB1$, i.e activate the steps $S3$ and $S4$.
3. De-enable the transitions $T1$ and $T2$.
4. Enable the transitions $T3$ and $T4$. Transition $T5$ will not be enabled since $S5$ is not active.

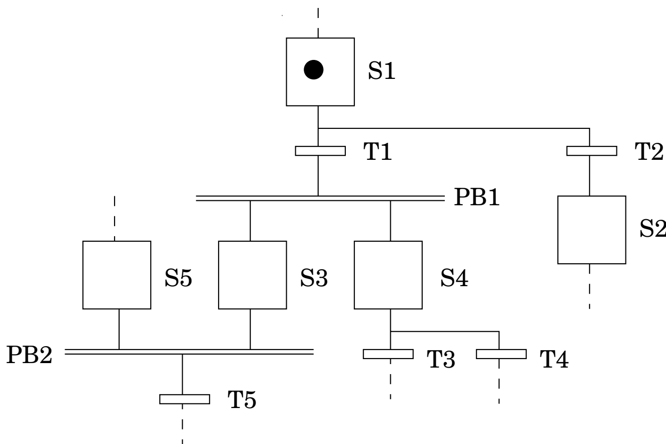


Figure 5.60 The firing of a transition.

#

Grafchart - *the high-level version*

The high-level version of Grafchart is the most recent version of Grafchart. The implementation of this version has tried to mimic the implementation of Grafchart-BV as much as possible concerning the object names, procedure names and the class hierarchy. This facilitates for the users who want to familiarize themselves with the implementational details of both languages.

Steps and actions The actions associated with a step are placed on the subworkspace of the step. The actions are represented by action-templates. Each action-template is related to a token class. During compilation each action-template is translated to a G2 rules. The G2 rule is placed on the subworkspace of the action-template, see Figure 5.61.

When a token enters a step its class is compared with the class specified in the action-templates placed on the subworkspace of the step. If the class of the token and the class of an action-template are the same the subworkspace of the action-template is activated, i.e., the G2 rule is activated. The G2 rule checks if the conditions and requirements for activation are fulfilled. If this is the case, the action is executed.

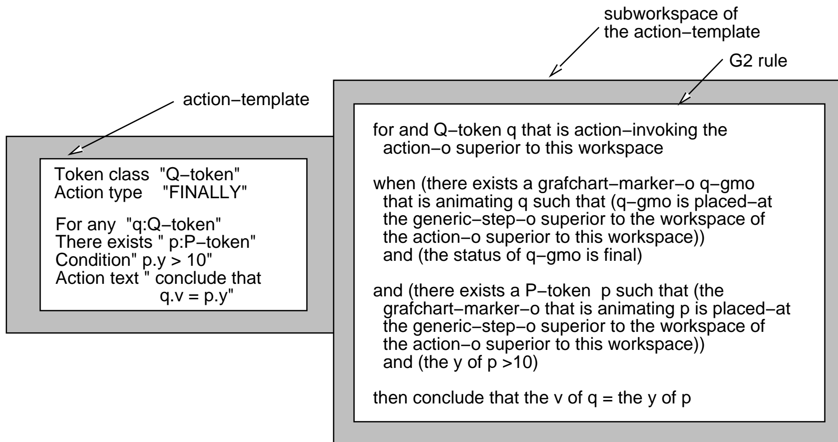


Figure 5.61 Step action and G2 rule.

Transitions and receptivities The receptivities associated with a transition are placed on the subworkspace of the transition. The receptivities are represented by receptivity-templates. Each receptivity is related to a token class. During compilation each receptivity-template is translated to a G2 rules. The G2 rule is placed on the subworkspace of the receptivity-template.

When a token enters a step its class is compared with the class specified in each receptivity-template belonging to the receptivities of the enabled transitions. This means that a receptivity-template associated with token class `Q` is only active if there is a token of class `Q` that enables the transition, i.e., if there is a token of class `Q` in the preceding step of the transition. If the class of the token and the class of a receptivity-template are the same the subworkspace of the receptivity-template is activated, i.e., the G2 rule is activated. The G2 rule checks if the receptivity is true. If so, the receptivity is fired.

Compilation Before a Grafchart-HLV can be executed it has to be compiled. During compilation all actions and all receptivities are translated into a corresponding G2 rules. This means that all `sup` notations and `self` notations are replaced by a G2 expression. Most often the G2 expressions become very complex and hard to read, this explains why the dot notation was introduced. The G2 rule is hidden for the Grafchart user.

The finally action in Figure 5.31 and its corresponding G2 rule is shown in figure 5.61.

Class hierarchy The graphical language elements in Grafchart-HLV are defined in a class hierarchy, see Figure 5.62. The class names are extended with '-o' to indicate that these classes are used in the high-level version of Grafchart.

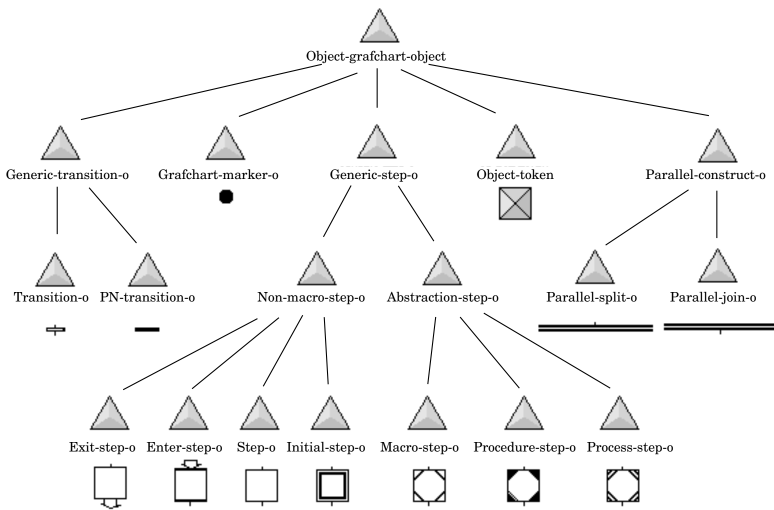


Figure 5.62 The class hierarchy in Grafchart-HLV.

To specify the actions associated with a step the user uses an object called `action-o`, to specify the receptivities associated with a transition the user uses an object called `receptivity-o` and to specify the name of the Grafchart procedure that is to be called from a procedure or process step an object called `procedure-call-o` is used. The class hierarchy of these three template-elements and their icons are shown in Figure 5.63 and 5.64.

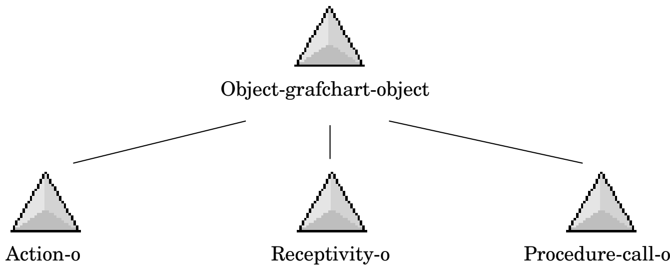


Figure 5.63 The class hierarchy in Grafchart-HLV.

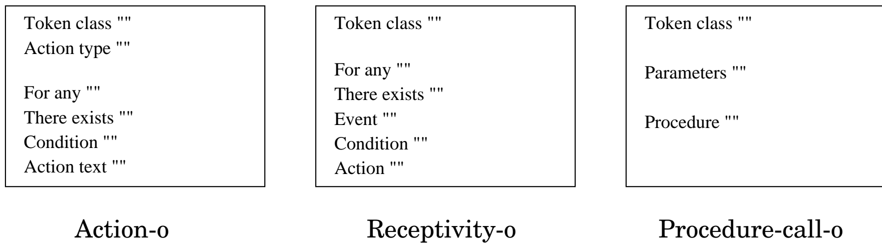


Figure 5.64 The class hierarchy of the template-elements in Grafchart-HLV.

Relations The implementation relies on the use of G2 relations. The following relations are used.

The relation between a grafchart marker and an object token is called `animating`. A grafchart marker may be animating at most one object token. The inverse relation is called `animated-by`. An object token may be animated-by more than one grafchart marker.

The relation between a grafchart marker and a step is called `placed-at` and the inverse relation is called `holding`. A grafchart marker may be placed-at at most one generic-step but a generic-step may be holding more than one grafchart marker.

A relation exists between grafchart markers and receptivities. This relation is called `activating`, the inverse relation is called `activated-by`. A

grafchart marker may be activating more than one receptivity. A receptivity may be activated-by more than one grafchart marker.

A relation exists between object tokens and receptivities. This relation is called `enabling`, the inverse relation is called `enabled-by`. An object token may be enabling more than one receptivity. A receptivity may be enabled-by more than one object token.

The relation between an object token and an action is called `action-invoking`. The inverse relation is called `action-invoked-by`. An object token may be action-invoking more than one action. An action may be action-invoked-by more than one object token.

A relation exists between a grafchart marker and a Grafchart-procedure. This relation is called `marker-invoking` and the inverse relation is called `marker-invoked-by`. A grafchart marker may be marker-invoking more than one Grafchart procedure. A Grafchart procedure may be marker-invoked-by at most one grafchart marker.

5.10 Applications

Grafchart, both the basic and the high-level version, has been used in several applications.

Training Simulator

Grafchart-BV has been used to implement a prototype of a training simulator for a sugar crystallization process, [Nilsson, 1991]. In this application Grafchart is used both to structure the simulation model of the process and to implement the control system of the process.

Hydrogen Balance Advisory Control

Grafchart-BV has been used to implement a knowledge-based system, (KBS), that generates on-line advice for operators regarding the distribution of hydrogen resources at the Star Enterprise Delaware City Refinery [Årzén, 1994a]. The system uses KBS techniques coupled with numerical optimization. The specific problem that is solved is to meet the needs of the hydrogen consuming units in the refinery while minimizing the hydrogen that is wasted. A catalytic reformer unit produces hydrogen as by-products. A hydro cracker unit consumes high purity hydrogen and vents low purity hydrogen. Hydrogen from these units is used to satisfy the needs of the hydrogen consuming hydro treaters, sulphur recovery, methanol, and naphtalene units. Any additional hydrogen needs must be met by a hydrogen production unit.

Flexible Manufacturing Cell

In [López González *et al.*, 1994] a system is described were Grafchart-BV is used to implement a flexible manufacturing cell. Grafchart is used in a four-layered hierarchical structure to represent the plant-wide operating phases of the control system, to describe the sequences of tasks to be executed to manufacture the parts, to describe the tasks at the workstation level and, finally, to describe the different services offered by the device drivers in the cell.

Lego Car Factory

A Grafchart model of the control system for a LEGO car factory has been implemented, using an early version of Grafchart-HLV. The LEGO car factory is a small car factory model built in LEGO that assembles toy LEGO cars. The factory itself is built in LEGO and consists of four conveyor belts, three storages for chassis, car frames, and car bodies respectively, two pressing machines, three machines that take parts from storage, and one machine that turns the car on the conveyer belt. The factory is supplied with several position sensors.

The model of the control system consists of two function charts. The car-controller is a straight sequence where each token represents a car. Each macro-step represents a certain operation that is performed on the car. The machine-controllers have one token for each machine. The actions and conditions in the car-controller depend on the presence of a car in a certain step. The control system is structured into one machine part and one product part. The product in this case is the cars. Similar structuring concepts can also be applied to general discrete manufacturing problems. The LEGO car factory is presented in [Årzén, 1994a].

Alarm Filtering

Grafchart-HLV has been used to implement alarm filters. Information overload due to large numbers of warnings and alarms is a common problem in, e.g., the process industry. This typically occurs in fault situations when a primary alarm is accompanied by large numbers of secondary alarms. Another case is nuisance alarms caused by improperly tuned alarm limits. One way of handling excess alarms is to use alarm filters. The task of the alarm filter is to filter out alarms and to aggregate multiple alarms into high-level alarms. In many cases it is the combination of multiple alarms that have occurred within a certain time period that gives the best indication of what is happening in the process.

Grafchart can be used to represent alarm (event) patterns. The approach is based on the possibility to use a finite state machine as an acceptor

for strings in a formal language. Since Grafchart is an extended state machine with temporal facilities it is possible to accept more general expressions than the regular expressions that are possible with ordinary state machines.

Due to the sampled nature of the control system and to measurement noise, in some cases the relative ordering between a pair of low-level alarms may be unimportant. Situations also exist when one wants to specify that n out of m alarms should have occurred, or that some alarms in a pattern could be omitted. By utilizing the possibilities in Grafchart for parallel and alternative paths, different sequence patterns of these types can be easily expressed. Situations when temporal constraints between alarms are important can also be expressed with Grafchart. Representation of different event patterns in Grafchart is presented in detail in [Årzén, 1996].

Batch Recipe Structuring

Batch recipe structuring is the main application area of Grafchart. Grafchart is used at two levels, to represent the sequential structure in the recipes and to represent the sequence control logic contained in the equipment units. A number of different ways to structure the recipes, using either Grafchart-BV or Grafchart-HLV, have been investigated, see Chapter 8.

5.11 Summary

Grafchart is the name of a mathematical sequential control model and a toolbox. Grafchart is based on a syntax similar to that of Grafcet. The graphical elements are steps, transitions, macro steps, procedure steps, process steps, Grafchart procedures and Grafchart processes. Grafchart also includes ideas from Petri nets, high-level nets and object-oriented programming. Advanced features like parameterization, methods and message passing, object tokens, and multi-dimensional charts, are included in the language. The language exists in to different versions: one basic version, mainly based upon Grafcet, and one high-level version including more ideas from Petri nets and object-oriented programming. Grafchart is aimed at sequential control application at local and supervisory level.

6

Batch Control Systems

Industrial manufacturing and production processes can generally be classified as continuous, discrete or batch, [Fisher, 1990]. How a process is classified depends mainly upon how the output from the process appears.

In continuous processes, products are made by passing materials through different pieces of specialized equipments. Each of these pieces of equipment ideally operates in a single steady state and performs one dedicated processing function. The output from a continuous process appears in a continuous flow, [ISA, 1995].

In discrete manufacturing processes, products are traditionally manufactured in production lots based on common raw materials and production histories. In a discrete manufacturing process, a specified quantity of products moves as a unit (group of parts) between workstations and each part maintains its unique identity, [ISA, 1995]. The output from a discrete manufacturing process appears one by one or in quantities of parts.

A continuous process might be, e.g., water purification plants, paper mills or the generation of electricity from a power plant, whereas a discrete manufacturing process could be, e.g., the production of cars.

In batch manufacturing the output appears in lots or in quantities of materials. The product produced by a batch process is called a batch. Batch processes are neither continuous nor discrete, yet they have characteristics from both.

An example of a batch process, [Rosenhof and Ghosh, 1987], taken from our daily life, is the preparation of a cake. The work can be divided into three major tasks: preparation, baking, and cooling and storing. The three tasks can be broken down into a sequence of substeps. The steps in each task should be done in a proper order to make a good cake. If not done in this way, the cake might not be very tasty and, as explained below, do not adhere to the formal definition of a batch process.

Batch processes define a subclass of sequential processes. The difference is that sequential processes not necessarily have to generate a product whereas batch processes do. In industry, batch and sequential processes are used in many ways and in many areas: food engineering, pharmaceutics, biotechnical manufacturing and chemical plants.

A formal definition of batch process is given by [Shaw, 1982]:

> *A process is considered to be batch in nature if, due to physical structuring of the process equipment or due to other factors, the process consists of a sequences of one or more steps that must be performed in a defined order. The completion of this sequence of steps creates a finite quantity of finished product. If more of the product is to be created, the sequence must be repeated.*

An other definition is given by [ISA, 1995]:

> *A batch process is a process that leads to the production of finite quantities of material by subjecting quantities of input materials to an ordered set of processing activities over a finite period of time using one or more pieces of equipment.*

6.1 Batch Processing

Continuous processes have been used to produce many products that were originally produced by batch processes. One reason for this is that batch processes have typically been labor intensive and experienced operators have been required to produce batch products with consistent quality. Since continuous plants were those that produced the largest volume of products this is where most of the research and development money were spent. However, increasing demands on flexibility and customer-driven production has led to an increased interest in batch processes. This is because batch processes are more economical for small scale production, as fewer pieces of process equipment are needed, and intermediate storage are not very expensive. Batch plants can also be made highly flexible, and thereby well suited for manufacturing of special products. For example, high quality malt whisky is produced in batch processes whereas grain liquor, the basis for blended whisky, is produced in continuous processes.

There are also processes that are not easily amenable to continuous operations. Some examples are given in [Rosenhof and Ghosh, 1987]:

1. Processes with feedstocks and/or products that can not be handled efficiently in a continuous fashion, such as solids and highly viscous materials.

2. Processes in which the reactions are slow, requiring the reactants to be held in process vessels for a long time (e.g. fermentation of beer and wine).
3. Processes in which only small quantities of products and/or different grades of the same product are required in limited quantities (e.g. dyestuff and specialty chemicals).
4. Processes that need precise control of raw materials and production along with detailed historical documentation (e.g. drug manufacturing).

Typically, batch plants are used to manufacture a large number of products. Within each product a number of different grades often exist.

Batch processes can be classified by: (1) the number of products they can make and (2) the structure of the plant, [Fisher, 1990].

1. A batch process can be single-product, multi-grade or multi-product. A single product batch plant produces the same product in each batch, e.g., a batch pulp digester. The same operations are performed on each batch and the same amount of raw materials is used. A multi-grade batch plant produces products that are similar but not identical. The same operations are performed on each batch but the quantities of raw materials and/or processing conditions such as, e.g., temperatures, may vary with each batch. The multi-product batch plant produces products utilizing different methods of production or control. The operations performed, the amount of raw materials and the processing conditions may vary with each batch.
2. The basic types of batch structures are series (single-stream), parallel (multi-stream) and a combination of the two. A series structure is a group of units through which the batch passes sequentially. If the plant has several serial groups of units placed in parallel but without interaction the plant has a parallel structure. If interactions exists between the parallel branches, a series/parallel structure is achieved. Other names for the series, parallel and series/parallel structures are single-path, multi-path and network-structure, [ISA, 1995].

The batch plant classification by product and by structure can be combined in a matrix to show the degree of difficulty in automating the various combinations. The single-product, single-path batch plant is the simplest, whereas the multi-product, network-structure combination is the most demanding.

6.2 Batch Control

Batch control projects have traditionally been among the most difficult and complex to implement [ARC, 1996]. Typically, batch control projects span over a wider scope of functionality than that required for either continuous or discrete manufacturing processes. With continuous and discrete processes, a reasonable level of automation can be attained merely by implementing basic regulatory or logic control. Batch operations typically require basic regulatory and logic control operating under sequential control; which in turn, is operating under basic recipe management in order to achieve process automation. The complexity of control within a process cell depends on the equipment available within the process cell, the interconnectivity among this equipment, the degree of freedom of movements of batches through this equipment, and the arbitration of the use of this equipment so that the equipment can be used most effectively [ISA, 1995].

The discussion about batch control systems and the progress in batch process control has been hampered by the lack of a standard terminology. In the last years, there have been three major initiatives with the aim to provide a common language. The first major effort was an outgrowth of a Purdue University workshop on batch control in the mid 1980s, [Williams, 1988]. The second was made by NAMUR, [NAMUR, 1992]. The third major effort is sponsored by the Standards and Practices division of ISA (Instrument Society of America), the International Society for Measurement and Control, [ISA, 1995]. The standard is divided into two parts, Part 1, called S88.01 deals with models, terminology and functionality. This part of the standard was approved by the main committee of ISA and ANSI in 1995. Part 2 will deal with data structures and language guidelines. The ISA S88.01 standard has recently been accepted as an international IEC (International Electrotechnical Commission) standard IEC 61512-1, [IEC, 1997].

6.3 The Batch Standard ISA-S88.01

The first part of the ISA S88 standard describes batch control from two different viewpoints: the process view and the equipment view, see Figure 6.1. The process view is represented by the process model and is normally the view of the chemists. The equipment view is represented by the physical model and is normally the view of the product engineer or the process operator.

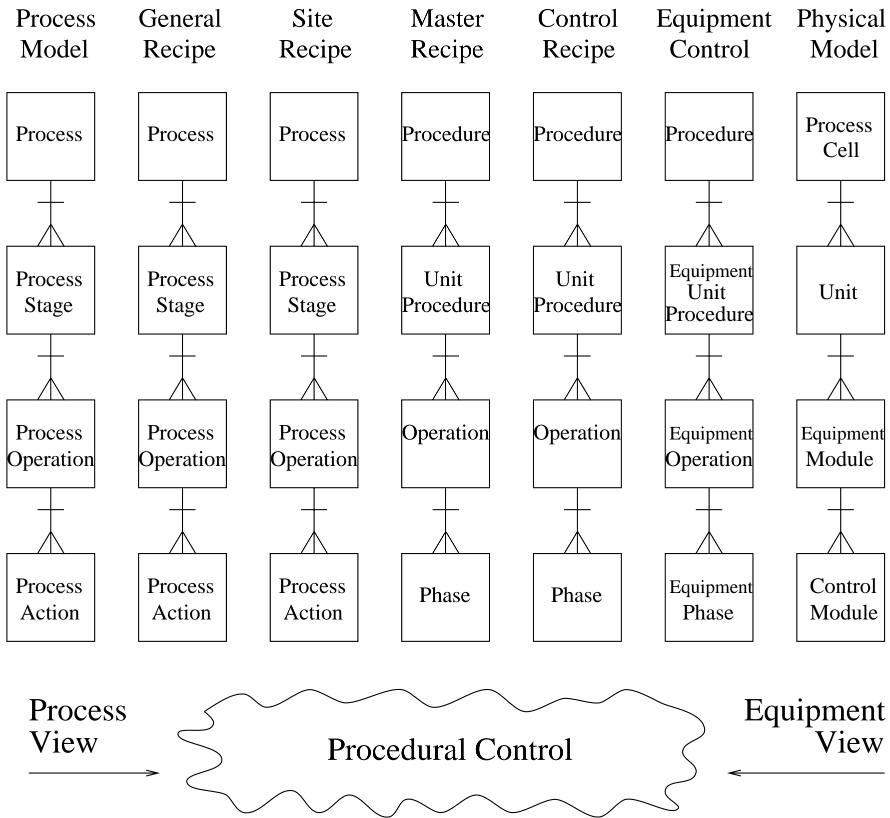


Figure 6.1 Relations between ISA-S88 models and terminology.

Process Model

A batch process can be hierarchically subdivided as shown in Figure 6.1 (left).

- The process consists of an ordered set, serial and/or parallel, of process stages. A process stage is a part of a process that usually operates independently from other process stages. It usually results in a planned sequence of chemical or physical changes in the material being processed. Typical process stages can be, e.g., drying or polymerizations.
- Each process stage consists of an ordered set of one or more process operations. Process operations describe major processing activities. It usually results in chemical or physical change in the material

being processed. A typical process operation is, e.g., react.

- Each process operation can be subdivided into an ordered set of one or more process actions that carry out the processing required by the process operation. Process actions describe minor processing activities that are combined to make up a process operation, Typical process actions are, e.g., add reactant, and hold.

In the process model, the procedure for making a product does not consider the actual equipment for performing the different process steps.

Physical Model

The physical model of S88 defines the hierarchical relationships between the physical assets involved in batch manufacturing. The model has seven levels, starting at the top with an enterprise, a site and an area. In Figure 6.1 (right) only the four lower levels are shown, with the following interpretation:

- A process cell contains one or more units.
- A unit can carry out one or more major processing activities such as react, crystallize or make a solution. Units operate relatively independently of each other. A unit is made up of equipment modules and control modules.
- An equipment module can carry out a finite number of minor processing activities like weighting and dosing. It combines all necessary physical processing and control equipment required to perform those activities. Physically, an equipment module may be made up of control modules and subordinate equipment modules. An equipment module may be part of a unit or may be a stand-alone equipment grouping within a process cell. It may be an exclusive use resource or a shared resource.
- A control module is typically a collection of sensors, actuators or controllers. Physically, a control module can be made up of other control modules.

Recipes

To actually manufacture a batch in a process cell the standard proposes a gradually refinement of the process model based on four recipe types; general recipe, site recipe, master recipe and control recipe. A recipe contains administrative information, formula information, requirements on the equipment needed, and the procedure that defines how the recipe should be produced. The procedure is organized according to the procedural control model, see Chapter 6.3 (Procedural Control).

- General recipe
  The general recipe is an enterprise level recipe that serves as a basis for the other recipes. The general recipe is created without specific knowledge of the process cell equipment that will be used to manufacture the product.

- Site recipe
  The site recipe is specific to a particular site. The language in which it is written, the units of measurements, and the raw materials are adjusted to the site.

- Master recipe
  The master recipe is targeted to a specific process cell. A master recipe is either derived from a general recipe or created as a stand-alone entity by people that have all the information that otherwise would have been included in the general or the site recipe.

- Control recipe
  The control recipe is originally a copy of the master recipe which has been completed and/or modified with scheduling, operational and equipment information. A control recipe can be viewed as an instantiation of a master recipe.

The four recipes are gradually refined to the stage where all necessary aspects for the execution of the recipe on a certain type of equipment are taken into account. The general and site recipes are equipment independent whereas the master and control recipes are equipment dependent. In order to distinguish between equipment independent and equipment dependent process steps different terminology is used. The terms Procedure, Unit Procedure, Operation, and Phase are introduced for the equipment dependent process steps, see Figure 6.1.

Procedural Control

The four recipe levels together with the equipment control constitute the link between the process model and the physical model, denoted procedural control, see Figure 6.1. Procedural control is characteristic for batch processes. It directs equipment-oriented actions to take place in an ordered sequence in order to carry out a process-oriented task.

The procedural structure is hierarchical, see Figure 6.2. A procedure can gradually be broken down into smaller parts.

- The procedure is the highest level in the hierarchy and defines the strategy for accomplishing a major processing action such as making a batch. It is defined in terms of unit procedures, and/or operations,

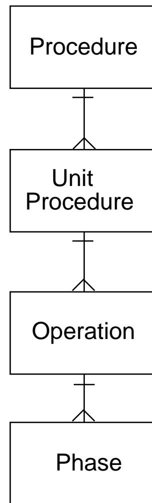


Figure 6.2 Procedure Control.

and/or phases. An example of a procedure is "Make a batch of product A".

- A unit procedure defines a set of related operations that causes a production sequence to take place within a unit. Examples of unit procedures are, e.g., polymerize, recover or dry. A unit procedure must be executed within a single unit process.
- An operation is a sequence of phases that defines a major processing sequence that takes the material being processed from one state to another. An operation usually involves a chemical or physical change. Examples of operations are, e.g., reaction and preparation.
- The smallest element that can accomplish a process oriented task is a phase. It defines a product independent processing sequence. A phase may be decomposed into steps and transitions according to Grafcet/SFC. Examples of phases are, e.g., add catalyst.

The IEC 1131-3 standard, which was published in 1993, specifies programming languages for controllers. This standard fills an important void since part 1 of ISA-S88 does not specify languages for configuring the sequential and batch control functions. Many suppliers have however, already incorporated the IEC 1131-3 standard in their products.

Sequential Function Charts (SFC) are gaining acceptance for configuration of the procedural part of recipes. The main reasons for this are that

SFC are graphical, easy to configure and easy to understand. SFC is also the basis for Procedural Function Charts (PFC), a graphical language for recipe representation currently being defined in the S88 working group.

Equipment Control

The control recipe itself does not contain enough information to operate a process cell. On some level it must be linked to the process equipment control, i.e., the control actions that are associated with the different equipment objects. S88 offers large flexibility with respect to at which level the control recipe should reference the equipment control. It is also allowed to omit one or more of the levels in the procedural model. The situation is shown in Figure 6.3. The dashed levels could either be contained in the control recipe or in the equipment control. Several examples of how this can be done will be given in Chapter 8.

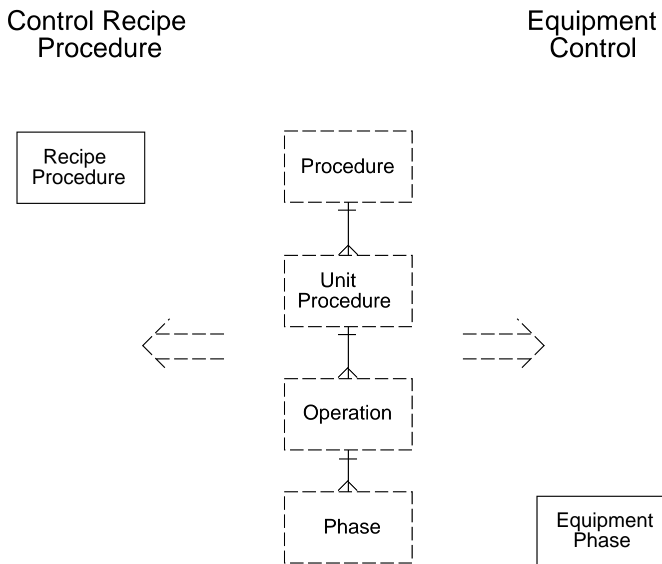


Figure 6.3 Control recipe/ Equipment control separation.

Control Activity Model

To successfully manage batch production many control functions must be implemented. The control activity model shown in Figure 6.4, identifies the major batch control activities and the relationships amongst them. This model was outlined in S88, and provides an overall perspective on batch control.

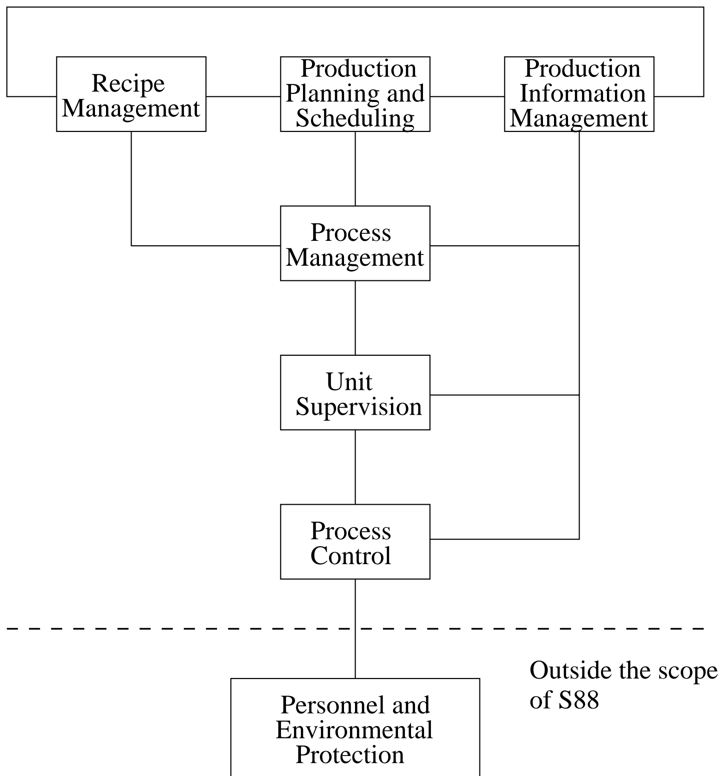


Figure 6.4 The control activity model of ISA-S88.

The control activities shown relate to real needs in a batch manufacturing environment.

- Recipe Management
  The need to have control functions that can manage general, site, and master recipes implies a need for the recipe management control activity.

- Production Planning and Scheduling
  Production of batches must occur within a planned time domain. Production planning and scheduling is the control activity where these control functions are implemented. Scheduling is discussed further in Chapter 6.4.

- Production Information Management
  Various types of production information must be available and the

collection and storage of batch histories is a necessity. The production information management control activity in the model covers these control functions.

- Process Management
  Control recipes must be generated, batches must be initiated and supervised, unit activities require coordination and logs and reports must be generated. These control functions fall under the process management control activity.

- Unit Supervision
  The need to allocate resources, to supervise the execution of operation and phases, and to coordinate activities taking place at the process control level are examples of control functions needed at the unit supervision control activity level.

- Process Control
  The Process Control control activity discusses control functions that deal directly with equipment actions.

6.4 Scheduling

Scheduling is a wide concept that can be broken down hierarchically, [Fisher, 1990], see Figure 6.5.

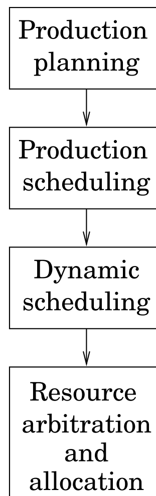


Figure 6.5 The scheduling hierarchy.

- At the highest level, scheduling is used to assign product orders to the various manufacturing plants. At this planning level, the horizon may be measured in weeks or month.
- The order is sent to the production scheduling system in a plant. Here, the inventories are checked against the production demands and it is determined how much raw material that needs to be ordered. A master recipe is created. The production scheduler is run once a day or once a week.
- At the dynamic scheduling level, it is necessary that the schedule more closely approximates real-time. Here, the horizon is measured in shifts, hours or minutes. A detailed schedule based on the specific resources and requirements of the batch production system is generated. The scheduling must be repeated if any activity is not completed in time (delayed, ahead of time). It may also be necessary to repeat the scheduling because of equipment malfunctions, lack of raw materials, etc.
- The resource arbitration system prevents a piece of equipment from being used by more than one user at a time. The resource arbitration system comes into play when it is necessary to decide which product is the most deserving of access to a shared resource that is being allocated by two resources at the same time. The resource allocation system takes care of the allocation of resources. Here, a simple first-come-first-served algorithm is not always sufficient.

Different scheduling techniques exist; linear programming, material requirement planning, simulation, expert systems. However, most batch plants are still scheduled using human made schedules.

6.5 Industry Practice

Many different commercial batch control systems exist today. They are produced and utilized all around the world. Some examples are: FlexBatch from GSE Systems, OpenBatch from PID Incorporate, VisualBatch from Intellution, FoxBatch from Foxboro, InBatch from Wonderware, SattBatch from Alfa Laval Automation, Advant from ABB Industrial Systems, etc.

The batch control system suppliers are well aware of the ISA S88 batch control standard and their systems do, to greater or lesser extent, depending on the age of the system, comply with it. Even thought the different systems have been developed at different independent companies they

all have, from an operator point of view, a common look-and-feel. This is probably because of (or thanks to) the standard.

Recipe Structures

The recipe structure in all the different systems is composed of steps and transitions. The steps are represented by rectangles and the transitions are represented by horizontal bars or small rectangles, see Figure 6.6.

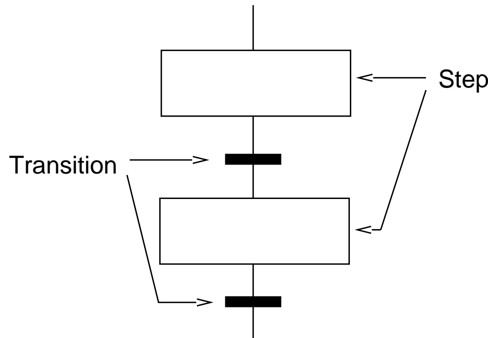


Figure 6.6 Building blocks in commercial batch control systems.

The steps visualize the different tasks that should be performed when producing a batch and the transitions represents a change of task. This resembles the general ideas in Grafcet.

However, the steps and the transitions do not necessarily appear one after the other. In most of the systems, a step may be connected to another step and a transition may follow directly upon another transition. The reason for this “strange” syntax is, according to the developers, that not all steps and transitions in a Grafcet have a meaning. There are cases where the step is only a dummy-step or the transition represents “the-always-true-condition”; these steps and transitions can be omitted without loosing any functionality. Some examples are given in Figure 6.7.

A recipe is always structured in such a way that the step that should be activated when the recipe is initialized is placed in the top position (uppermost position), and the recipe executes from the top position to the bottom position, see Figure 6.8.

Recipe Execution

The execution of a recipe is normally color-coded. Even though the colors are not fixed but can be chosen by the operator, the default setting is always the same. The step(s) that is currently executing is green. The

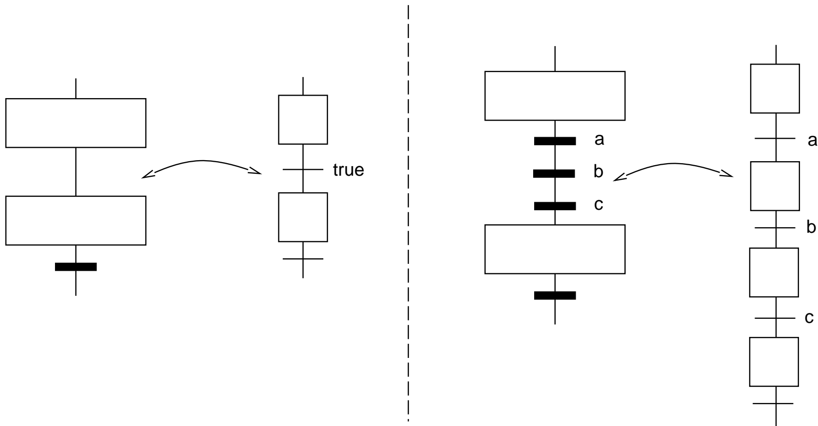


Figure 6.7 A commercial batch recipe structure (left) and the corresponding Grafcet structure (right).

step(s) that has been executed is red and a step that has been halted, paused or is waiting for any other reason is yellow. The execution of the recipe structure is thus not visualized by a token that moves around in the net, like in Grafcet or Grafchart, but with different colors.

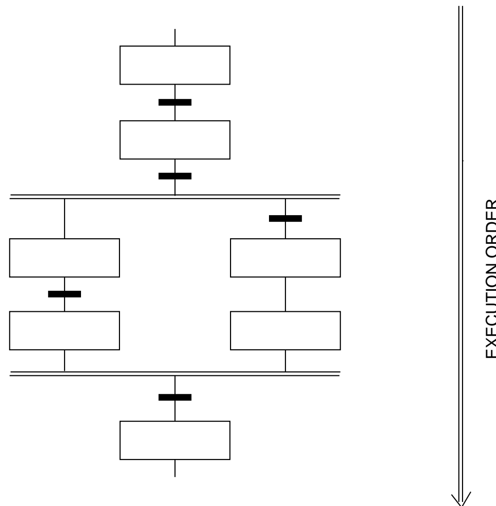


Figure 6.8 Execution order; top to bottom.

Recipe Implementation

The recipes are implemented according to the hierarchical procedural model. When a recipe is first displayed, each step represents a unit-procedure. If a step, corresponding to a unit-procedure, is “opened-up” a sequence of steps and transitions shows up, compare with macro-steps in Grafcet. In this sequence the steps correspond to an operation. There is no special syntax that indicates for the user that a step can be "opened-up". All commercial systems support three hierarchical levels: unit-procedures, operations, and phases. In almost all systems it is required that all levels are implemented, even thought this is not required in the standard. A step that represents a phase can not be opened. The phases are most often implemented in a textual language.

Plant Status

Even though a single recipe can be represented graphically, the overall status of the plant is always represented textually. Each recipe is named by a tag name followed by some information about its status (run/hold/to-be-started/completed). In Figure 6.9 an example of how the screen can look for the user is given. In the upper left corner of the screen a textual view of the plant status is shown. Five batches are under production. One is completed, two are executing, one is halted and one is not yet started. Next to the textual view of the plant a recipe structure is shown, this corresponds to the procedure for the recipe named Recipe-D-001. The first unit-procedure has executed while the second step is halted. This is indicated with the colors of the respective step.

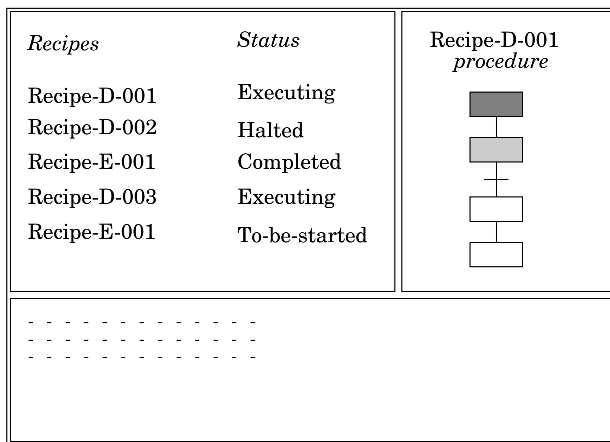


Figure 6.9 Textual view of the plant status (upper right corner).

The reason for having a textual representation is, according to the developers, that it would be to messy to display all recipes at the same time on the screen. A textual overview is less space demanding and gives thus a better overview. The user can select one or several of the recipes from the list and display them graphically.

Resource Allocation

Associated with each recipe is a list of resource requirements. The user can either specify a resource by directly type in the name of the desired resource or by specifying a group of resources that are all equally good. No commercial systems of today visualize the resource allocation.

It is possible for the user to specify when the required resource should be allocated. Either all resources that are going to be used when producing a batch can be allocated immediately when the recipe is started or the resources can be allocated one-by-one just before they are needed for production. The first method is to prefer if the batch is very sensible and valuable and therefore not should be exposed for a waiting-period. The method guarantees that the batch will have the resources when needed. In the second method, the allocation is done just before the resource is needed. It might then happen that a resource is not available, it could, e.g., be allocated by another batch. In that case the batch either has to allocate another resource or wait until the desired resource becomes available.

Using the first method, the risk of getting into a deadlock situation is removed. On the other hand the resources are unavailable for other batches for an unnecessary long period of time. Using the second method, the operator must manually assure that the production does not run into a deadlock situation. In no commercial system there is support for deadlock detection.

Phase Implementation

The language chosen for the phase implementation differs from enterprise to enterprise and depends on their type of PLC's.

The batch control system including the recipe system is sold by a batch control system vendor. The vendor helps the enterprise to structure the supervisory implementation of the recipes. However, the implementation of the phases, i.e., the actual local equipment control, is done by the enterprise themselves.

Interlocks and safety assurances are most often taken care of within the phase logic.

Compliance to PLC's

Not all batch control systems are compliant with all PLC types. Certain batch control systems require that the PLCs are of a certain type. Most often the PLCs are required to be bought from the same vendor as the batch control system. However, commercial batch control systems exist that are PLC-type independent, e.g., OpenBatch from PID Incorporate and InBatch from Wonderware.

Distributed Control Systems

The batch control system is implemented in a distributed control system. A general problem is how data should be transfered from one node in the system to another. Today many different solutions exist. The data that should be transfered consist of both logged data and batch specific information such as equipment requirements.

Remarks

The way a recipe is structured in commercial systems reminds of the structure of Grafcet. Even though the similarities are obvious no explicit connection exists. No batch system vendor has, as far as we know, experience or knowledge about the theory of Grafcet and/or Petri nets. The similarities have occurred by mere accident. The many similarities do however, confirm that the graphical syntax of Grafcet is very clear and intuitive.

6.6 Summary

Industrial manufacturing and production processes can be classified as continuous, discrete or batch. Traditionally the areas of continuous and discrete processes have been the most research intensive areas. However, increased demands on production flexibility, small scale production and customer-driven production have led to an increased interest for batch processes and batch control. A batch process is neither continuous nor discrete, yet it has characteristics from both. A batch cell can be made highly flexible, both with respect to the number of different products it can produce and with respect to the structure of the plant. The multi-product, network-structured batch cell is the most flexible but also the most difficult type of plant to control. The recent batch control standard ISA-S88.01, formally defines the terminology, models and functionality of batch control systems. The standard mentions the possibility to use Grafcet but only at the very lowest level of control. However, there is also

a need for a common representation format, complying with the standard, at the recipe level.

Several different commercial batch control systems exist today. All suppliers are well aware of the ISA S88 standard. The recipes are structured according to the hierarchical procedural model. Steps and transitions are used for visualizing tasks and the changing of tasks. The recipe structure reminds of a Grafcet structure, however, no formal links exists, the similarities have occurred by mere accident. Most commercial batch systems of today lack a user-friendly way of presenting the overall status of a plant, neither do the systems have an efficient way of handling the resource allocation. These concerns are treated further in Chapter 8.7.

7

Batch Control Systems and Grafchart

The aim of this thesis is to show various ways in which the concepts of Grafchart can be used in the context of recipe-based batch control. The aim is not to present a complete batch control system. As will be shown Grafchart can be used at all levels in the hierarchical procedure model, from the PLC level sequence control to the representation of entire recipes. Grafchart also facilitates the linking that has to exist between the control recipe procedural elements and the equipment control procedural elements.

A batch scenario, consisting of one batch cell, has been defined and implemented in G2. The batch scenario is used as a test platform to investigate how Grafchart can be used in a batch control system.

This chapter starts with a presentation of related work. The common point of the works presented, is that they are all based on Petri nets or Grafcet. The chapter also contains a presentation of how Grafchart can be used in a batch control system and, it contains a description of the defined batch scenario.

7.1 Related Work

Batch processes receive increasing interest in industry as well as the academic control community. This can be noticed by the increased number of researchers in the field. The control of batch processes can be attacked in various ways. In this section an overview of related work and activities, all with Petri nets or Grafcet as a basis, is presented. The presentation does not try to cover the field completely, other activities, not presented in this section, but yet interesting, most likely exist.

By modeling batch plants and recipes mathematically and by applying design methods to these models control laws can be generated. The generated control laws should ensure safe and correct operations. The PhD-thesis by Tittus, [Tittus, 1995], proposes both models, frameworks and design methods for this task.

The plant is modeled by a Petri net describing all physically possible connections between the process units. Each process unit is modeled by an individual PN presenting the state of this unit. The recipe for a batch, is modeled by a PN describing all possible execution paths through the plant. If more than one batch is to be produced at the same time, the transitions of the Petri nets of all recipes are interleaved and one PN describing all possible intermixings of the recipes is achieved. This net is synchronized with that of the plant, forming the control recipe (Note: the terminology is not the same as that of ISA-S88.01). The control recipe is reduced to all physically possible intermixings of the recipes and the plant by considering the fact that each resource can only be used by one recipe at a time. Further reduction can be done by removing the places and transitions that can lead to an unsafe or a deadlock situation. From the reduced control recipe, a discrete supervisor, that guarantees correct and parallel execution of the recipes, can be generated.

The work by Tittus differ from that in this thesis in that it is more focused upon formal verification and synthesis, see Chapter 1. The aim is to generate a safe control recipe and no effort is put into the representation or implementation of the recipes.

Wöllhaf and Engell have developed an object-oriented tool for modeling and simulation of the production process in a recipe-driven multipurpose batch plant, [Engell and Wöllhaf, 1994]. Models of the production plant, the recipes and the batches of material are developed. Both continuous and discrete aspects of the simulation are included in order to support the supervision of the plant and the scheduling of production tasks. The work follows the German NAMUR standard, [NAMUR, 1992], see Chapter 6.2. The control recipe, represented with Sequential Function Charts, essentially contains the discrete model which describes the production steps and the transitions. The plant model together with the batches of material, constitute the continuous system. By substituting the basic functions of the recipe by the technical functions of the plant, a hybrid system, possible to use for simulations, is created.

The focus of the work by Wöllhaf and Engell is modeling and simulation. They focus upon the mathematical algorithms used to solve the differential and algebraic equations for simulation of the materials. A large

database for physical and chemical properties of the most common substances is included in their tool. However, they do not focus upon the recipe management system and the issue of making the recipes reusable and flexible.

The aim of the work by Hanisch and Fleck is to join the theoretic work on resource allocation problems with the need of such strategies in recipe-control framework (recipe-based control systems), [Hanisch and Fleck, 1996]. They use high-level Petri nets to model both the recipe-based operations and the resource allocation strategy. The resource allocation module is implemented separately within the process control system. The strategy of the resource allocation module is to optimize the productivity. Simulation of the system allows the control engineer to answer questions about the utilization of resources, bottlenecks and resource dimensions.

Superbatch is a scheduling engine developed at Imperial College. Superbatch matches the demand for products to available resources in order to find what and how much that should be made and also when and where it can be made. This task is done by the off-line planner in Superbatch. Superbatch executes the off-line schedule by sending the appropriate commands and parameters to a control system. However, a change in demand, a variation in process yield or resource availability will cause the actual production to deviate from the original schedule. Superbatch therefore includes also an on-line monitor that updates the schedule to match the current status of the plant. It alerts messages to advise of problems ahead in plenty of time to take action. Superbatch was developed at the Center for Process Systems Engineering at Imperial College.

Not many tools nor methods exist that allow verification of the recipe before it is used for control. Hazardous processes exist and motivate the need for such methods. The idea of the approach of Brettschneider, Genrich and Hanisch, [Brettschmeider *et al.*, 1996], is to model the recipe, the plant and the device control (equipment control) by means of high-level Petri nets. By merging the nets, verification and performance analysis can be performed using the analysis methods of Petri nets.

7.2 Grafchart for Batch Control

The physical and procedural models of ISA-S88, as well as the different recipe types, can all be nicely represented in Grafchart and the G2 environment.

Physical Model

The physical model of ISA-S88 defines the hierarchical relationships between the physical units involved in batch control. Only the lower four levels will be considered as shown in Figure 7.1 (left).

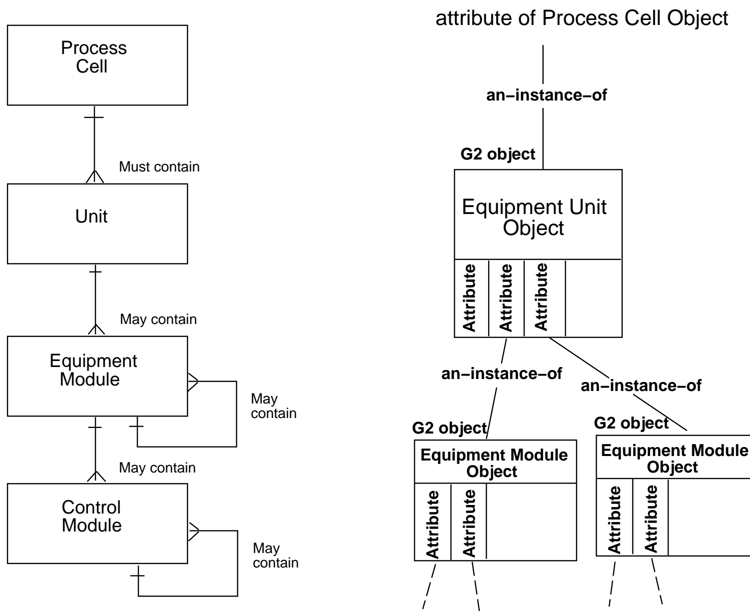


Figure 7.1 ISA-S88 Physical Model (left) and its G2 representation (right).

The hierarchical structure of the physical model is straightforward to implement in an object-oriented environment such as G2. The attributes of a G2 object can either be values (numbers, strings, or symbols) or other objects. The latter case gives a hierarchical object structure that well matches the physical model. A G2 object representing an equipment unit may have attributes that contain other G2 objects representing the equipment modules in the unit. The equipment modules may have attributes containing the control modules and other equipment modules within the module. The equipment unit object may itself be a part of an object representing the batch cell. The situation is shown in Figure 7.1 (right).

Procedural Model

The procedural model of ISA-S88 is also hierarchical, see Chapter 6.3 (Procedural Control). The structure of the procedural model is shown in Figure 7.2 (left).

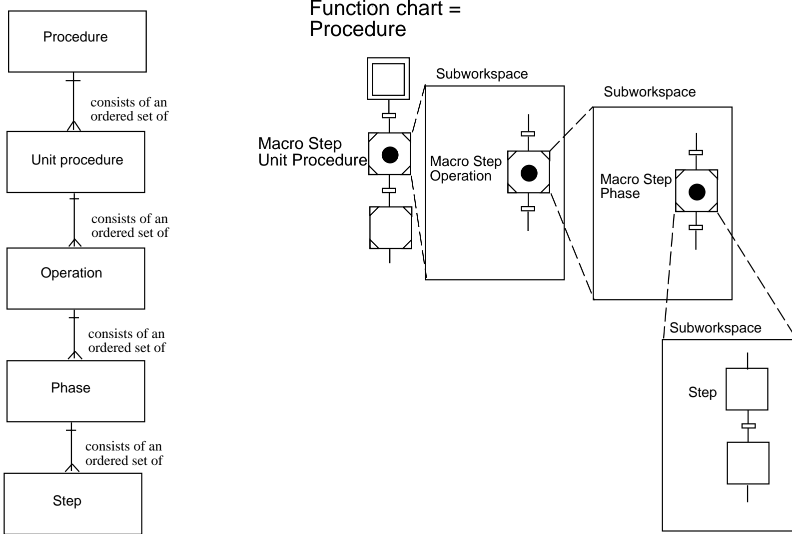


Figure 7.2 ISA-S88 Procedural Model (left) and its representation in Grafchart (right).

The hierarchical levels of the procedural model can conveniently be described with the hierarchical abstraction facilities of Grafchart. A procedure can be represented as a function chart composed of macro steps representing the unit procedures. These macro steps may contain other macro steps or procedure steps representing the operations. Similarly, the macro steps representing the operations contain other macro steps representing the phases. Finally, the phase macro steps contain ordinary steps with associated step actions. The situation in shown in Figure 7.2 (right).

Recipes

ISA-S88 defines four recipe types: general recipes, site recipes, master recipes and control recipes. Here, only the master and control recipes are considered. A recipe contains administrative information, formula information, requirements on the equipment needed to produce the recipe and the procedure that defines how the recipe should be produced. The procedure is organized according to the procedural control model. The master recipe is targeted to a process cell. Each individual batch is represented by its control recipe. The control recipe is originally a copy of the master recipe which has been completed and/or modified with scheduling, operational and equipment information. A control recipe can be viewed as an instantiation of a master recipe.

With Grafchart, recipes can be represented using Grafchart-BV or using Grafchart-HLV. Using Grafchart-BV, a recipe is represented as a function chart with parameters (attributes). The procedure part of the recipe is represented by the function chart and the formula information and equipment requirements are represented by parameters (attributes) of the Grafchart process encapsulating the function chart. This information can be accessed from the recipe procedure using the `sup.attribute` notation. This way of representing a recipe is shown in Figure 7.3.

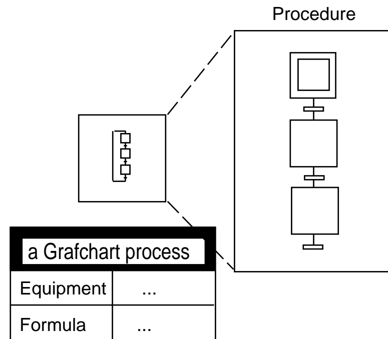


Figure 7.3 Control recipe as a function chart.

Using Grafchart-HLV, a batch is represented as an object token. The object token contains the formula information and equipment requirements as attributes. The recipe procedure can be represented in two main ways. In the first case it is represented by the function chart in which the token resides, see Figure 7.4 (left). In this case, the formula information and equipment requirements can be accessed using the `inv.attribute` notation or the `temporary-tokenname` notation. The recipe procedure can also be represented by a Grafchart method belonging to the object token itself, see Figure 7.4 (right). The recipe procedure method can then access the formula information and equipment requirements using the `self.attribute` notation.

Recipe and Equipment Control Separation

The control recipe does not contain enough information to operate the process cell. At some level it must be linked to the equipment control that is responsible for the actual operation of the process equipment. The separation between the control recipe procedure and the equipment control is illustrated in Figure 6.3. The dashed levels could either be contained in the control recipe or in the equipment control. The linking could either be done on the procedure level, unit procedure level, operation level, or phase level.

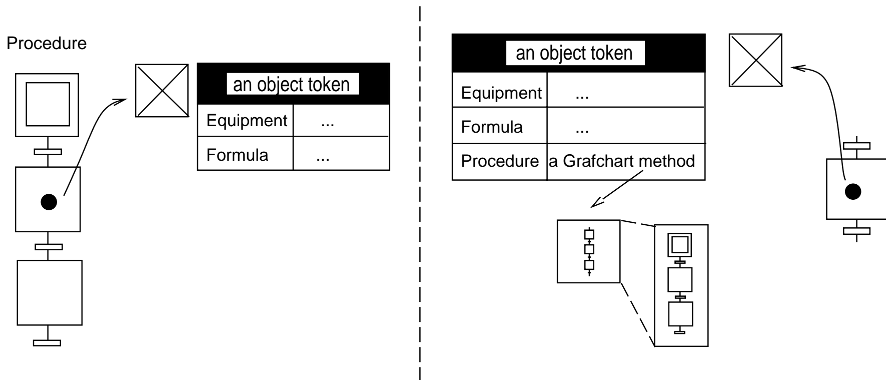


Figure 7.4 Control recipe as an object token without method (left) or with method (right).

Using Grafchart the linking is implemented using methods and message passing according to Figure 7.5. The element in the control recipe where the linking should take place is represented by a procedure step. Depending on at which level the linking takes place, this procedure step could represent a recipe procedure, recipe unit procedure, recipe operation or recipe phase. This procedure step calls the corresponding equipment control element which is stored as a Grafchart method in the corresponding equipment object. If the linking takes place at the unit procedure level then the equipment control element corresponds to an equipment unit procedure. If the linking is done at the operation level the equipment control element is an equipment operation, etc.

7.3 A Batch Scenario Implemented in G2

A batch scenario consisting of one batch cell has been defined and implemented in G2. The batch scenario is used as a test platform to investigate how Grafchart can be used for recipe handling.

In the batch cell, different products can be made. The batch cell is structured as a network, which means that for each batch that will be produced there are several possible ways through the plant. It is possible to have several batches in the plant at the same time, the batches may be of the same or of different types. The batch cell is thus of the multi-product, network-structure type, see Chapter 6.

The cell consists of three raw material storage tanks, two mixers, three buffers, two batch reactors, and three product storage tanks. The units are

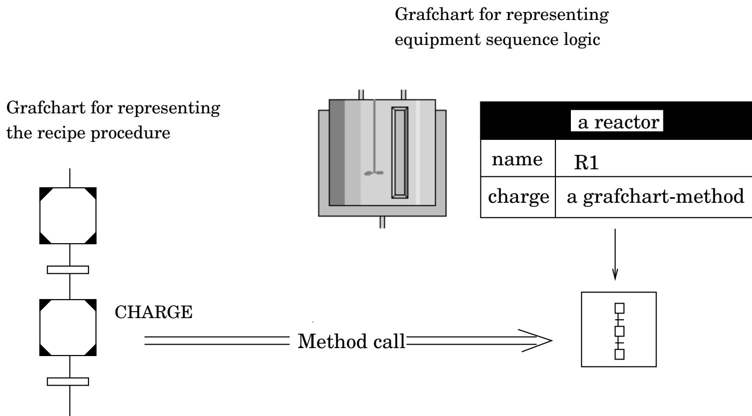


Figure 7.5 Control recipe/Equipment Control linking.

interconnected through valve batteries. The cell can produce two products named D and E using three reactants A, B and C. A schematic of the batch cell is shown in Figure 7.6.

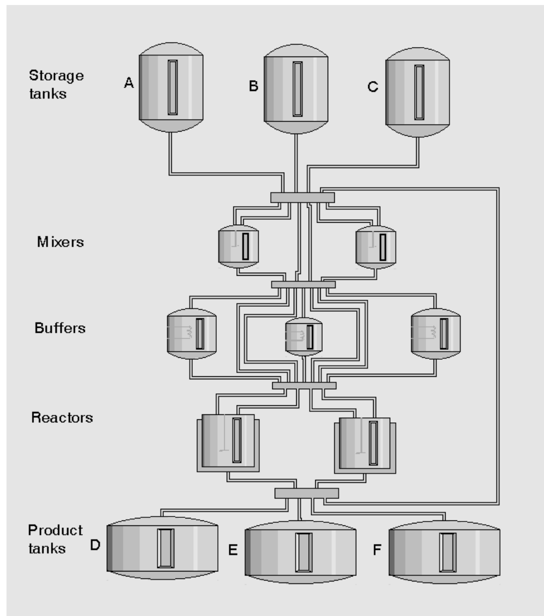


Figure 7.6 The batch cell.

The scenario is divided into three major parts: a simulator that simulates the dynamics of the scenario, a process control system and an operator interface. Figure 7.7 shows the three parts of the batch cell.

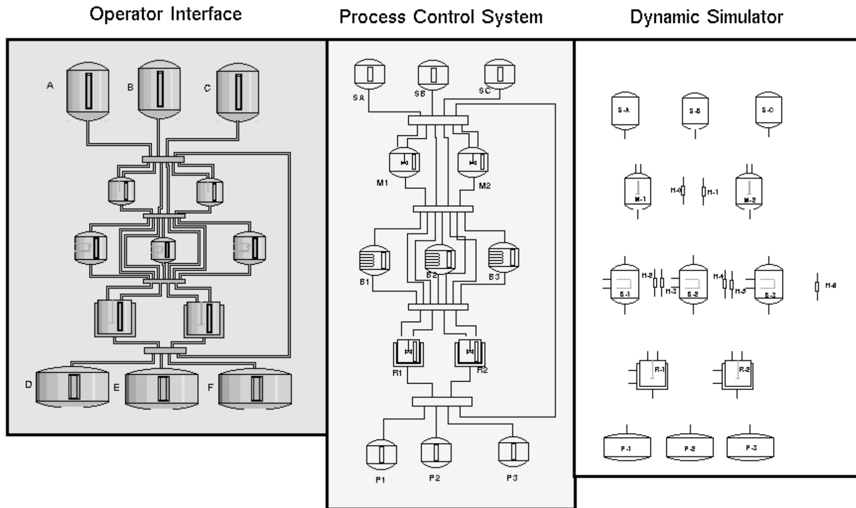


Figure 7.7 The three parts of the batch scenario.

Unit Processes

The batch cell is composed of several unit processes.

Storage tanks The storage tanks contain the raw materials, i.e the reactants. Since there are three different reactants there are three storage tanks in the cell, one for each reactant. Each storage tank has a level-transmitter and a temperature-transmitter. Each storage tank also has an output on-off valve and pump connected to the outlet of the tank. The storage tanks are assumed never to be empty. The schematic of a storage tank is shown in Figure 7.8 (left).

Mixers The main task of a mixer is to mix different substances, but a mixer can also be used to store substances while waiting for some other unit. The mixer is equipped with an agitator. The mixer has two inlets, each controlled by an on-off valve. In this way it is possible to fill the mixer in parallel. The outflow of a mixer is controlled by a pump and an on-off valve. The mixer has three transmitters: a level-transmitter, a temperature-transmitter and a pressure-transmitter. In Figure 7.8 (middle) a mixer is shown.

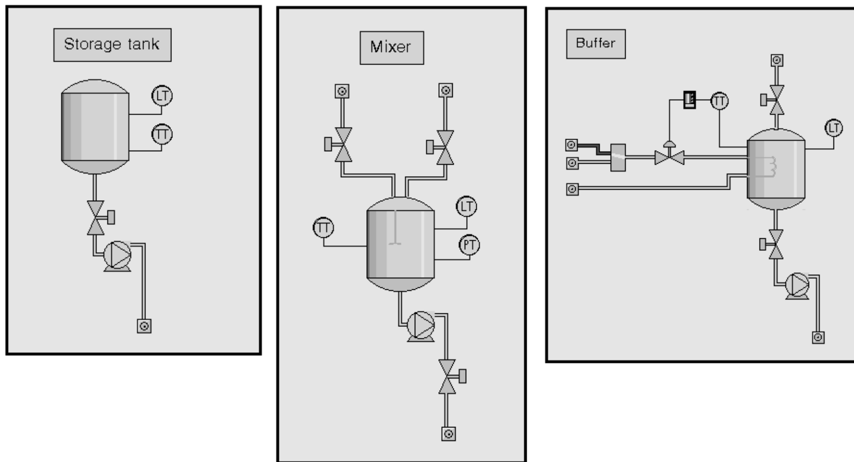


Figure 7.8 A storage tank (left). A mixer (middle). A buffer (right).

Buffers The task of the buffer is to store substances and if necessary maintain the temperature of the substance. A buffer can also be used to heat or cool a substance. The inflow of the buffer is controlled by an on-off valve and the outflow is controlled by a pump and an on-off valve. As shown in Figure 7.8 (right) a buffer has a level and a temperature transmitter. The temperature transmitter is connected to a PID-controller. The PID-controller adjust the flow of hot or cold liquid to the heater of the buffer and, thus, controls the temperature of the content of the buffer.

Reactors A reactor has two inlets and one outlet. It has an agitator and four transmitters: level, temperature, pressure and flow. The outflow is, as before, controlled by a pump and an on-off-valve. The inflow, and thereby the level in the reactor, is controlled by control valves using a PID-controller. A schematic of the reactor is given in Figure 7.9 (left).

The main task of the reactor is to perform chemical reactions. This is done by heating or cooling the substance inside the reactor. The reactor has a jacket which can be filled with a hot or cold liquid. The temperature in the reactor is controlled by two cascade coupled PID-controllers that adjust the flow through the jacket.

Product tanks The product tanks are used for storing the finished products. They are assumed never to be full. Each product tank has an inlet valve and a level transmitter. A schematic of a product tank is shown in Figure 7.9 (right).

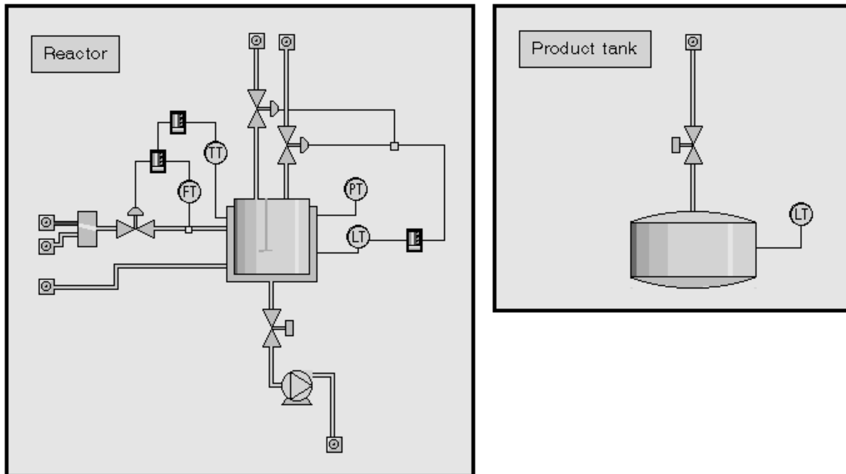


Figure 7.9 A reactor (left). A product tank (right).

Valve-batteries Valve-batteries, see Figure 7.10, are placed in between the different units. A valve battery is organized in a matrix structure with `n` rows and `m` columns where `n` is the number of inlets and `m` is the number of outlets. By opening the valve placed in position `(i,j)`, inlet `i` will be connected to outlet `j`.

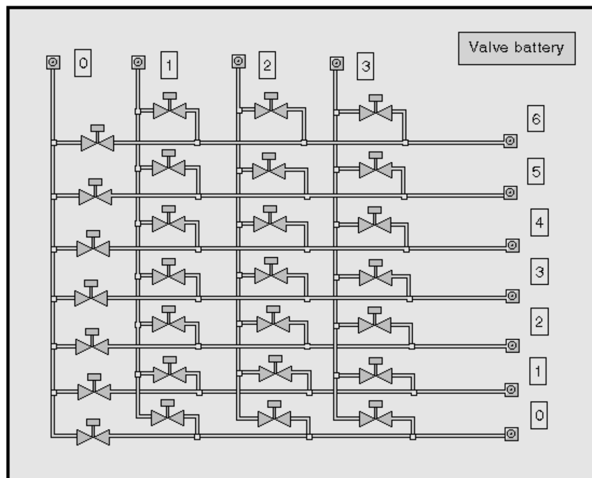


Figure 7.10 A valve-battery.

Operator Interface

The operator interface, see Figure 7.6, provides the operator with a user friendly interface, through which he or she can get information about the status of the plant. Through the interface the operator can also interact with the plant.

By clicking on any of the units the internal structure of this unit is shown. The operator can open and close the valves manually by clicking on them. The pumps can be turned on and off manually. Some of the units also have PID-controllers, these can be operated manually or automatically. When clicking on any of the sensors a trend-curve of the measured signal is shown, see Figure 7.11.

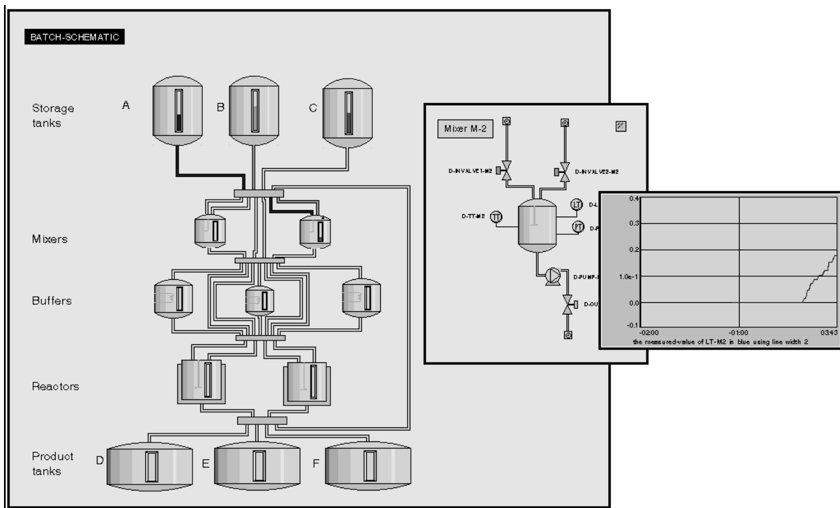


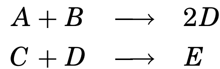
Figure 7.11 The operator interface showing the internal structure of a mixer and a trend curve.

As the batch moves through the plant, its actual position is highlighted. A unit currently used by a batch is marked with a black dot placed at the upper right corner. The level indicator of this unit shows the actual level in the unit. If the batch is using a pipe, i.e. is on its way from one unit to an other, this pipe is marked with a colour.

Dynamic Simulator

The dynamic simulator is used as a substitute for a real plant. It simulates the mass balances, energy balances and the chemical reactions. All simulations are done in real-time.

Two reactions can be simulated in the scenario:



$$
\begin{aligned}
A + B &\longrightarrow 2D \\
C + D &\longrightarrow E
\end{aligned}
$$

The total mass balance, the component mass balances and the energy balance for the reactions are calculated and simulated. A detailed description of the dynamic simulator is found in Appendix D.

Process Control System

The process control system implements the basic control functions. It both receives information from and sends information to the simulator. For example, if the operator starts a pump this information must be propagated to the simulator and the pump should be turned on. Similarly, information about, e.g., the actual level in a unit should be sent from the simulator to the process control system.

The process control system contains equipment objects (unit processes) representing the units in the plant, i.e., the tanks, mixers, buffers and reactors. The equipment objects have an internal structure of sensors, actuators and PID-controllers. These correspond to the equipment and control modules of ISA-S88. These objects are stored as attributes of the unit process objects, see Figure 7.12.

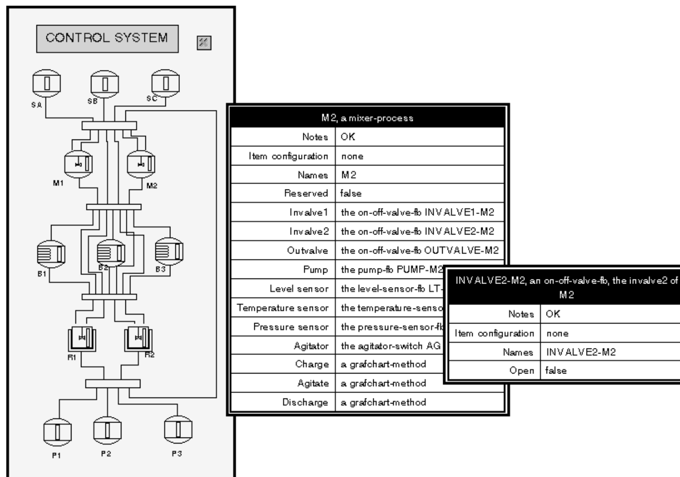


Figure 7.12 The process control system with its equipment objects and their internal structure.

To each unit process a number of methods are associated. These methods are descriptions of the operations or phases that the unit can perform, i.e., the methods are detailed descriptions of how to perform a certain task on one particular unit. The names of the methods are found in the attribute table of each object whereas the implementation of the different methods are found on the subworkspace of the class definition it belongs to. A mixer can, e.g., perform the following tasks: charge, agitate and discharge as shown in Figure 7.13. The methods are implemented as Grafchart methods.

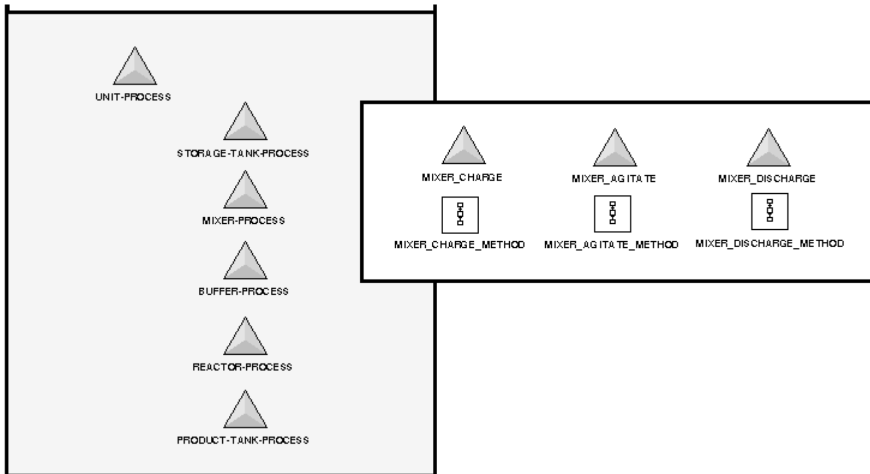
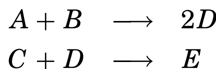


Figure 7.13 The methods of the class 'mixer-process'.

Recipes

Two reactions can be simulated in the scenario:



$$\begin{aligned} A + B &\longrightarrow 2D \\ C + D &\longrightarrow E \end{aligned}$$

Recipe $A + B \longrightarrow 2D$: First the same amount of the two reactants are filled into a mixer. The filling can be done either in parallel or in sequence. When the filling is completed the agitation is started. When the two reactants are well mixed the content of the mixer is transfered into a reactor. In the reactor the reaction takes place. This reaction needs to be heated for the reaction to start. Finally the reactor is emptied and the substance is stored in a product-tank.

Recipe $C + D \longrightarrow E$: The production of substance E can be done in either of two ways. Both start with producing substance D as described above.

- Substance D is transported from the reactor to a buffer and then reactant C is added. The content of the buffer is emptied into a reactor and the reaction is started by heating the content. When the reaction is completed the reactor is emptied and the product is stored in a product-tank.
- Reactant C is filled into a buffer and heated to the same temperature as substance D at the same time as product D is being produced. Then the content of the buffer is transfered to the reactor and the two substances are agitated, and, if needed, further heated. When the reaction is completed, the product, i.e., substance E, is stored in a product-tank.

7.4 Summary

An object oriented language is well suited to represent the hierarchical structure of the physical model. Grafchart can be used to represent all the levels in the hierarchical procedural control model and, by using the support for methods and message passing, the linking between the control recipe procedural elements and the equipment procedural elements can be nicely represented.

8

Recipe Structuring using Grafchart

The main application of Grafchart is recipe structuring and resource management in multi-product, network-structured batch production cells. A number of different ways of how to structure the recipes and how to incorporate resource allocation have been investigated.

Grafchart is used at two levels, to represent the sequential structure in the recipes, i.e., the step by step instructions of how to go from raw material to product, and to represent the sequential control logic contained in the equipment units. To link the two parts together, method calls are used, as discussed in Chapter 7.2 (Recipe and Equipment Control Separation).

In this chapter, four alternative representations of recipe structures are presented. The structuring alternatives have large differences and they therefore have different advantages and disadvantages. Each major structuring alternative is presented in a separate section. Drawbacks and advantages are presented for each alternative. The first structuring alternatives, Chapter 8.1, use the basic version of Grafchart whereas the reminding alternatives, Chapter 8.2, 8.3, and 8.4, use the high-level version of Grafchart.

In all structuring alternatives, a batch can be followed through the plant and it is possible to record its history in logs. This has, however, not yet been fully implemented and is therefore not described.

As a common example, the simple recipe for producing product `D`, see Chapter 7.3, will be used. The production is assumed to be performed in the process cell described in Chapter 7.3.

8.1 Control Recipes as Grafchart Function Charts

The alternatives given in this section use the basic version of Grafchart.

In this alternative, a master recipe is represented by a Grafchart process. The Grafchart process has attributes specifying the type of equipment needed when producing the batch. The control recipe is a copy of the master recipe where the attributes are given their actual values, i.e., the equipment that will be used is specified. The situation is shown in Figure 8.1 .

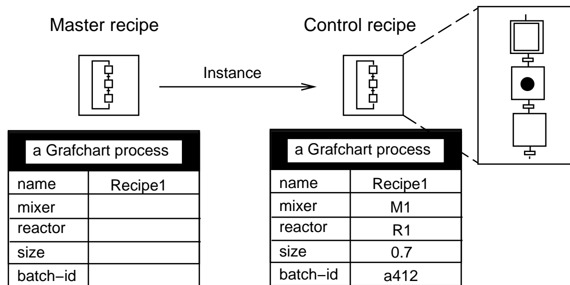


Figure 8.1 Master and control recipes.

From the steps and transitions within the control recipe the attributes can be referenced using the sup notation, see Chapter 5.3.

Recipes Based on Recipe Phases

In the first example the control recipe procedure consists solely of phases. Each phase references the associated equipment phase through a method call according to Figure 8.2.

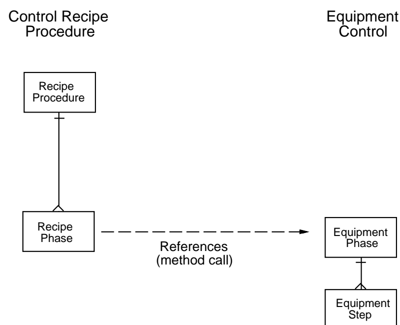


Figure 8.2 Recipes based on recipe phases.

The resulting control recipe is shown in Figure 8.3. The different unit processes that the batch passes through are indicated by the column-based organization of the recipe. The recipe starts by an assignment step in which the resources necessary to produce the recipe are allocated. Two versions of the recipe have been implemented. One where the operator manually assigns all the equipment before starting the procedure and one where the equipment is assigned automatically through a simple search algorithm immediately before the equipment is needed. After the assignment step the two reactants A and B are filled into a mixer. This is done in parallel, since the mixers have two inlets. Then the agitation is started. When the agitation is finished the mixer is emptied and at the same time

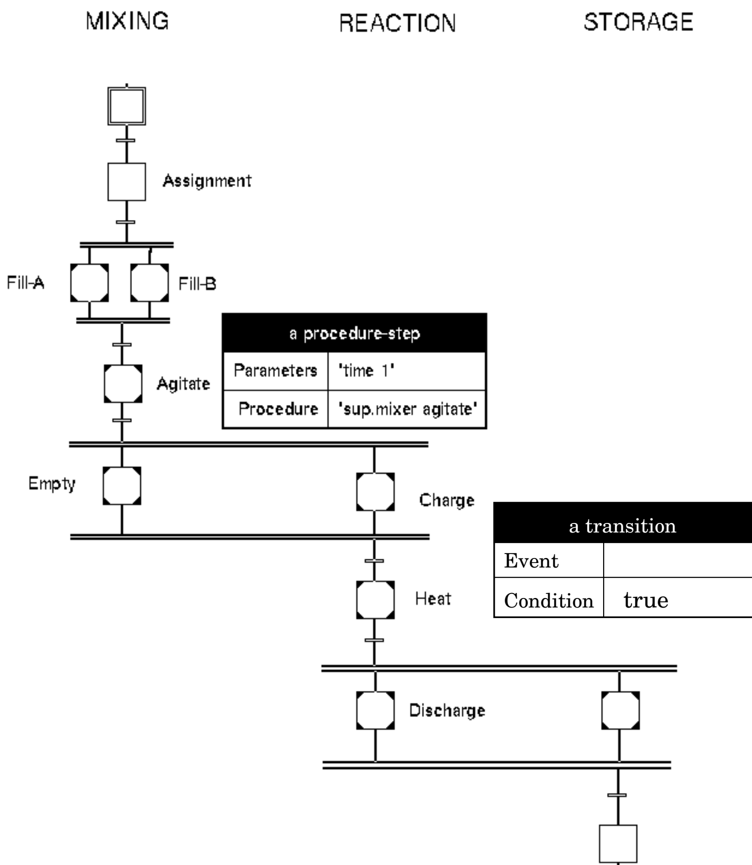


Figure 8.3 Control recipe based on recipe phases.

the reactor is charged. This is expressed with the Grafchart parallel construct. When the charging is finished, the reaction is started by heating the batch. Finally, the content of the reactor is transfered to the product tank and the production of the batch is completed.

Each recipe phase is represented by a procedure step that calls the corresponding equipment phase. In Figure 8.3 the attribute table for the agitate step is shown. The procedure step calls the agitate methods in the mixer that has been assigned to the batch. This mixer is contained in the `mixer` attribute of the recipe. The parameter `time` is given the value `1`. This denotes how long the agitation will take. The agitate equipment phase consists of two steps where the agitator motor is active until the defined time has elapsed. The situation is shown in Figure 8.4.

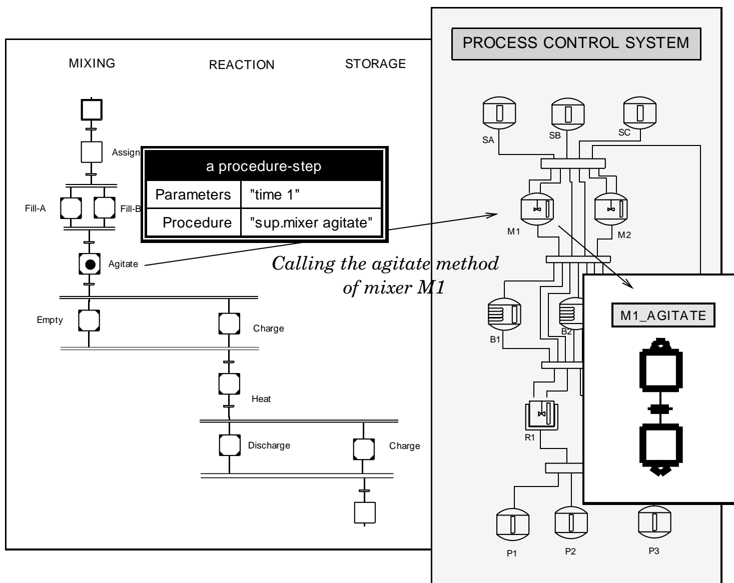


Figure 8.4 Recipe - Equipment linking.

Between the procedure steps in a recipe, transitions are placed. The equipment phases called by the procedure steps are most often written so that they themselves take care of the initialization and the termination of the phase. This means that the receptivity of a transition placed between two procedure steps becomes very simple. In Figure 8.3, the transition receptivity of the transition following the heat step is shown. As can be seen this receptivity only contains an “always-true” condition.

Recipes Based on Recipe Operations

In the second recipe structuring alternative the control recipe is structured into operations that internally are decomposed into recipe phases. The linking between the control recipe and the equipment control still takes place at the phase level. The structure is shown in Figure 8.5.

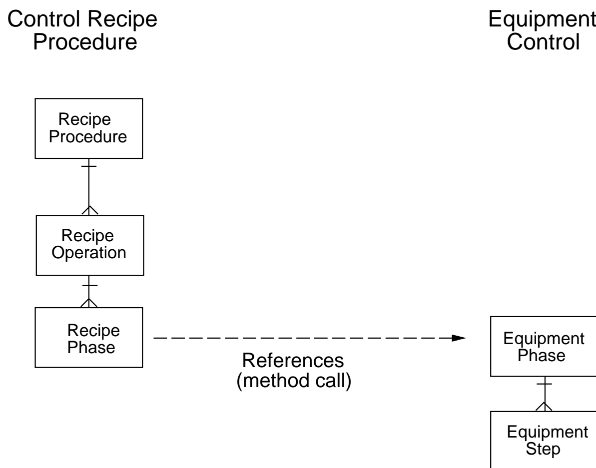


Figure 8.5 Recipes based on recipe operations.

The resulting control recipe is shown in Figure 8.6. The control recipe is a straight sequence of recipe operations. Each recipe operation is represented as a macro step that contains the recipe phases. The phases are represented as procedure steps that call the corresponding equipment phases.

A problem with this approach is how to handle the parallelism between the operations. The emptying of the mixer and the charging of the reactor should be performed in parallel. Here this is solved using process steps. The `Empty` phase of the mixer operation is started through a process step, i.e., as a separate execution thread. The result will be that the `Empty` phase will execute at the same time as the `Charge` phase of the reaction operation. A drawback with this approach is that the parallelism is not expressed explicitly with the Grafchart constructs and therefore not directly visible to the operator as in the previous example. The recipe structure in Figure 8.6 could also be used if the linking was done at the operation level. In that case the equipment operations are internally structured as equipment phases.

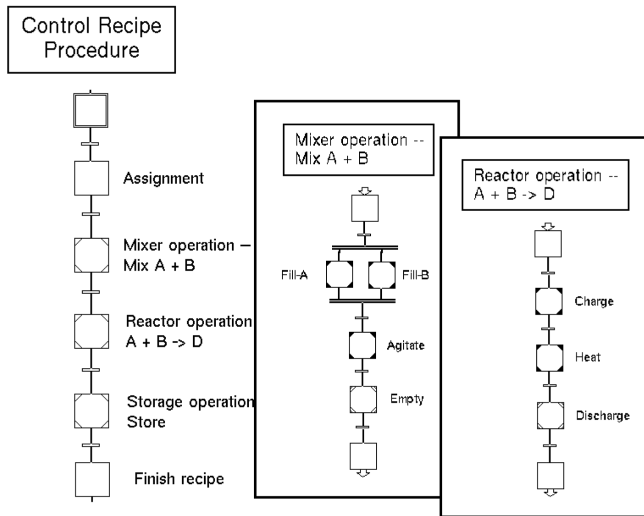


Figure 8.6 Control recipe based on recipe operations.

Recipes Based on Equipment Operations

In the third example the control recipe and the equipment control are linked at the operation level according to Figure 8.7.

In the previous examples, the control recipe has explicit information about which type of unit processes that the batch should be processed in, e.g., a mixer, buffer or reactor. Another possibility is to let the control recipe be completely independent from which unit type it should use. This can be achieved by defining a number of generic equipment independent operations. Examples of generic operations could be store, store-with-agitation, mix, mix-with-agitation, store-under-heating, exothermal-reaction etc. It is possible to mix two reactants in both mixers, buffers and reactors. However, it is preferred to mix reactants in a mixer. The mix-with-agitation operation cannot be performed in a buffer. The store-under-heating operation can be performed in both buffers and reactors but it is preferred to use the buffers for this. Hence, for each generic operation a preference ordering must be given. It is the task of the resource allocation scheme to find the best suited unit processes for each recipe operation and to assign this to the recipe when it is needed. The equipment operations are represented as equipment object methods. These methods call the equipment phases which also are methods of the equipment objects. This structure has not been implemented and tested in the batch scenario.

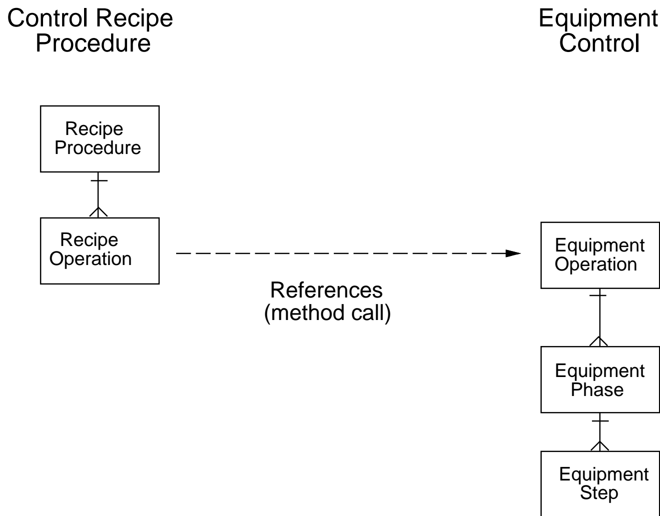


Figure 8.7 Recipes based on equipment operations.

Advantages and Disadvantages

Structuring a recipe management system with control recipes represented by Grafchart function charts, has advantages and drawbacks. The main advantage is that the system is easy to comprehend, one batch equals one Grafchart process. Each control recipe, i.e., each Grafchart process, can be modified independently of the other control recipes simply by changing the parameters of the procedure steps or by changing the function chart. However, the drawback is that it is not easy to get simultaneous information about all batches being produced in the cell.

8.2 Control Recipes as Object Tokens

The recipe structuring alternatives presented in this section use the high-level version of Grafchart.

In the previous section, one copy of the master recipe was needed for each batch that was produced. The reason for this is not that the batches are produced in different ways or that they use different types of equipment, but that they do not necessarily use the same equipment or have the same batch-identity number. Since the attributes, in the last section, were global in the context of the function chart, the same control recipe cannot be used simultaneously by more than one batch. If, however, the

batch specific information is stored locally, e.g., in the tokens, the same function chart can be used by more than one batch. Information that the batches share is represented globally, whereas batch specific information is represented locally. Since the token contains the batch specific information, the token represents the control recipe and the function chart that the token resides in is the common master recipe. This way of structuring a recipe is shown in Figure 8.8.

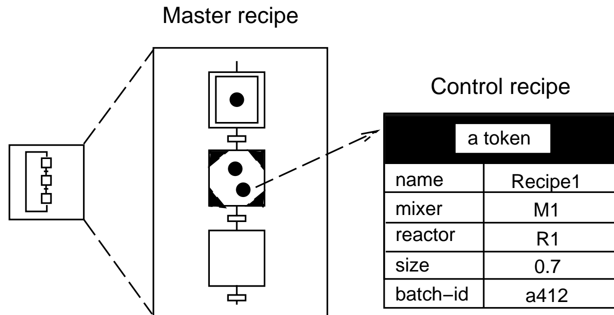


Figure 8.8 Control recipes as tokens.

The recipe alternatives presented in this section are all based on recipe phases and each phase calls the associated equipment phase through a method call. It is also possible to build up the recipe using operations instead of phases, in the same way as described in Chapter 8.1 (Recipes Based on Recipe Operations).

The recipe shown in Figure 8.3 can be structured with object tokens. The attributes previously contained in the control recipe function chart will now be contained in the tokens. It is possible to have several tokens in the same master recipe. Each token corresponds to one control recipe, i.e., one batch.

Recipes Extended with Explicit Resource Allocation

Using the object token structuring possibility it is possible to combine resource allocation with recipe execution in a Petri net fashion, see Figure 8.9.

In the recipe two batches are under production. One batch is currently filling reactant `A` and reactant `B` in one of the mixers and the other is heating the content in one of the reactors. At the moment there is thus only one mixer (out of two) and one reactor (out of two) reserved by a batch.

Figure 8.9 Master recipe with resource allocation.

In this approach an available equipment unit is represented as a token residing in a step. The token corresponds to a semaphore that guarantees a batch exclusive access to the resource (equipment unit). PN-transitions with multiple input and output arcs are used to express the resource synchronization. If a batch has specific resource requirements, these can be expressed and checked against the equipment capacity in the transition receptivity.

The chart can also be extended with more equipment oriented operations such as cleaning (CIP - clean in place). In Figure 8.10 an extremely simple master recipe is shown. Here, a CIP-operation must be performed on each unit after it has been used by a batch.

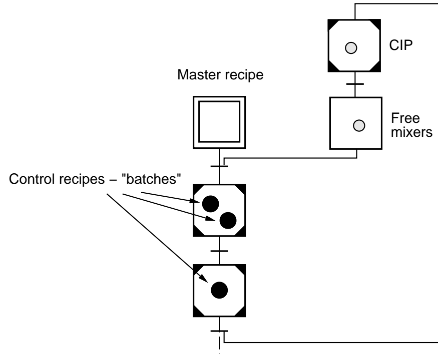


Figure 8.10 A master recipe with combined equipment oriented operations.

Combined Network

Using the object token structuring possibility combined with resource allocation it is possible to merge all different master recipes into one single Grafchart. This chart visualizes the resources sharing that exists between the batches. The chart can be analyzed with Petri nets analysis methods to detect if there is a risk for deadlock. The case for two simple master recipes is shown in Figure 8.11.

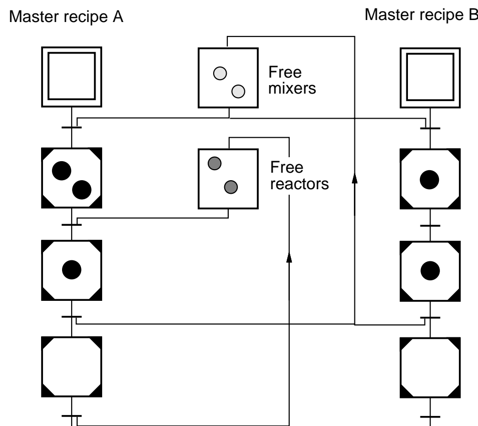


Figure 8.11 Combined net with all master recipes.

The combined network of the two recipes $A + B \rightarrow 2D$ and $C + D \rightarrow E$, see Chapter 7.3, is shown in Figure 8.12. The recipe $A + B \rightarrow 2D$ is represented to the left in the figure and the more complex recipe, $C + D \rightarrow E$ is represented to the left. The resources needed are shown in the center of the figure. The tokens representing mixers reside in the upper step, the tokens representing the buffers in the mid-step and the tokens representing the reactors in the bottom-step. The buffers are only used by the recipe $C + D \rightarrow E$. This net is large and complex and therefore not suitable for representation.

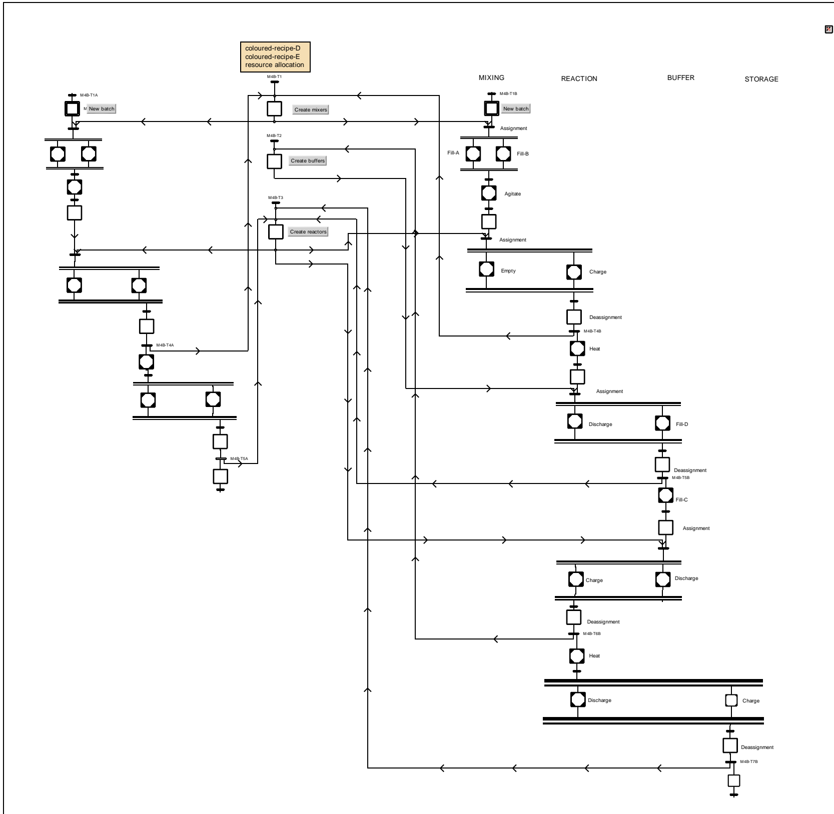


Figure 8.12 Combined net with all master recipes.

A combined net can however be represented in a more attractive way by using connection posts. Each master recipe function chart is represented separately. The different function charts are connected through connection posts, see Figure 8.13.

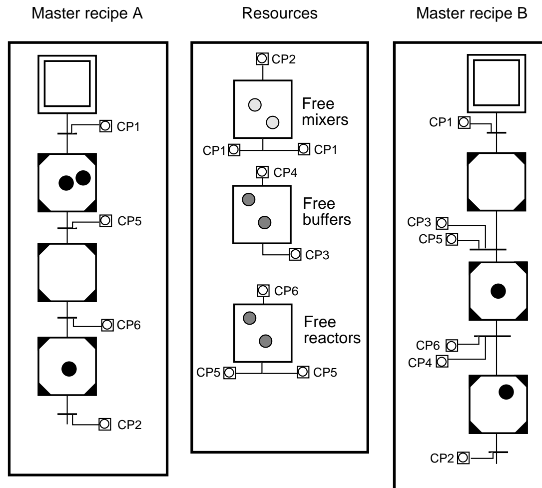


Figure 8.13 Combined net with all master recipes using connection posts.

Advantages and Disadvantages

The structuring alternative presented in this section gives a more compact representation compared with the structuring alternative given in Chapter 8.1. It makes it easy to see how many batches of a certain type that are under production. The recipes have been structured using phases. It can however also be structured using operations, in the same way as in Chapter 8.1. If the master recipe function chart is changed this will affect all control recipes. This can be both an advantage and a disadvantage. A drawback is that it is not possible to modify the procedure of one control recipe alone. This can however be resolved by using one Grafchart function chart for each batch, according to Chapter 8.1, together with connection posts. This situation is shown in Figure 8.14.

8.3 Multi-dimensional Recipes

The multi-dimensional feature of Grafchart fits nicely into the batch control problem. In the first recipe structuring alternative, Chapter 8.1, only one token, representing one batch, could reside in one chart. In the second recipe structuring alternative, Chapter 8.2, several tokens, each representing one batch, could reside in a chart, provided they were all of the same type. The tokens shared the structure of the master recipe but contained specific information about the equipment to use when being

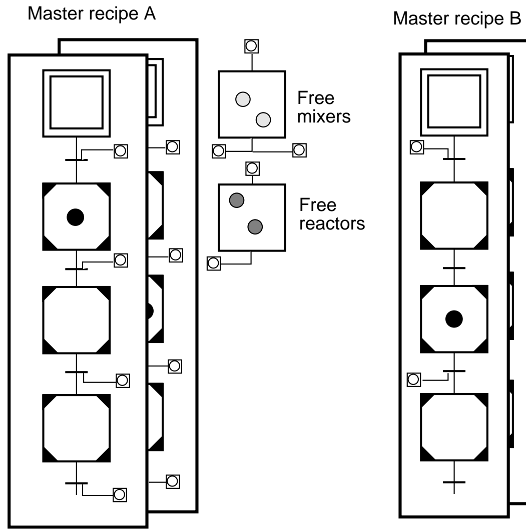


Figure 8.14 Combined net with all master recipes using connection posts.

produced. Another, more sophisticated structuring alternative is to let the token contain, not only information about the equipment but also a method describing how it is produced. The chart in which the tokens reside now becomes more general and allows for tokens of different types to reside in the same chart. The idea is shown in Figure 8.15

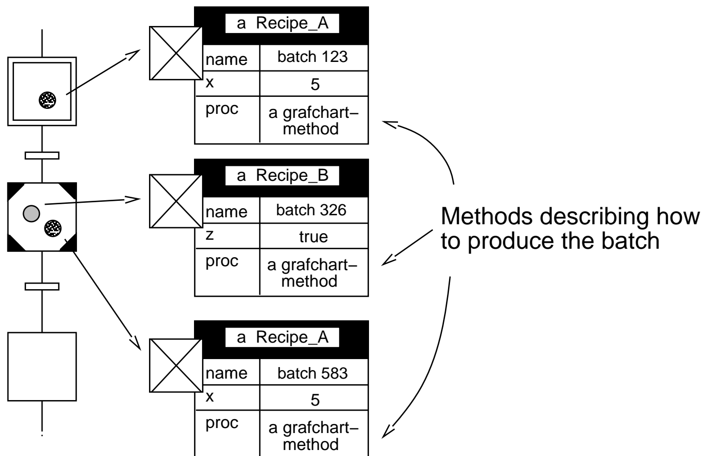


Figure 8.15 A multi-dimensional chart.

Multi-dimensional Recipes with Implicit Resource Allocation

The first multi-dimensional recipe is shown in Figure 8.16. The function chart of the base-level of the multi-dimensional recipe is very general and applies to all batches, independently of their type. In Figure 8.16, this chart consists of one initial step, two steps and one procedure step. The tokens in the initial step can represent the orders effected to the plant. The step following the initial step contains initializational tasks that have to be performed before the production can start. The batches currently under production reside in the procedure step and the finished batches in the last step.

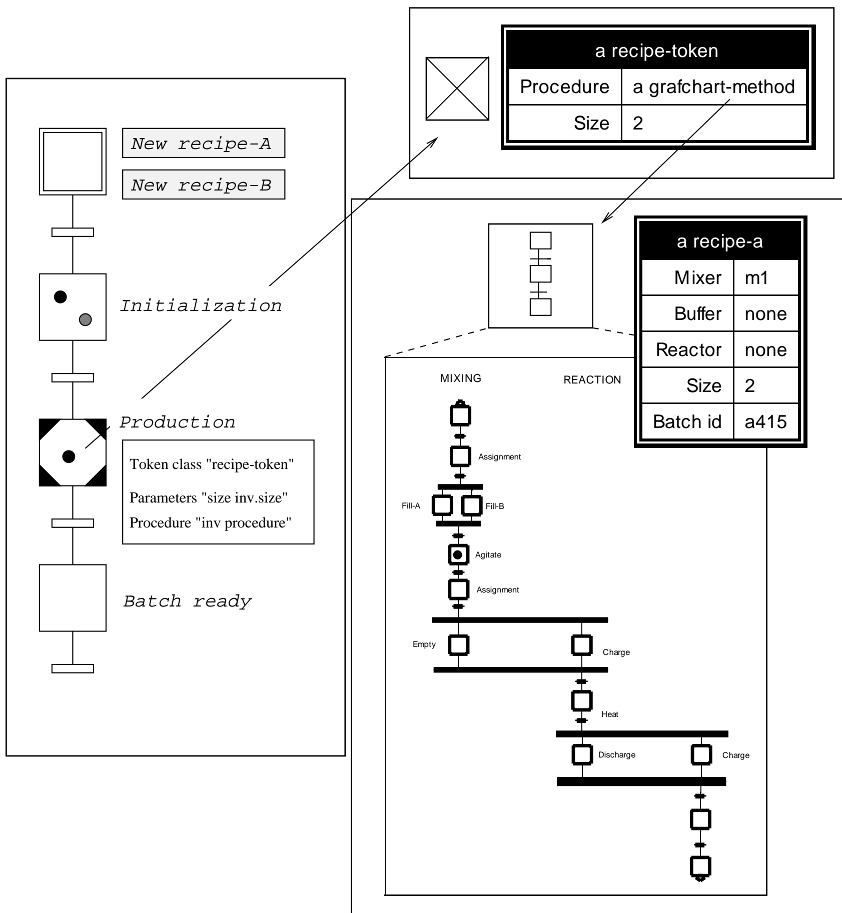


Figure 8.16 Multi-dimensional chart with implicit resource allocation.

When a token enters the procedure step a call is performed and a method of the token is started. This method describes how this specific batch should be produced and constitutes a part of the second level in the multi-dimensional recipe. The method is implemented as a Grafchart procedure, see Chapter 5.5. The attributes of the Grafchart procedure contain the equipment specification of the batch. The different levels of the multi-dimensional chart communicate through the different dot notations, see Chapter 5.5 (Multi-dimensional Charts).

Each control recipe, i.e., each token, is represented by an object token and visualized by a grafchart marker. When a new batch is to be produced a new control recipe object token is created and a grafchart marker of the correct type is placed in the initial-step of the function chart. A master recipe is represented as an object token with attributes and a method. A control recipe object token is created by copying the master recipe object token.

The resource allocation of this recipe structuring alternative is done implicitly. Immediately before an equipment is needed a simple search algorithm is executed and the correct equipment is automatically assigned to the batch.

Multi-dimensional Recipes with Explicit Resource Allocation

In this structuring alternative the resource allocation is explicitly represented. The function chart of the base-level contains an initial-step, a process step, two circular path (one for the batch and one for the equipments) and a final step, see Figure 8.17.

After initialization of a new batch, which is done in the initial step, the production of the batch is started. This is done in the process step, from which the method of the token is called. The method represents the recipe procedure. A multi-dimensional structure of the chart is now achieved. When the call has been performed the token directly moves to the next step. It enters a circular path where it sequentially requests resources and continues its recipe execution in that resource. The token does not leave the circular path until the execution of the batch has reached its end, i.e., until the batch is ready.

The resources (equipment units) in the plant are also represented by tokens. These tokens are included in another circular path, describing the state of the resources. A resource can either be available, occupied by a batch or in cleaning mode (CIP). A batch requiring a resource can only reserve a resource if the resource is available. If the resource required by the batch is not available the batch has to wait until it becomes available.

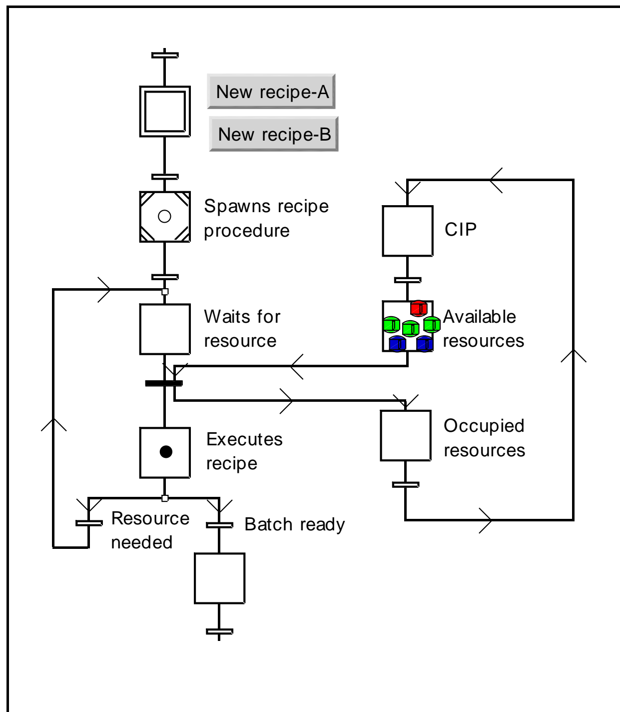


Figure 8.17 Multidimensional chart with explicit resource allocation.

When a batch releases a resource, the resource has to be cleaned (CIP) before it can be utilized again by another batch.

Advantages and Disadvantages

The recipe structures presented in this section are more intricate, and maybe harder to understand at a first glance, than those previously presented. However, the structures have advantages that the previously ones do not have. All types of control recipe tokens are allowed to circulate in the base-level function chart, and as a consequence of this only one chart is needed for a production cell. The base-level function chart gives complete information about: (1) the number of ordered batches, (2) the number of batches in production, (3) the ready produced batches and, (4) the status of the equipment units. If more specific information about a particular batch is required, this can easily be obtained since the information is well structured and placed within the token. This structuring alternative gives a compact graphical overview of the plant status.

8.4 Process Cell Structured Recipes

An alternative to the previous solutions is to represent the process units as steps instead of tokens. This gives a resource allocation chart that resembles the physical structure of the process cell. In Figure 8.18 the chart for a simplified process cell, consisting of one mixer, two buffers and two reactors, is shown.

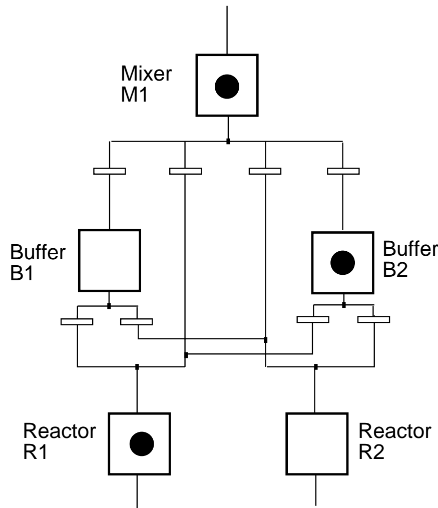


Figure 8.18 Process cell structured recipe.

Each batch is represented as a control recipe token that moves around in the resource allocation chart. The presence of a control recipe token in an equipment step indicates that the recipe is executing an operation in that unit process. Using this approach there are two unclear areas: How should the resource allocation be done and how should parallelism between operations performed in different units be handled.

The resource allocation task can be treated in different ways. In Figure 8.18, the resource allocation mechanism is hidden in the transitions between the equipment steps. In the chart it is assumed that there can be at most one token in each step. In the Petri net field such net structures are refered to as maximum-capacity nets. The net structure obtained is very similar to the Petri net structures used for analyzing batch control cells in [Genrich *et al.*, 1994], [Yamalidou and Kantor, 1991]. The difference is that here the resource allocation is integrated with recipe execution through the multi-dimensional chart nature.

If the maximum-capacity constraint on each process unit should be visualized, each process unit must be represented by two steps. The two steps correspond to the two possible states that a resource can be in: allocated and not-allocated. When a batch needs to allocate a resource, e.g., a reactor, there must be a token in step named "free-reactor". If there is none, the resource can not be allocated. This solution is shown in Figure 8.19. A drawback with this approach is that the resource allocation chart becomes large and less resembles the real plant structure. However, using this approach it is easy to extend the chart with equipment oriented operations, e.g., a CIP-operation.

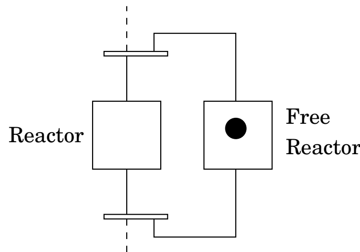


Figure 8.19 A part of a process cell structured recipe.

In order to decrease the size of the net macro steps can be used to encapsulate the extra states. An example of this is shown in Figure 8.20. This representation can, however, be confusing, since the token is placed in the "resource-macrostep" before the resource is allocated.

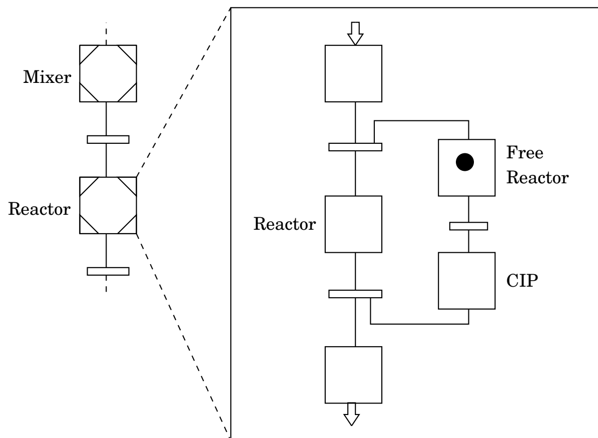


Figure 8.20 Process cell structured recipe with macro steps.

The problem of how to handle parallelism between operations performed in different units can be solved in different ways. Assume a scenario where, e.g., the emptying of a mixer is done in parallel with the charging of a reactor. One trivial solution could be to have one token placed in each of the units that are allocated by a batch, i.e., one token placed in the mixer and one token placed in the reactor. This solution requires that the transition connecting the "mixer-step" with the "reactor-step" has two receptivities. The first receptivity becomes true when the charging of the reactor is started. The firing implies that a new token is created and placed in the "reactor-step". The second receptivity becomes true when the emptying of the mixer is finished. The firing implies that the token placed in the "mixer-step" is deleted. In this way, all units that are used by a batch are clearly marked. However, the user cannot find out how many batches that are under production simply by counting the tokens. It can also be hard to see which of the tokens that refer to the same batch.

Another solution could be the following. When the emptying/charging is started, the token placed in the mixer step is transformed into a real number, e.g., the number 1.0 indicating that all of the content is contained in the mixer. At the same time the number 0.0 is placed in the reactor indicating that nothing of the content is in the reactor. As the mixer is emptied the token number of the mixer is decreased and the token number in the reactor is increased. When the mixer is empty, i.e., the number equals 0.0, and all of the content is in the reactor the number of the mixer is deleted and the number in the reactor, which is equal to 1.0, is transformed in to a black dot again. Such a sequence is shown in Figure 8.21.

In the Petri net field, nets with tokens as real numbers are known as continuous Petri nets, see Chapter 2.6.

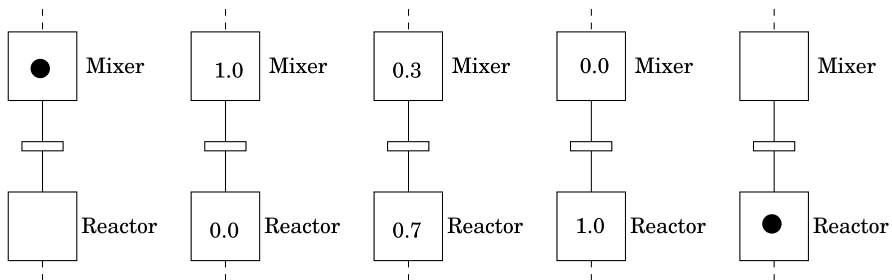


Figure 8.21 Visualization of parallel operations in different units.

The process cell structured recipe easily becomes complex if there are many resources in the plant. This is due to the fact that between every two units there is a transition. In the process plant shown in Figure 7.6 each mixer is connected to three buffers and two reactors. This means that there should be five transitions following each “mixer-step”. In order to improve the situation, a transition-battery can be introduced. The task of the transition-battery can be compared to the one of the valve-battery, see Figure 7.10, i.e., to give a more compact visualization. A resource allocation chart structured with transition batteries is shown in Figure 8.22.

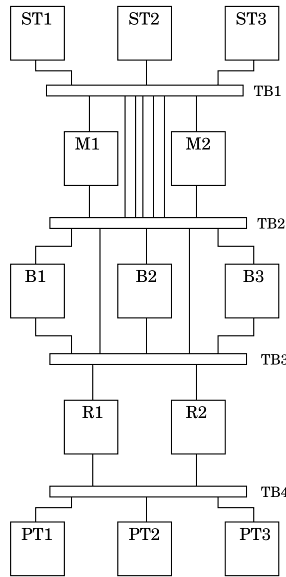


Figure 8.22 Resource allocation chart structured with transition batteries.

In the resource allocation chart there are four transition batteries, TB1, TB2, TB3 and TB4. The four transition batteries are shown in Figure 8.23.

Advantages and Disadvantages

The recipe structure presented in this section focuses more upon the process cell then on the recipe itself. Having a recipe that mimics the configuration can be an advantage since it is very easy to see which units that are currently being used. However, in order for the recipe to mimic the process cell, the Petri net concept of maximum-capacity nets has to be used. Alternatively, a “resource-macrostep” or a “transition-battery” have to be introduced. A difficulty with the approach is how to visualize parallelism between operations performed in different units.

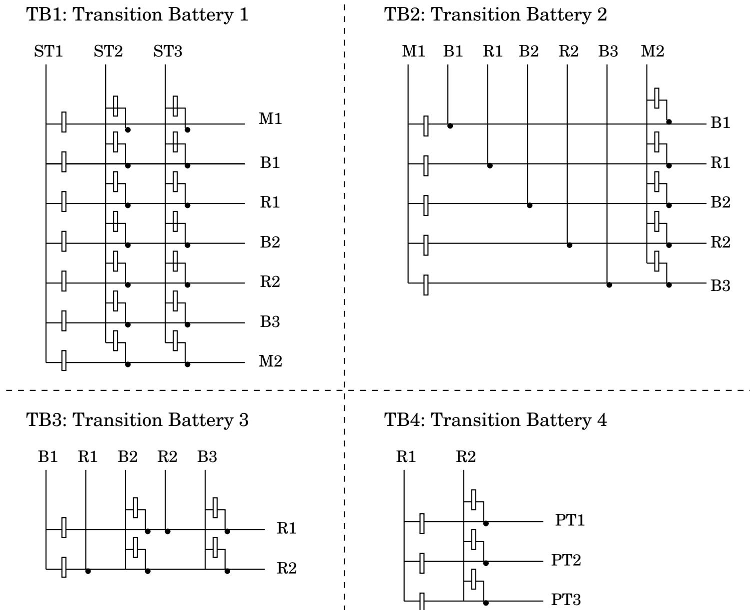


Figure 8.23 Four transition batteries.

8.5 Resource Allocation

Resource allocation is an important part in a recipe management system. If the resource allocation is not done in a correct way batches can get mixed, e.g., if a buffer is allocated to more than one batch their contents will be mixed. This will lead to a costly loss of product. Improper handle of exclusive-use common resources, such as the process units, might also lead to a deadlock situation, i.e., a situation were two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.

The resource allocation problem is treated in two main ways: implicitly and explicitly. In the implicit way, the resources are not represented by any token, instead the allocation is hidden in the actions of a step. This is the case of the structures presented in Chapter 8.1 (Recipes Based on Recipe Phases) (Recipes Based on Recipe Operations), Chapter 8.2 (Recipes with Several Tokens), and Chapter 8.3 (Multi-dimensional Recipes with Implicit Resource Allocation). There are two versions of the implicit resource allocation. The first and most basic version is to let the operator prespecify, before starting the recipe execution, all the equip-

ment that the batch will use. If the specified resource is not available when needed, the recipe execution will pause and wait until it becomes available. In the second version of implicit resource allocation, the resources are automatically allocated. A simple search algorithm looks for an available resource of the correct type, and, if found, allocates it to the recipe.

In explicit resource allocation, resource tokens are used to represent the resources, i.e., the state of the resources in the plant are graphically presented for the user. This is the case of the structures in Chapter 8.2 (Recipes Extended with Resource Allocation) (Combined Network), and Chapter 8.3 (Multi-dimensional Recipes with Explicit Resource Allocation).

Resource allocation of exclusive-use resources and deadlock analysis have been looked at in other areas and their results might be useful also for a recipe management system in the batch control field. This is discussed further in Chapter 9.

Semaphores

In real-time programming the concept of semaphores are used to obtain mutual exclusion, [Dijkstra, 1968]. A semaphore is a flag indicating if a resource is available or not. If a process requests a non-available resource it is placed in a queue. The processes have different priorities and the queue is priority-sorted. When the requested resource is released the first process in the queue can allocate it. For further reading about semaphores see [Burns and Wellings, 1996].

The resource token in the recipe structures with explicit resource allocation can be compared to a semaphore. The fact that different recipes could be given different priorities and that they could be put in a waiting queue is relevant also for batch processes. One way of approaching this would be to introduce special FIFO-steps that maintain a queue of tokens. In Figure 8.24 a small example of a system with mutually exclusive resources is shown.

Shared Resources

The semaphore construct can also be used for shared resources, i.e., resources that can be used by maximum n simultaneous users. A situation like this can be implemented in Grafchart. Each sharable resource is represented by as many tokens as its maximum usage limit, see Figure 8.25.

Sharable resources might also have real-value capacity constraints. This can be implemented in Grafchart, according to Figure 8.26.

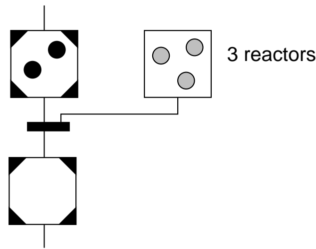


Figure 8.24 A Grafchart structure representing three mutually exclusive resources.

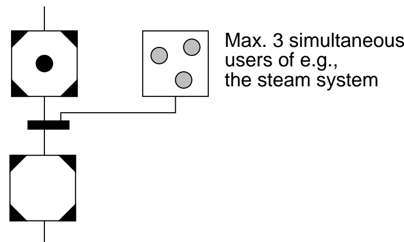


Figure 8.25 A Grafchart structure representing a limited sharable resource.

Situations were the resource has to fulfill certain constraints, e.g., the resource has to have a volume not less than a certain value, can be implemented in Grafchart using the same structure as in Figure 8.26 extended with a receptivity checking the constraint.

Other Approaches

An algorithm commonly used for resource allocation in real-time systems is `Banker's algorithm`, [Silberschatz and Galvin, 1995]. This algorithm is discussed in Chapter 9.7.

A Petri net can formally be analyzed with respect to deadlocks, using the analysis methods described in Appendix B. Petri net analysis of batch recipes is further treated in Chapter 9.

8.6 Distributed Execution

The recipe alternatives proposed implicitly assumes a centralized execution environment where the recipe execution and the equipment controllers all reside within the same computer. In an industrial batch control system this is very seldom the case. Instead, a distributed environment is

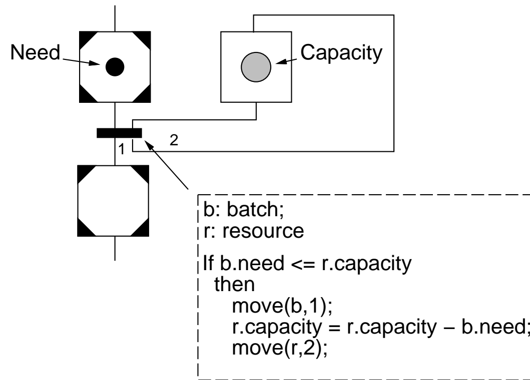


Figure 8.26 A Grafchart structure representing a sharable resource with real-valued capacity constraints.

used where, e.g., each equipment controller is implemented as a separate node. The different nodes communicate with each other over a network. The question is then how well the presented recipe alternatives are suited for distributed execution.

The simplest way to obtain distributed execution is to assume that all recipe management, including recipe execution, is performed in a single supervisory node. The linking between the recipe level and the execution control level, that previously was realized by a Grafchart method call, will now be implemented as a remote procedure (method) call. A drawback with this approach is that it is still, by large, centralized in nature. If the supervisory recipe execution node fails all recipe execution in the entire batch cell will stop.

True distributed recipe execution can only be obtained by sending recipe information between the different equipment controller. Assume that all the equipment units that a control recipe will use are prespecified by the operators before the recipe is started. This is done in one of the nodes of the system. When the recipe is started the recipe is sent to the first equipment node. When the execution in the first equipment node is finished, the recipe (or only the remaining part of the recipe) is sent to the next equipment node, etc.

Depending on which structuring alternative that is used different amount of information has to be passed between the equipment nodes. If each control recipe is represented by a separate function chart, then the function charts have to be passed between the nodes. If, however, function charts are used only to represent the master recipes and object tokens are used

for representing the individual batches, then things become a bit simpler. It is now possible to store a copy of each master recipe function chart in each node, and to pass only the object token information together with information about the current step of an object token, between the nodes.

If the resource allocation should be integrated with the recipe execution as, e.g., in Figure 8.9 or 8.11, things become more complicated. One possibility is to let one equipment node for each equipment type, be responsible for the allocation of resources of that type. When a recipe needs to allocate a resource of a certain type, a request is sent to the resource allocation node for the equipment type. The batch will wait in the preceding equipment unit until a suitable equipment unit becomes available. Information about the identity of the new equipment unit is sent back to the equipment node where the batch is currently waiting, and the transfer to the new equipment unit can start. A rather elaborate handshaking procedure is needed to implement this synchronization. Distributed recipe execution of this nature is available in modern commercial batch control system such as, e.g., the batch control system in SattLine from Alfa Laval Automation.

8.7 Grafchart Structured Recipes vs Industry Practice

The status of commercial batch control systems of today is presented in Chapter 6.5. Most systems follow, to a greater or lesser extent, the ISA S88.01 standard concerning the recipes and their structuring and execution. The Grafchart structured recipes presented in this chapter also follow the standard. The recipes in commercial systems and the Grafchart structured recipes have some similarities but they also have a lot of differences. In this section, the similarities and differences are pointed out and discussed, and the advantages with Grafchart structured recipes are presented.

Recipe structures Both the commercial system and the Grafchart recipe system have a graphical representation of the recipes. According to the standard, a recipe is built up hierarchically. It is allowed to leave out one or several of the layers. However, in most commercial systems it is not possible to skip a layer. Using Grafchart, the layers can be treated with great freedom, compare Figure 8.2, Figure 8.5 and Figure 8.7. Each layer contains a structure of steps and transitions. In the recipes in commercial systems, the steps and the transitions can be connected in any order whereas they have to appear in an alternating sequence (step, transition, step, transition, etc) in the Grafchart structured recipes. The fact that the steps do not have to be placed one after the other might, in certain

cases be "space-effective" but it can also cause confusion for the user. In Figure 8.27 a possible sub-sequence of a recipe of a commercial system is shown. The interpretation of this structure is ambiguous.

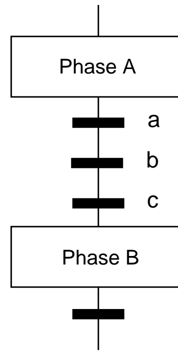


Figure 8.27 A possible sub-sequence of a commercial-structured recipe.

The substructure can be interpreted in many different ways, some of them are shown in Figure 8.28. The left part of the figure corresponds to an interpretation where the execution of the upper step should terminate as soon as the uppermost transition becomes true but where the execution of the lower step should not start until the bottom-most transition has fired. The middle part of the figure corresponds to an interpretation where the execution of the upper step should not terminate until all transitions are true. A third possible interpretation of the recipe from the commercial system in Figure 8.27 is shown in the right part of Figure 8.28. In this alternative the ordering between the three transitions are of importance. Yet other interpretation alternatives are possible. Using Grafchart, unambiguities like this do not exist.

Recipe execution The main and the most important difference is the token concept in the Grafchart-structured recipes. The fundamental role of the token is to indicate the step that is currently active. In a commercial-structured recipe this is indicated by colors. However, as has been shown in the recipe structures presented earlier in this chapter, the role of the token can be extended and a token may contain a lot of information. The same amount of information can not be stored in the "colors" used in a commercial-structured system. The fact that a token can contain a lot of information makes it possible to reduce the size of the representation of the recipes.

In a recipe of a commercial system, the overall plant status is represented textually, compare Figure 6.9. The reason for having a textual representa-

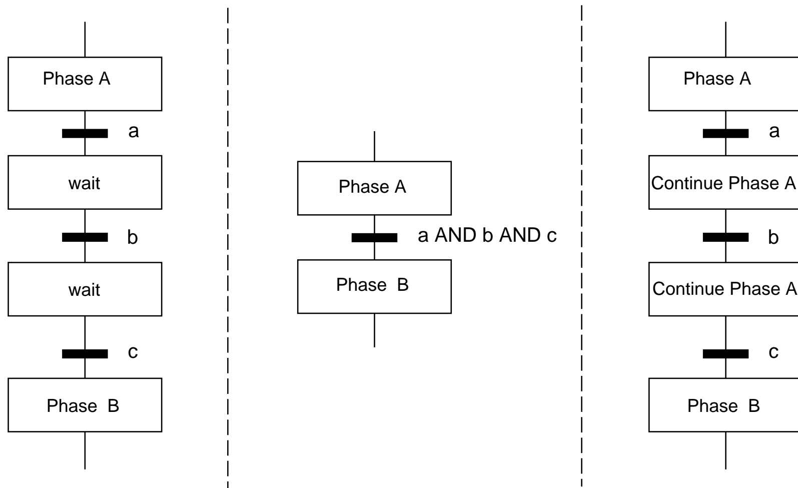


Figure 8.28 Some possible interpretations of the commercial-structured recipe in Figure 8.27.

tion is that there does not exist a compact way of representing the status of all recipes. Using Grafchart and the object token feature, the status of the plant could easily be represented graphically in a nice way, compare Figure 8.16. By allowing the token to contain information, the required logged batch data can be stored in the token in a convenient way.

Resource allocation In most commercial batch systems of today, there are no means for recipe analysis. In order to guarantee that the execution of batches does not enter a deadlock situation, the user can pre-allocate all resources needed for the production of a batch. This is, as will be discussed in Chapter 9.1, very conservative and non-effective. There are no means for deadlock detection, deadlock prevention or deadlock avoidance. How recipe analysis can be combined with the Grafchart structured recipes will be discussed in Chapter 9.5. Grafchart does also, in contrast to the commercial systems, allow for a visualization of the resource allocation.

Recipe implementation Even though most recipes in commercial systems have a graphical interface, the implementation of the lowest level in the recipe, i.e., the phases, is most often done in a textual language. Using Grafchart, the same language can be used on all levels in the implementation of a recipe. This helps both the developer and the user, since they feel comfortable when writing/reading a program.

8.8 Other Batch Related Applications of Grafchart

Grafchart has been used in other batch related applications.

Automating Operating Procedure Synthesis

Plant personnel often spend considerable amount of time and effort, preparing and verifying the recipes, i.e., the operating procedures. It would therefore be valuable to automate the synthesis. A framework for this task is presented in [Viswanathan *et al.*, 1998a]. The framework is based on Grafchart. Grafchart is used to represent the procedural knowledge. The declarative knowledge, i.e., the plant and process specific knowledge, is represented in an object-oriented way. Grafchart is also used to model the information that is incrementally generated during operating procedure synthesis, the so called inferred knowledge. An hierarchical planning strategy is presented that processes the declarative knowledge, utilizing the procedural knowledge, to generate the inferred knowledge incrementally, which leads to the synthesis of the operating procedures.

The implementation of this framework, named *i*TOPS (Intelligent Tool for Operating Procedure Synthesis), is described in [Viswanathan *et al.*, 1998a] and [Viswanathan *et al.*, 1998b]. *i*TOPS is implemented in G2. The application of *i*TOPS to an industrial case study is also presented in the articles.

An Integrated Batch Plant Operation System

The activities involved in batch plant operation are of a diverse nature and call for a range of different views to be solved efficiently. However, the activities interact strongly with each other and it is therefore not possible to treat the different activities as different problems.

In [Simensen *et al.*, 1996] an integrated batch plant operation system is presented. The information model underlying this system is based on a knowledge base containing different models of the plant. The models depicts the problem area in terms of functions, domain and dynamics. To structure the system both user views and representation views are used. The system has the ISA-S88.01 standard as a basis.

A prototype of the integrated batch plant operation system has been implemented and it has been applied to a simulated batch scenario. Depending on what kind of activity the user would like to perform, different views are desirable. The user uses an extended version of the Control Activity Model of ISA-S88.01 as a navigator metaphor. The integrated batch plant operation system is implemented in G2.

8.9 Summary

The main application of Grafchart is recipe structuring and resource management in multi-product, network-structured batch production cells. Grafchart is well suited for these tasks. By using the features of Grafchart in different ways, large variations can be made when structuring the recipes. They still however, comply with the ISA S88.01 standard. The simplest way is to represent each control recipe with a Grafchart function chart with batch specific information represented as attributes of the function chart. The recipes can also be structured using object tokens. In this case, control recipes of the same type, move around in the same chart. Batch specific information is stored within the token. By using the multi-dimensional structure of Grafchart, the chart in which the tokens reside can be given a general structure. Tokens of different types, i.e., of different master recipes, can now reside in the same chart. The token contains its own recipe-procedure as a method.

Resource allocation can be made in two ways: implicitly or explicitly. If the resource allocation is done explicitly, i.e., if each resource is represented by a token, formal analysis methods can be applied to the net, and properties like deadlocks can be detected. Resource allocation of exclusive-use resources and deadlock-analysis have been looked at in other areas, e.g., Petri nets, and their results can be useful also for a recipe management system in the batch control field.

9

Recipe Analysis

The main objective when developing Grafchart has been to provide a user-friendly high-level language for sequential control applications. In doing so, the work has been inspired by the concepts used in several different areas, e.g., concepts that are normally used within the context of formal modeling languages for discrete event systems. Hence, it is easy to get the conception that Grafchart is also developed with the intention to be a formal modeling language. This is not the case. However, since Grafchart is based on Grafcet and Petri nets and since it is possible to transform a Grafchart into a Grafcet and/or a Petri net, see Chapter 5.8, it is possible to apply some of the formal methods for, e.g., verification, that have been developed for Grafcet and Petri nets, also to Grafchart.

There are advantages of having a programming language that mechanically can be transformed to an analyzable format. In this way the controller does not have to be manually modeled before applying the formal methods, and the risk of introducing errors or inconsistencies is therefore reduced. This means that the Grafchart implementation can be seen as an executable controller model. Still, however, the plant that is controlled by the Grafchart has to be modeled in order for the formal analysis to be complete.

The aim of this chapter is to show how formal methods for Petri nets can be used in the context of recipe-based batch control. The methods are used for deadlock analysis due to improper resource allocation. The approach is based on transforming the combined Grafchart recipe containing all the master recipes together with the shared resources into a Petri net. The size of this Petri net is then reduced and the reachability graph is calculated. By inspecting the reachability graph, deadlock situations can be found. The reachability graph can also be used to detect situations where the firing of a transition in a recipe may lead to a deadlock situation. The analysis results can be used statically to prevent deadlocks from occuring, or dynamically in order to to avoid deadlocks.

A batch cell similar to the one in Chapter 7.3 is used in this chapter. Two simplifications are made. The first one concerns the connectivity of the units in the batch plant. All units in the batch cell are assumed to be fully connected. This means that it is not necessary for a batch to allocate any transportation equipment such as pipes, valves or pumps, in order for the batch to be moved from one unit to another. If this would not have been the case, the transportation equipment would have to be modeled as resources that the batch would have to allocate. The second simplification concerns the units. Units of the same type are all assumed to be equivalent, e.g., it does not matter which of the mixers that is allocated to a batch if there are several mixers to choose among. The reason why this assumption is made is that only a structural analysis is performed. Additional constraints, expressed through the transition receptivities are not taken into account.

The chapter starts, (Chapter 9.1), with a presentation of the general deadlock problem. The consequences that the deadlock problem can have in a batch cell are discussed. Thereafter two batch recipes are presented (Chapter 9.2). The recipes are structured with Grafchart and they are assumed to be executed in the same plant. After a general discussion about the Grafchart to Petri net transformation, the two Grafchart structured recipes are transformed into Petri nets (Chapter 9.3). The Petri net structured recipes are analyzed (Chapter 9.4) and different ways of using the results are presented and discussed. (Chapter 9.5). The chapter also contains a presentation of other analysis approaches, (Chapter 9.7), and a presentation of the relation between resource allocation and scheduling, (Chapter 9.8).

9.1 The Deadlock Problem

The most efficient batch plants are the multi-product, network structured plants where several batches, of the same or of different types, can be produced at the same time. The batches share the equipment in the plant. If the allocation of the equipment is handled improperly a deadlock situation might occur, e.g., if a batch currently in one unit requires another unit and if that unit is reserved by another batch that requires the unit held by the first batch. Analysis can be performed on the recipes in order to look for certain properties, e.g., deadlock situations. The analysis methods can also be used to look for other properties, e.g., safe, bounded, home-states, reversible, etc.

Four conditions exist that must be fulfilled for a deadlock situation caused by improperly handled resource allocation, to occur, [Burns and Wellings, 1996]:

1. The batches need to share resources that are such that they can only be used by one batch at a time.
2. Batches exist that are holding a resource while waiting for another resource.
3. A resource can only be released voluntarily by the batch.
4. A circular chain of batches exist such that each batch holds resources which are requested by the next batch in the chain.

Deadlock situations are always unwelcome since they result in additional work. To recover from a deadlock one might have to use an additional unit. One of the batches is moved to this unit and the other batches can thus proceed. In some cases, however, a deadlock situation might lead to the deletion of an entire batch. This can, e.g., be the case when all units are utilized by batches. In order to get out of the deadlock situation one resource has to be liberated and this can only be done by deleting one batch. A deletion of a batch is costly and should therefore be avoided. Batches do also exist that are very sensible and that, after one operation, e.g., a reaction operation, immediately requires that the following operation, e.g., cooling, starts. If this is not the case, the quality of the batch decreases. If the quality does not fulfill the requirements, the batch must be deleted, or its price must be reduced. It is therefore important to know, not only when and why a deadlock situation can arise but also how they can be avoided.

If a deadlock situation has occurred, it is sometimes possible to recover from it by temporarily moving one of the batches to an unreserved unit in the plant. In this way, the batches can proceed and all batches involved in the deadlock can be completed. This method is refered to as deadlock detection and recovery, [Åkesson and Tittus, 1998]. Detection and recovery is unsatisfactory since it interrupts the production. A better solution is achieved by using the analysis results to make sure that a deadlock situation never occurs. This can be done by restricting the possible resource allocations, either statically (deadlock prevention), or dynamically (deadlock avoidance), [Åkesson and Tittus, 1998].

The allocation of the resources must be restricted in such a way that one or several of the four conditions above do not hold.

1. Removing condition number 1 is often unrealistic. Almost all batch plants have a shared resource structure where a resource only can be used by one batch at a time.

2. Condition 2 can be removed if the batch pre-allocates all resources needed during its production before it starts. However, this is a very conservative method which implies that the resources are not used in an optimal way.

3. Removing condition 3 means that it must be possible to force a batch to release a resource. This implies that the batch has to be deleted or that there are additional resources to which the batch can be transported.

4. Condition 4 can be removed if an ordering between the resources is defined, e.g., an hierarchical ordering scheme, and it is required that the resource allocation is always done in this order. This can however, also be a relatively conservative method.

How to restrict the allocation of resources is discussed further in Chapter 9.5.

9.2 Batch Recipes

The batch recipes presented in the examples in this section are supposed to be run in a cell containing three storage tanks, three product tanks, two reactors and one mixer. In addition to this the cell contains one extra mixer that can be connected if desired. One of the reactors can be used for heating, rector$_{heat}$ and one for cooling, reactor$_{cool}$. The cell is shown in Figure 9.1.

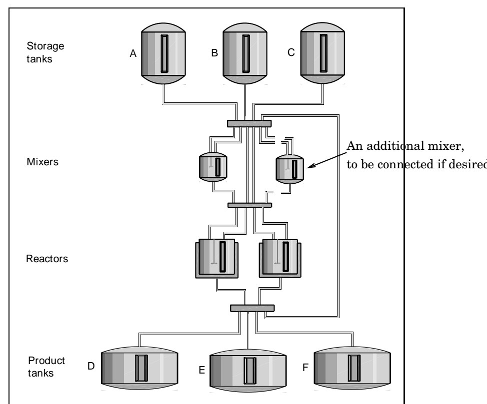


Figure 9.1 A batch plant.

The batches that can be run in the plant originate from two different master recipes called recipe-A and recipe-B. Both recipes are structured with Grafchart (object tokens without methods). Recipe-A is shown in Figure 9.2 (left). The recipe starts by assigning a mixer to the batch. This is indicated by the first transition. After the assignment the two reactants A and B are filled into the mixer, and the mixture is agitated. Thereafter a reactor is assigned and half of the content in the mixer is transfered to the reactor. The other half of the batch is left in the mixer. The content in the reactor is heated and then transfered to another reactor where it is cooled. Two different reactors, $reactor_{heat}$ and $reactor_{cool}$, are used for the heating and the cooling. After the transfer from $reactor_{heat}$ to $reactor_{cool}$, $reactor_{heat}$ is no longer needed and it is therefore released. When the content has been cooled down it is transfered back to the mixer where the rest of the batch is waiting. $Reactor_{cool}$ is now released. The mixture is agitated and thereafter discharged. Finally the mixer is released.

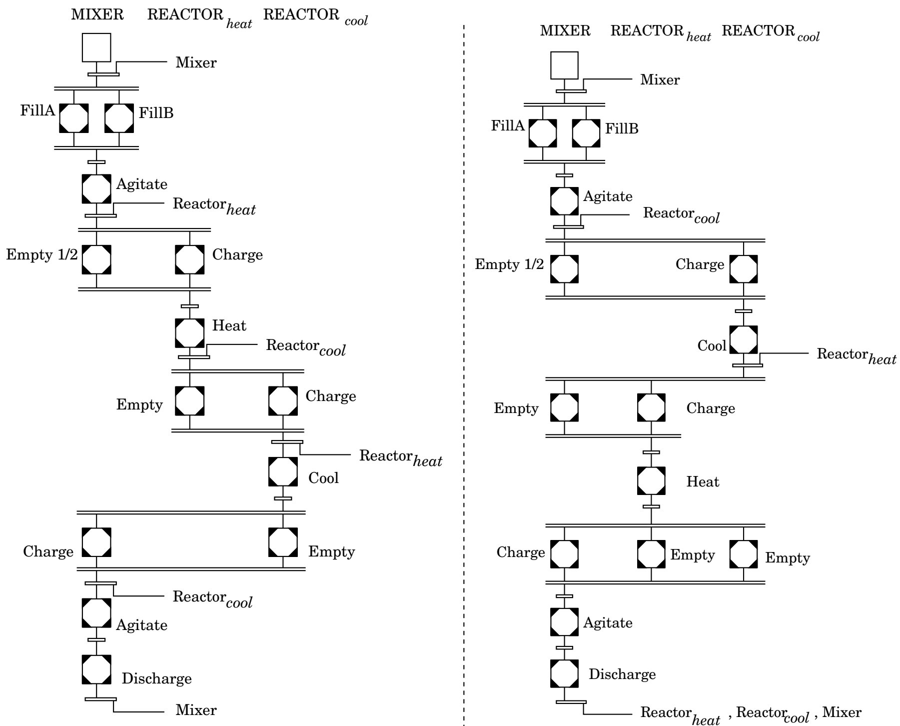


Figure 9.2 Two Grafchart recipe structures: recipe-A (left) and recipe-B (right).

Recipe-B is shown in Figure 9.2 (right). This batch mixes two reactants in a mixer, and then transfers half of the content to a reactor for cooling.

When the cooling is ready, the other half of the batch is transfered from the mixer to a reactor for heating. Finally the content of the two reactors are transfered back to the mixer for final agitation and discharging.

Since batches of the two recipes are produced in the same plant and therefore utilize the same units, they will effect each other. The two recipes can be combined into one Grafchart where the resource allocation is visualized, compare Chapter 8.2 (Combined Network). This is shown in Figure 9.3. The number of tokens placed in the initial step in each of the recipes corresponds to the aim of producing the same number of batches.

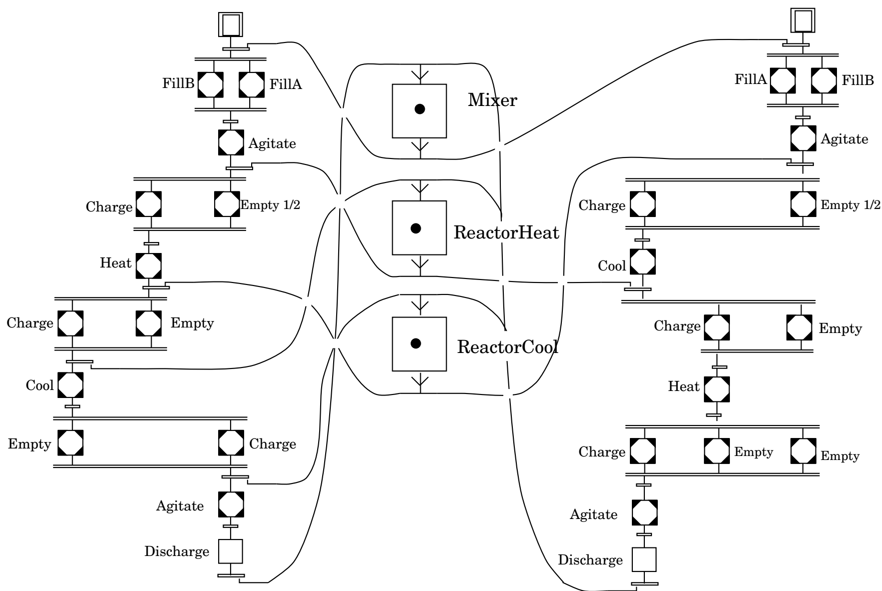


Figure 9.3 Two recipes combined into one Grafchart.

Both recipes are structured with phases. According to the ISA S88 batch standard, a phase must always execute within the same unit, see Chapter 6.2 (Procedural Control).

Remark: The batch cell as well as the two recipes are not based on a real scenario but have been made up in order to illustrate the analysis concepts as clearly as possible.

9.3 Batch Recipe Transformation

In order to analyze a recipe with respect to deadlock situations, the Grafchart structure is transformed into a Petri net structure. The analysis methods presented in Appendix B apply to autonomous Petri nets. This means that the corresponding Petri net will not have any transition receptivities nor will it, since it is a Petri net, have any step actions. The question is then how the lack of actions and receptivities will influence the analysis results. Can a Grafchart that is deadlock free be transformed into a Petri net that, when analyzed, claims that the Petri net could have a deadlock? If the corresponding Petri net is deadlock free, does this always mean that the Grafchart is deadlock free or do there exist cases when the Grafchart could get into a deadlock situation? The different questions at issue are illustrated in Figure 9.4.

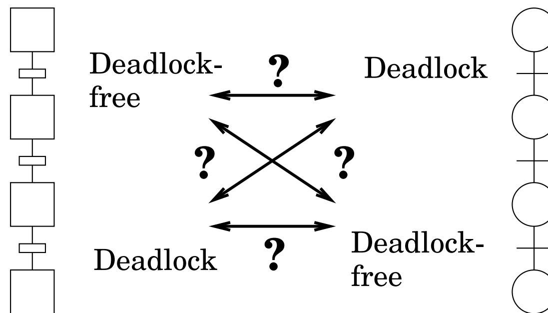


Figure 9.4 Deadlock relation between Grafchart and Petri net.

In order to be able to make a transformation between a Grafchart and an autonomous Petri net some assumptions and restrictions have to be made. What differs a Grafchart from a Petri net are, compare Chapter 5.8 (Grafchart - *the basic version*: Transformation to Grafcet): the graphical elements, the actions and receptivities in Grafchart, and the dynamic behavior.

Graphical elements: The graphical syntax for steps, transitions, parallel branches and alternative branches are not identical in Grafchart and Petri nets, see Figure 9.5.

Grafchart contains a larger number of graphical elements than Petri nets. The additional graphical elements that exist in Grafchart must therefore be replaced with a larger number of Petri net elements. Macro steps, procedure steps and process steps should be replaced with the internal structure or the internal structure of the associated Grafchart procedure, see Chapter 5.8 (Grafchart - *the basic version*: Transformation to Grafcet).

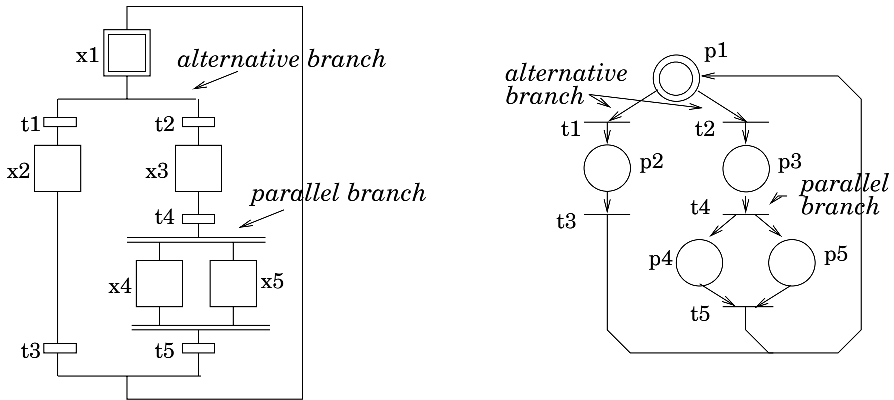


Figure 9.5 A Grafcet (left) and a Petri net (right).

Receptivities and actions: An autonomous Petri net does not have any receptivities associated with the transitions. A Grafchart receptivity can either cause or prevent a deadlock situation.

In Figure 9.6 (left) a Grafchart structure is shown. The Grafchart contains an or-divergence situation with two branches. The right branch loops back to the step preceding the or-divergence situation while the left branch ends in a step without any succeeding transitions, i.e., a possible deadlock state. However, since the transition receptivity in the left branch is equal to false the token can never come to this deadlock state. The transition receptivity prevents a deadlock from occuring. The corresponding autonomous Petri net, see Figure 9.6 (right), does not have any transition receptivities and is therefore not deadlock-free.

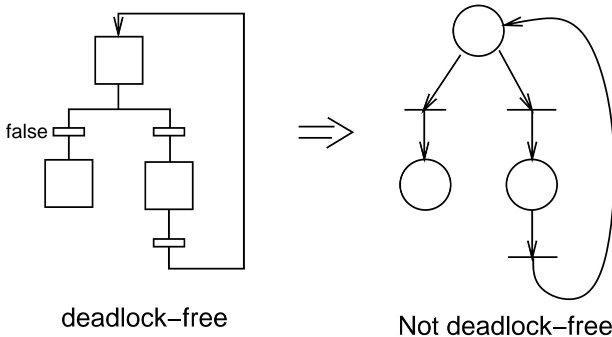


Figure 9.6 A Grafchart receptivity that prevents a deadlock situation.

In Figure 9.7 (left) a Grafchart structure is shown. The Grafchart contains an or-convergence with two branches. However, the transition receptivity associated with the left branch is equal to false which means that if the token enters this branch branch it can never leave, i.e., a deadlock situation has occurred. The transition receptivity causes a deadlock situation. The corresponding Petri net, see Figure 9.7 (right), does not have any transition receptivities and can therefore not get into this deadlock state.

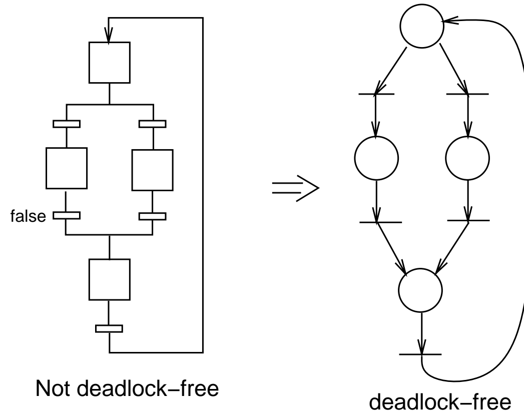


Figure 9.7 A Grafchart receptivity that might cause a deadlock situation.

A Petri net does not have any actions associated with the places. A Grafchart action can cause a deadlock situation to occur. An example of this is given in Figure 9.8. The Grafchart structure in the figure contains two steps and one transition. The action associated with the first step sets the transition receptivity to false which means that the transition never can fire and a deadlock situation has therefore occurred. A corresponding Petri net would not contain the action nor the receptivity and could therefore not result in the deadlock situation.

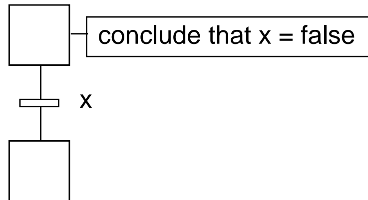


Figure 9.8 An action that cause a deadlock situation to occur.

Reversed scenarios where an action prevents a deadlock situation to occur also exist.

As has been shown, actions and receptivities might cause deadlock situations to occur. These situations cannot be found by analyzing the corresponding Petri net. Action and receptivities can also prevent deadlock situation from occuring. In these cases the corresponding Petri net will not always be deadlock-free.

Dynamical behavior: The dynamical behavior of a Grafchart and a Petri net differs with respect to the firing instance of a transition. In Grafchart an enabled transition fires immediately whereas the firing instance of an enabled transition in a Petri net is not uniquely given. However, this does not influence the reachability graph and is therefore of no importance for the analysis results.

Remark As has been shown it is only the topology, i.e., the graphical structure, of a Grafchart that can be analyzed. Influences from step actions and transition receptivities can not be taken into account.

Recipe transformation procedure When transforming a Grafchart into a Petri net, the transformation rules presented in Chapter 5.8 (Grafchart - *the high-level version*: Transformation to Petri nets) should be used. Since the effect of a step action or a transition receptivity not can be analyzed the transformation rule number 4, 5 and 9 can be replaced with the simplified rules:

4. The actions are neglected since the influence of an action not can be analyzed.
5. The receptivities are neglected since the influence of a receptivity not can be analyzed.
10. Parameterization of receptivities are of no importance because of (5).

Using the transformation rules, the recipe structure of Figure 9.3 can be transformed into a Petri net. The corresponding Petri net is shown in Figure 9.9. Recipe-A is placed to the left in the figure and Recipe-B to the right. In addition to the transformation rules, the fact that a phase always executes within the same equipment (no resource allocation or deallocation allowed), is used. This implies that each procedure step, corresponding to a phase, can be replaced by one single Petri net place, i.e., the procedure step does not have to be replaced by the internal structure of the corresponding Grafchart procedure.

The Petri net structure shown in Figure 9.9 has two initial places, one in each recipe. In each initial place one token is placed. If the dotted

transition and dotted arc that closes the loop are disregarded, the tokens indicate that one batch of each type should be produced. If, however, the Petri net structure is closed, i.e., if the dotted transition and arc are taken into account, the tokens placed in the initial-places instead indicate that at most one batch of each type should be produced at the same time in the plant. When a batch of, e.g., type A, has been produced, the token can be returned to the initial place and the production of a new batch of this type is allowed to start.

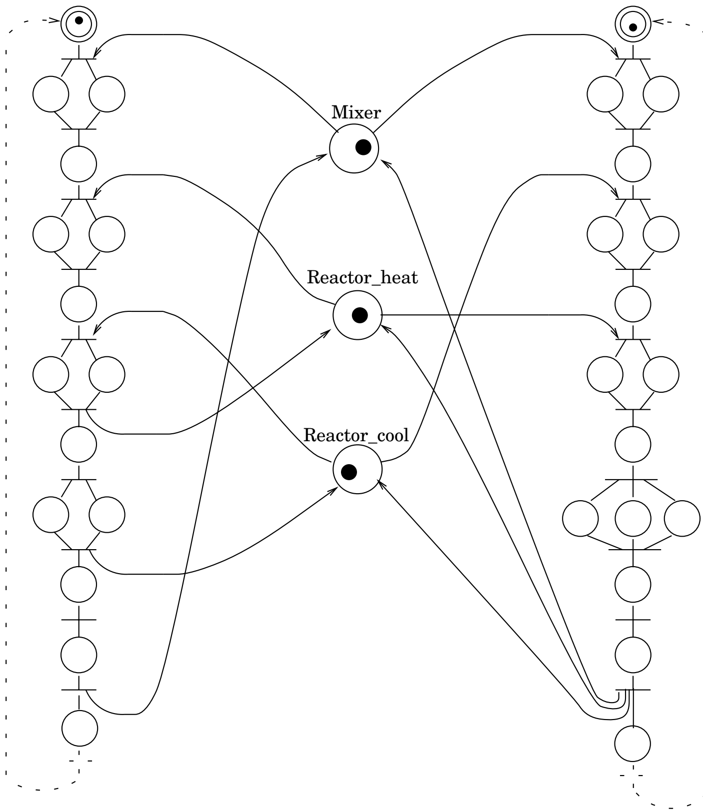


Figure 9.9 A Petri net recipe structure corresponding to the combined Grafchart recipe in Figure 9.3.

The two transformed recipes in Figure 9.9 can now be reduced according to the reduction rules presented in Appendix B, see Figure 9.10. The transformed recipe on the left side in the figure corresponds to Recipe-A and the transformed recipe on the right side corresponds to Recipe-B. The Petri net structure can be open or closed.

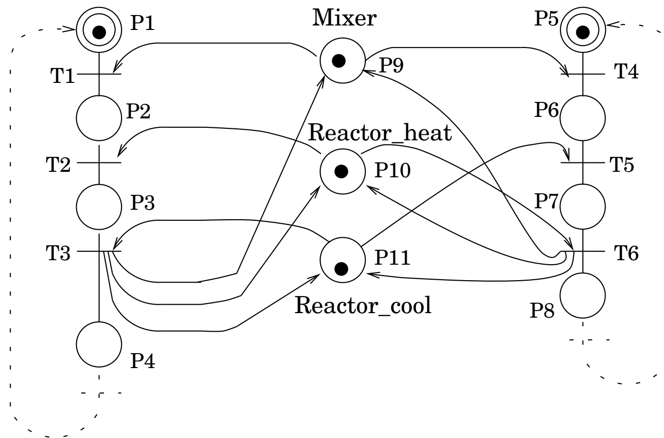


Figure 9.10 The reduced Petri net recipe structure for to the Petri net recipe structure in Figure 9.9.

9.4 Analysis

If only one batch of each type should be produced, the system cannot get into a deadlock situation. This can be shown by drawing the reachability graph, see Figure 9.11. The dotted lines correspond to the return of the

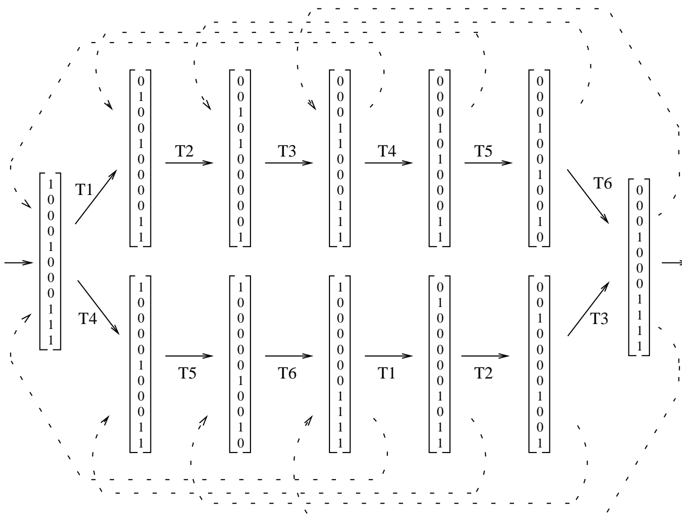


Figure 9.11 Reachability graph.

token to the initial place. As can be seen, these lines do only loop back to already existing states and they can therefore not lead to a new deadlock situation. In the sequel of this chapter, these lines are left out in all reachability graphs.

What if we add an extra resource to the plant, e.g., connect the additional mixer that we have in the plant, see Figure 9.1? The intuitive reaction is that adding an extra resource cannot lead to a deadlock since deadlocks occur when there are to few resources. Thus, adding an extra resource cannot cause the system to get into a deadlock situation. However, by drawing the reachability graph one can show that this is not true. This means that informal arguments about behavioral properties are dangerous, [Jensen, 1992]. The reachability graph, corresponding to Figure 9.10, using two mixers, is shown in Figure 9.12.

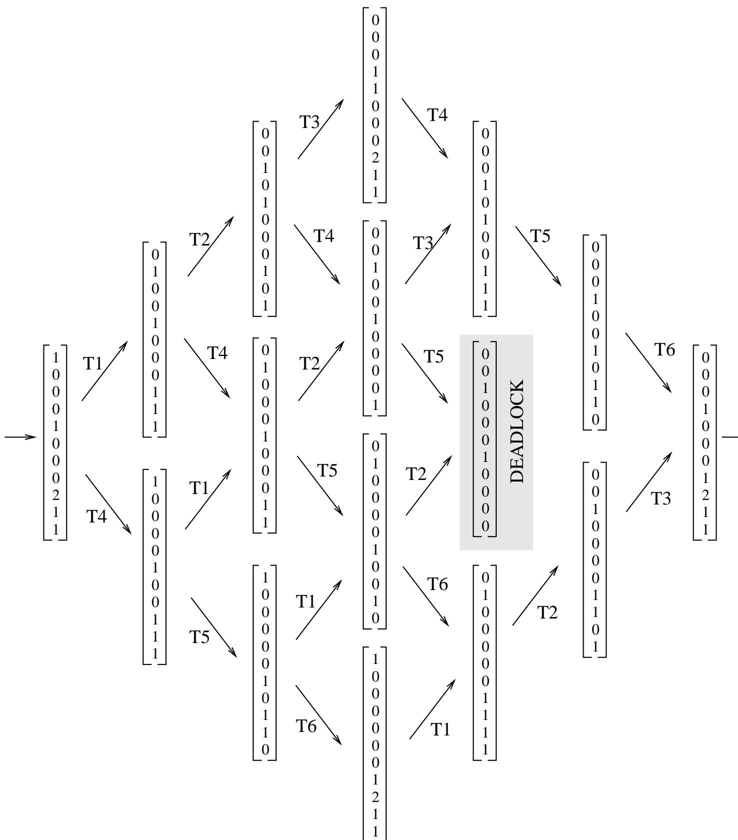


Figure 9.12 Reachability graph.

The deadlock situation that arises when a second mixer is added to the plant can intuitively be understood by examining Figure 9.10. The two recipes use reactor$_{heat}$ and reactor$_{cool}$ in reversed order, this means that if a deadlock situation should occur, it is due to this. When there is only one mixer in the plant, the mixer assures that a second batch cannot start until the first batch has finished, and a deadlock situation can therefore not occur.

Situations where the resource has to fulfill certain constraints, e.g., a minimum value or a high temperature tolerance, can be implemented by using the transition conditions in Grafchart-HLV. Examples of this are given in Chapter 8.5 (Shared Resources). Since the analysis methods presented do not take the receptivities into account, the specific demands that a batch can have on a resource are not considered. To solve this problem, resources with different capacity must reside in different steps.

A drawback with the reachability graph method is that the complexity increases rapidly when the size of the net increases. However, this drawback is also present in most formal analysis methods for this type of problems.

9.5 Using the Analysis Results

The analysis results can be used both statically and dynamically. Static reasoning deals with the question of which combinations of number of batches and number of units that can be safely started in the plant at the same time. The answer to this question can, e.g., be stored in a table. Dynamic reasoning is about determining, depending on the status of the plant, which units that are unsafe to allocate and therefore not should be reserved by a batch, i.e., to find the unsafe states. To assure that the system never reaches an unsafe state a supervisor can be used.

Static Deadlock Prevention

Analysis can be performed and the results can be stored in a table. The results can, e.g., show the number of batches in different recipe combinations, that can execute at the same time in the plant without risk for a deadlock. Figure 9.13 shows“possible-deadlocks” with respect to the two recipes, versus different plant configurations. Assume that the default plant is 3 mixers (M), 1 reactor$_{heat}$ (Rh) and 2 reactor$_{cool}$ (Rc). The table shows the “risk of deadlocks” for the default plant as well as for the plant with one or several of the equipments broken down.

The “risk-of-deadlocks” depends on the number of batches that are produced in the plant. The uppermost row in the table shown in Figure 9.13

corresponds to the production of two batches of recipe-A and two batches of recipe-B. The three rows below correspond to the production of sub-adjacent cases. If the uppermost row indicates that the production in a certain plant configuration is “ok”, then it follows that the production of the sub-adjacent cases are also “ok”. This is the case of column two, three and six. A similar reasoning can be done for the bottommost row. This row corresponds to the production of one batch of recipe-A and one batch of recipe-B. The three rows above corresponds to the production of a higher number of batches. If the bottommost row indicates that the production in a certain plant configuration is “not-ok”, it can be concluded that the production of the three other rows are “not-ok” either. This is the case in column four and five.

	3M+1Rh+2Rc	2M+1Rh+2Rc	1M+1Rh+2Rc	3M+1Rh+1Rc	2M+1Rh+1Rc	1M+1Rh+1Rc
2* Recipe-A + 2* Recipe-B	--	ok	ok	--	--	ok
2* Recipe-A + 1 Recipe-B	ok	ok	ok	--	--	ok
1 Recipe-A + 2* Recipe-B	--	ok	ok	--	--	ok
1 Recipe-A + 1 Recipe-B	ok	ok	ok	--	--	ok

Figure 9.13 A table showing the possibility of deadlock for different recipes in different plant configurations, ok = deadlock not possible, – = deadlock possible.

The table can be used as a help when determining how to prevent deadlock situations from occurring in a batch plant. According to the table in Figure 9.13 two batches, one of type A and one of type B, can be run in the same plant simultaneously with no risk of entering a deadlock situation if the plant consists of one, two or three mixers together with one $reactor_{heat}$ and two $reactor_{cool}$ or if the plant consists of one mixer, one $reactor_{heat}$ and one $reactor_{cool}$. These combinations of batches versus plant configurations can be run without any deadlock considerations. However, if the plant consists of two or three mixers together with one $reactor_{heat}$

and one reactor$_{cool}$ there is a risk of getting stuck in a deadlock situation. In order to completely eliminate this risk, this combination of batches versus plant configuration should never be run.

Using the analysis results in this way is an example of deadlock prevention. However, this method is very conservative. Even if the starting of a certain number of batches, e.g., one of type A and one of type B, in a certain plant configuration, e.g., two mixers, one reactor$_{heat}$ and one reactor$_{cool}$, might lead to a deadlock situation, this is not always a consequence. The execution of this batch-vs-plant combination can complete normally. This can be seen in the reachability graph shown in Figure 9.12. However, in order to eliminate the risk of getting into a deadlock situation, all cases that might lead to a deadlock situation are statically excluded. The table based method is more conservative than needed and might, in a worst case, lead to the conclusion that no batches should be started in any plant configuration.

Dynamic Deadlock Avoidance

Dynamic deadlock avoidance can be realized through a forbidden state approach. The forbidden state method is less conservative than the table method. In the table based deadlock prevention method, the starting of entire batches are excluded. In the forbidden state method it is a matter of excluding only the firing of certain transitions, i.e., the allocation of certain resources. The transitions that should be excluded vary with the situation in the plant, i.e, dynamically.

Figure 9.12 shows the reachability graph for the case when the plant consists of two mixers, one reactor$_{heat}$ and one reactor$_{cool}$ and when two batches, one of type A and one of type B, are assumed to be produced in the plant. The initial marking of the plant is $[10001000211]^T$. As can be seen in the reachability graph, it is possible to get into a deadlock situation. By using the table, Figure 9.13, the user is adviced not two run the two batches in the plant. However, the deadlock situation can be avoided by preventing the firing of transition `T2` when the marking is $[00100010000]^T$ and preventing the firing of transition `T5` when the marking is $[00100100001]^T$. The reachability graph, with these two transitions indicated, is shown in Figure 9.14.

More advanced reasoning can also be included when using the forbidden state method. Assume, for example for recipe-B, that when the two substances have been filled in to the mixer and agitated, a Reactor$_{cool}$ is immediately needed, i.e., the batch is not allowed to wait for this unit. Can this requirement be guaranteed by excluding the firing of certain transitions? Place `P6` in Figure 9.10 corresponds to the filling and agitating phases

in recipe-B, compare Figure 9.9. In the reachability graph, Figure 9.12, being in this place corresponds to the marking $[*****1******]^T$ ($*$ = any number). Whenever in this state there must always exists a possibility to fire transition `T5`, since the firing of this transition corresponds to the allocation of a reactor_{cool}.

There are four markings that fulfill $[*****1******]$, these are encircled in Figure 9.15 (left). From marking $[00010100111]^T$ it is possible to fire only transition `T5`, i.e, being in this state will cause no problem concerning the requirement. From marking $[00100100001]^T$ it is not possible to fire transition `T5` directly since this would result in a deadlock, but it is possible to first fire transition `T3` and then transition `T5`. However, this might

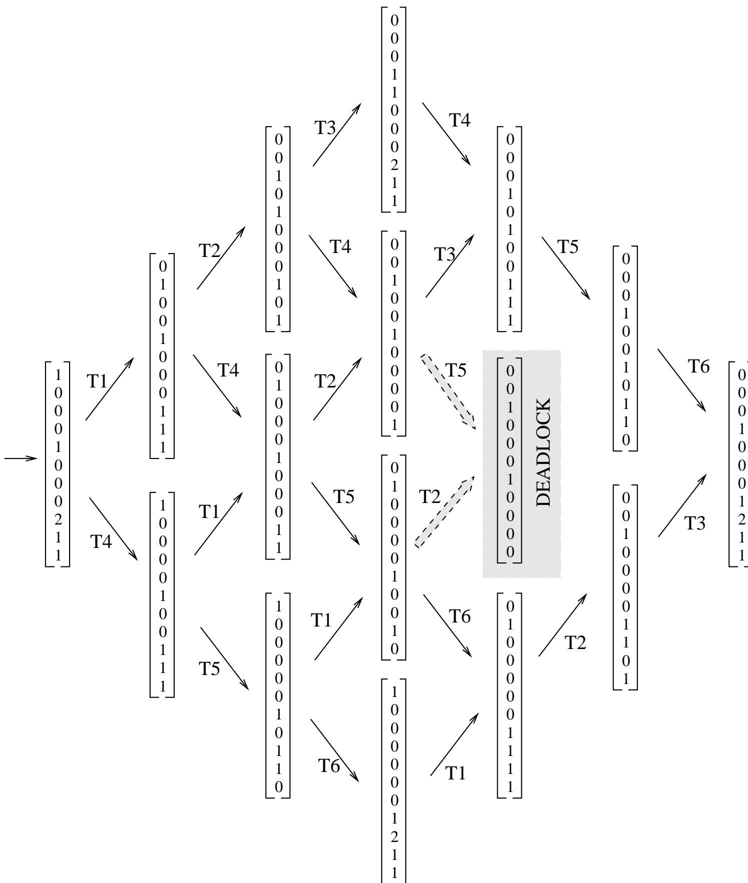


Figure 9.14 Reachability graph.

require that the batch will have to wait for transition T3 to fire. In order to guarantee that such a wait does not occur, the state represented by the marking $[00100100001]^T$ should be considered a forbidden state and to avoid arriving in this state, the firing of the two transitions T4 and T2, leading to this state, must be excluded. From marking $[01000100011]^T$ it is only possible to fire transition T5 since transition T2 has been excluded. From marking $[10000100111]^T$ it is possible to fire transition T5 either directly or to fire transition T1 and then transition T5. The reachability graph with the necessary transitions excluded, i.e., the reachability graph that fulfills the requirement, is shown in Figure 9.15 (right).

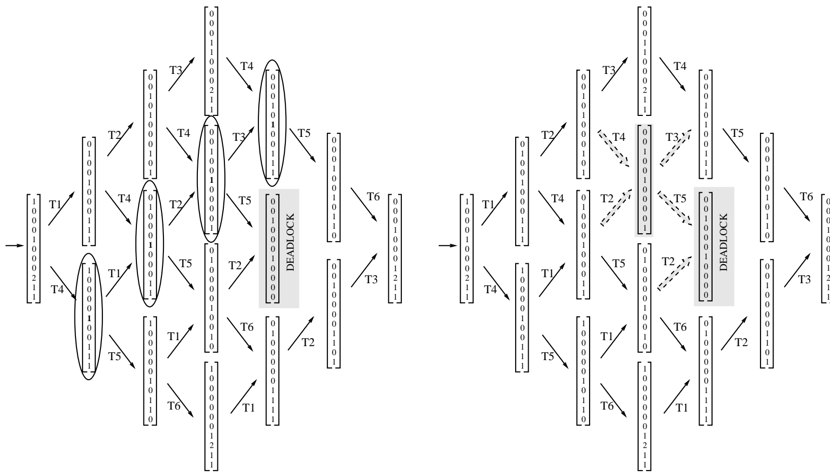


Figure 9.15 Reachability graphs.

Supervisor

The results from the forbidden state analysis can be implemented by a supervisor using the supervisory control theory (SCT) of Wonham and Ramadge, [Ramadge and Wonham, 1989]. The theory can be formulated both in terms of automata and formal languages. A modified version of SCT has been developed by Charbonnier and Alla, [Charbonnier, 1996]. The modified version is based on Grafcet and makes a clear distinction between the controller and the supervisor. Yet an alternative approach, called Procedural Control Theory (PCT), has been developed at Imperial College, [Alsop *et al.*, 1996].

SCT In SCT, [Ramadge and Wonham, 1989], a process is modeled as an automaton that generates events. The set of possible events, denoted Σ, can be divided into two subsets: controllable events, denoted Σ_c and

uncontrollable events, denoted Σ_{uc}. The controllable events can, unlike the uncontrollable ones, be prevented from occuring. The task of the supervisor is to enable or disable the controllable events in such a way that certain process states are not reached. A process together with a supervisor is shown in, Figure 9.16.

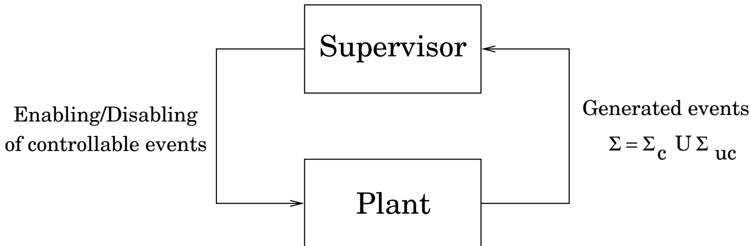


Figure 9.16 A process and a supervisor according to SCT.

SCT can be used for both verification and synthesis. The verification approach consists of modeling the open system, i.e., the process, and constructing a supervisor manually. Both the open system and the supervisor are modeled in terms of a formal language. The language of the closed loop system, i.e., the process together with the supervisor, is then formed. Finally, it is checked if the closed loop language is controllable. If so, the user knows that the supervisor guarantees that the process never enters certain undesired states. What remains for the user to do is to implement the supervisor.

The synthesis approach consists of modeling the open system, i.e., the process, and specifying the desired closed loop behavior. The open system and the closed loop system are both defined in terms of a formal language. The controllability of the closed loop language is checked and a supervisor is generated automatically. The supervisor is given in terms of a formal language. What remains for the user to do is to implement the supervisor. The verification approach is more common than the synthesis approach. A major reason for this is the difficulties involved with specifying the closed loop system completely for users without training in formal languages.

SCT has received a lot of criticism:

- The theory is difficult to grasp.
- It can only be applied to small practical systems. This is due to the combinatorial complexity (state space explosion).
- The process is viewed as an event generator. In most real problems, the process reacts to inputs and generates outputs.

- SCT literature provides few indications of how to implement a supervisor.

SCT has been used for batch applications, [Tittus, 1995] and [Åkesson and Tittus, 1998], see Chapter 9.7.

Grafcet Supervisor In the SCT approach a supervisor can prevent some events from occuring, but it can not force any events to occur. In addition to a supervisor, most real systems need a controller that can force events to occur. A modified version of SCT has been defined by Charbonnier and Alla, [Charbonnier, 1996]. The new approach allows for a clear distinction between the process, the controller and the supervisor. The approach is based on Grafcet, i.e., the supervisor and the controller are represented in Grafcet instead of in a formal language.

The process and the controller constitute the extended process. The events generated by the process, denoted Σ_{pr}, are assumed to be non controllable, $\Sigma_{pr} \subseteq \Sigma_{uc}$. These are typically the output from the sensors in the plant. The events generated by the controller, denoted Σ_{co} are assumed to be potentially controllable. However, in a specific case they need not all to be controllable, i.e., $\Sigma_c \subseteq \Sigma_{co}$. The events generated by the controller are typically control actions effectuated by the actuators in the plant. The task of the supervisor is to enable and disable some of the controllable events in such a way that the extended process never reaches certain undesired states. The list of authorized events provided by the supervisor is denoted Γ. In Figure 9.17 a process and controller together with a supervisor is shown.

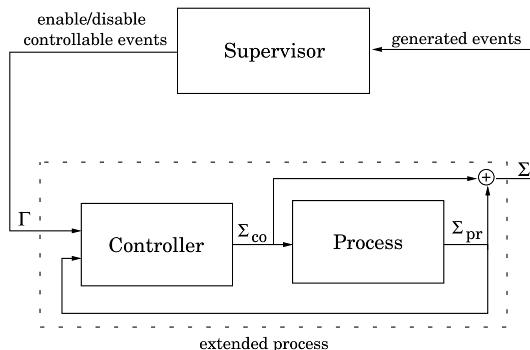


Figure 9.17 A supervisor.

As can be seen, the controller and the supervisor are separated. Also in the design procedure, the two parts are designed independently from one

another. The design of the controller constitutes the control specification for the process. The design of the supervisor constitutes the supervision specification for the process. The extended process and the supervisor are converted into an automata model and the language of the closed loop system is generated. The controllability of the closed loop language can now be checked in the same way as was done in the SCT.

The advantage of the Grafcet supervisory method by Charbonnier and Alla compared with classical SCT is that it often reduces the complexity of the problem. The structure of the Grafcet controller and the structure of the Grafcet supervisor both gets a relatively small size with a clear interpretation.

Grafcet based SCT has not been applied in any batch applications yet. As has been shown, Grafchart resembles Grafcet a lot. The ideas of a Grafcet supervisor could therefore also be applicable for Grafchart. How this can be done is discussed further in the following section.

9.6 Grafchart Supervisor

The events taking place in a batch plant need to be controlled and supervised. The plant can be controlled by a Grafchart controller, i.e., by a recipe. The plant also needs to be supervised, this can be done by a Grafchart supervisor.

The task of the supervisor is to assure that the system never reaches a forbidden state, i.e., to assure that certain transitions in the recipe not are fired. The transitions that should not be fired vary with the state of the system. A plant controlled by a Grafchart controller (recipe) and supervised by a supervisor is shown in Figure 9.18. The supervisor receives

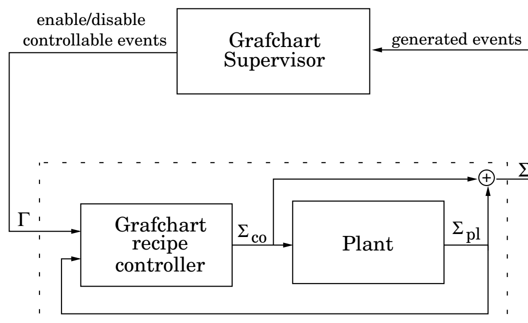


Figure 9.18 A supervisor.

information about the commands sent by the Grafchart controller to the plant Σ_{co}, the supervisor also receives information about the events taking place in the plant, Σ_{pl}. Depending on the new state of the system the supervisor disables or enables events, Γ.

The supervisor presented here does, however, not take the internal state of the equipment units into consideration. Assume, for example, that a mixer in the plant is unreserved but not empty. The supervisor will only disable the allocation of this mixer if an allocation of it it will lead to a deadlock state. This means that the non-empty mixer can be allocated by a batch and the contents of the mixer and the batch will be mixed.

Reachability Graph Based Supervisor

In Chapter 9.5 (Dynamic Deadlock Avoidance), the reachability graph for a system is shown, Figure 9.14. The system consists of two mixers, one reactor$_{heat}$ and one reactor$_{cool}$, and two batches, one of type A and one of type B, should be produced. In order to exclude the deadlock situation certain transitions must be disabled. The supervisor for this particular case is shown in Figure 9.19. An easy way to construct a supervisor is to let it resemble the structure of the reachability graph: each state in the reachability graph is modeled by a step in the supervisor and the arcs in the reachability graph are replaced by transitions in the supervisor.

The arcs in the reachability graph, e.g., Figure 9.14, correspond to the firing of transitions in the reduced Petri net structure, Figure 9.10. An arc takes the system from one state to another. For example, the arc labeled `T1` takes the system from the state $[1000****1**]^T$ to the state $[0100****0**]^T$ and the arc labeled `T2` takes the system from the state $[0100****01*]^T$ to the state $[0010****00*]^T$. A state in the reduced Petri net structure corresponds to one or several states in the unreduced Petri net structure, Figure 9.9. The state, in the reduced Petri net structure, that is reached when `T1` has been fired but not yet `T2`, i.e., state $[0100*******]^T$, corresponds to the states $[0110***....]^T$ and $[00010***.....]^T$ in the unreduced Petri net structure. This is equivalent to the execution of the phases `mixer_fillA` and `mixer_fillB` or of the phase `mixer_agitate` in the Grafchart recipe. This means that the transition named `T1` in the supervisor should be fired when the system has arrived in the `mixer_fillA` and `mixer_fillB` phase or when the system has arrived in the `mixer_agitate` phase. Which one that is chosen is unimportant. The approach is illustrated in Figure 9.20.

The supervisor enables or disables the events "OKT1", "OKT2", "OKT3" and "OKT4", "OKT5", "OKT6". These events make it possible or impossible for the Grafchart controller to allocate a Mixer, Reactor$_{heat}$ or Reactor$_{cool}$

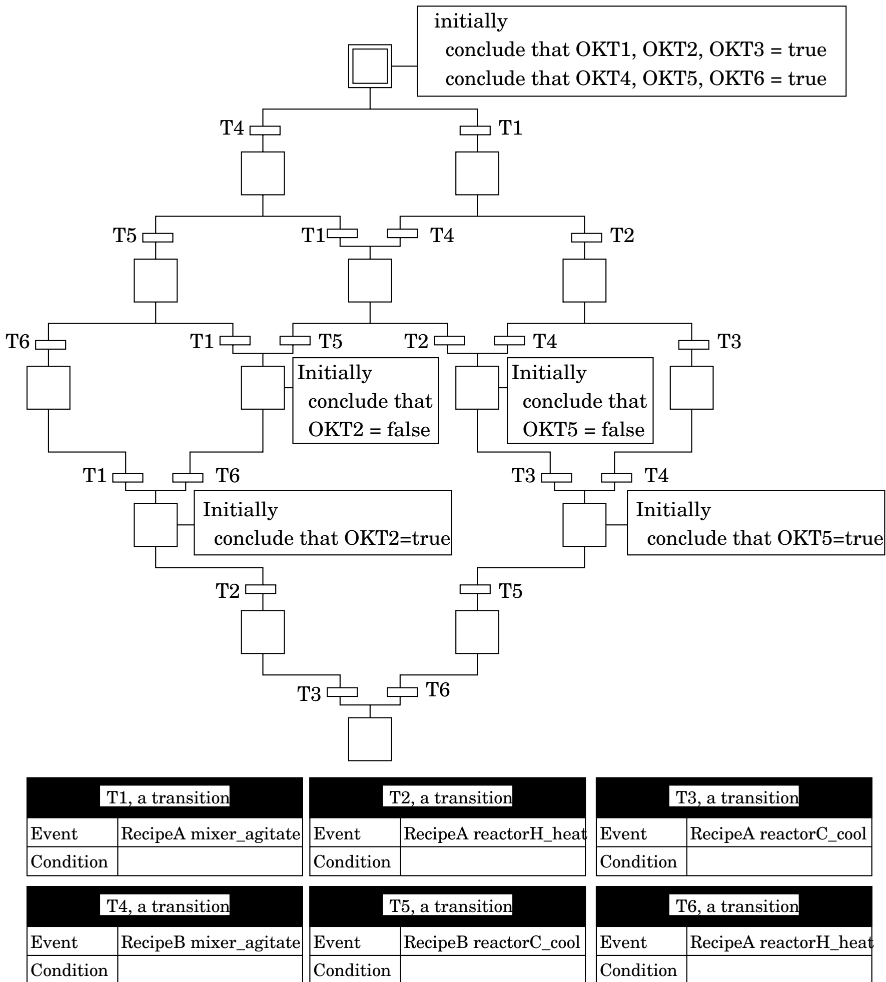


Figure 9.19 A supervisor for the case in Figure 9.14.

for recipe-A and a Mixer, Reactor_{cool} or Reactor_{heat} for recipe-B respectively. In order to take the events generated by the supervisor into account, the two recipes must be modified. The unmodified versions of Recipe-A and recipe-B are shown in Figure 9.2 and the modified versions are shown in Figure 9.21. As can be seen it is only certain transition receptivities that have been modified. The modification consists of an additional condition on "OKTx" in the receptivity. If "OKTx" is true, the enabled transition can fire, but if "OKTx" is false it cannot fire.

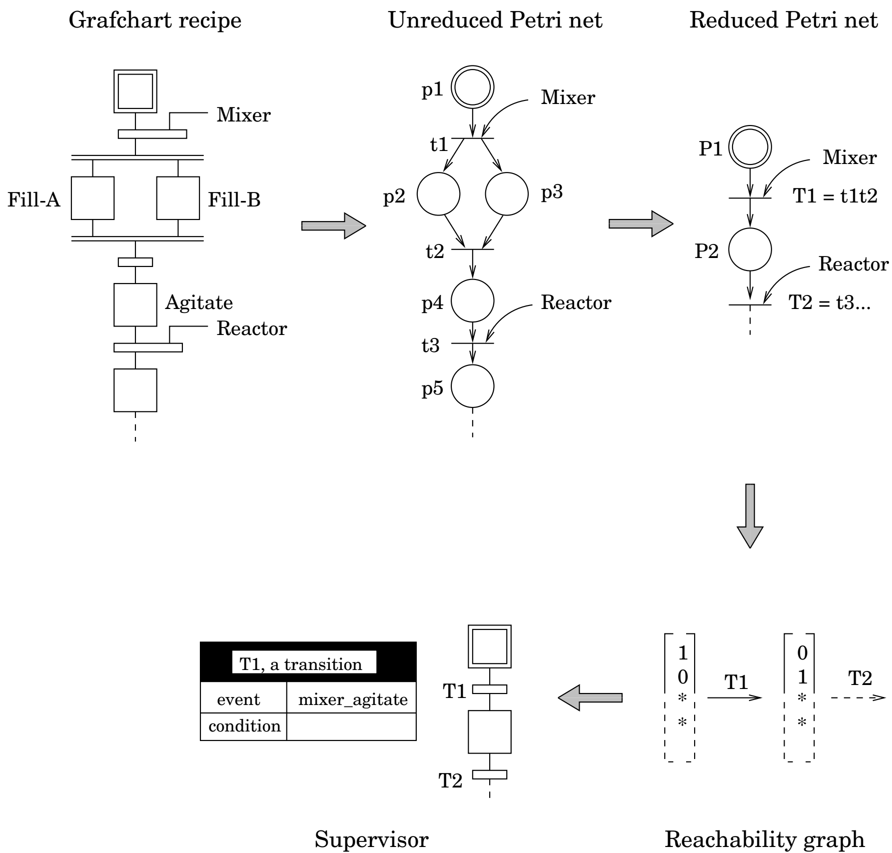


Figure 9.20 The train of thought for finding out the transition receptivities in a supervisor.

Depending on the number of batches in the plant, the configuration of the plant, or the requirements on the production of the batches, the structure of the reachability graph is changed and thus the transitions that should be disabled varies. This means that the structure of the supervisor is modified. However, the two recipes, recipe-A and recipe-B, see Figure 9.21, need not to be further modified depending on the structure of the supervisor.

The forbidden state method can be used to find the states that should be avoided and the transitions that should be disabled in order to prevent a deadlock from occuring. It can, however, also be used to assure other scenarios, e.g., that a unit is always available for allocation after a certain

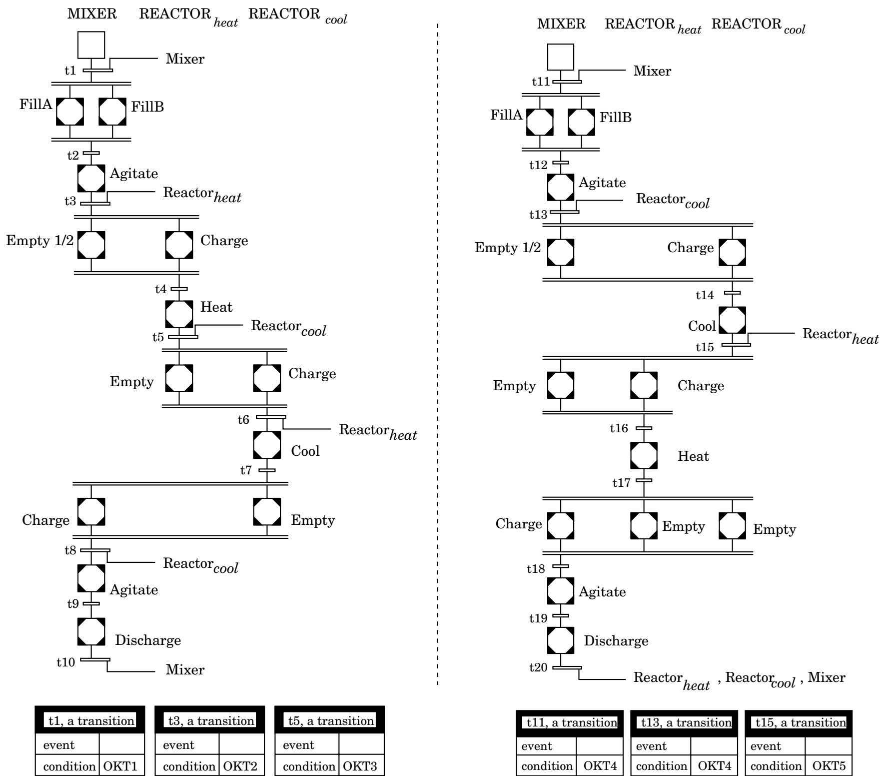


Figure 9.21 Two Grafchart recipe structures susceptible for events generated by a supervisor: recipe-A (left) and recipe-B (right).

operation. Such an example was presented in the Chapter 9.5 (Dynamical Deadlock Avoidance) and the corresponding reachability graph can be seen in Figure 9.15 (right). The supervisor for this case is shown is Figure 9.22.

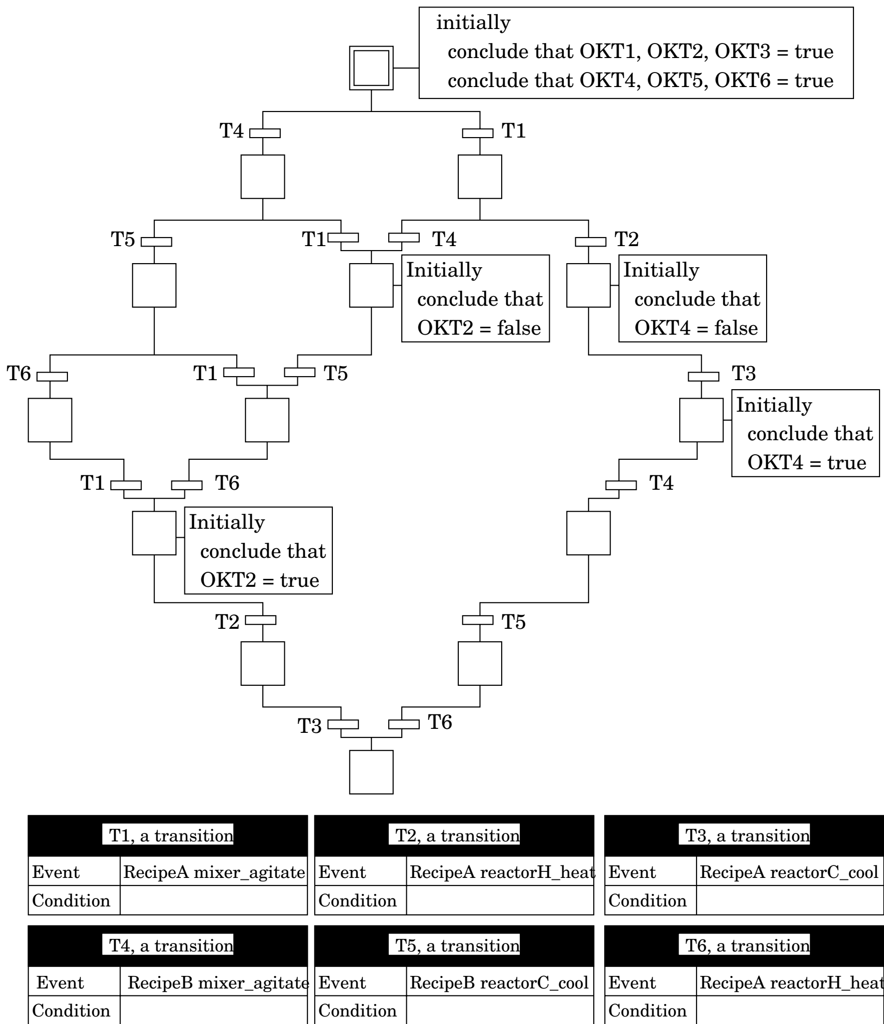


Figure 9.22 A supervisor for the case in Figure 9.15.

Recipe State Based Supervisor

The size of the supervisors shown in the last subsection can easily become very large. If the size gets too large it becomes impossible to implement them. It is therefore desirable to reduce the size of the supervisors.

Consider once again the plant consisting of two mixers, one reactor$_{heat}$ and one reactor$_{cool}$. In the plant two batches, one of type A and one of type B,

should be produced. The reachability graph for this system is shown in Figure 9.15. In order for the supervisor to disable and enable the transitions correctly, the supervisor must know when the system arrives in or leaves from certain states. These states only constitute a subset of the total number of states in the reachability graph. For the system under consideration, the important states are $[00100100001]^T$ and $[01000010010]^T$. In Figure 9.23, these states are marked out and given names.

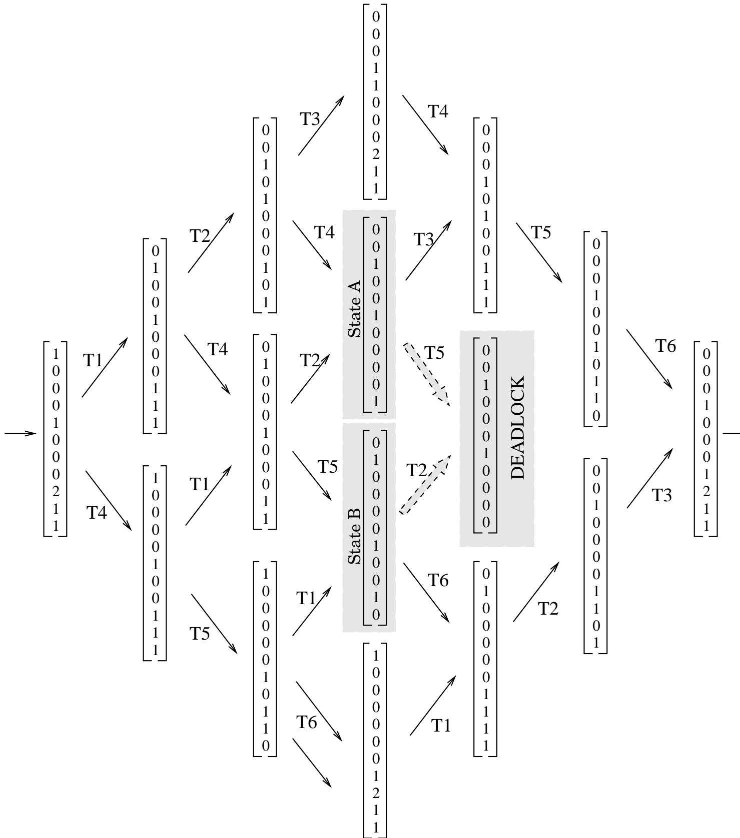


Figure 9.23 A supervisor for the case in Figure 9.15.

The reachability graph based supervisor, see Figure 9.19, contained fifteen states. The structure of the supervisor can be reduced so that it contains only four states. The reduced supervisor is shown in Figure 9.24.

In this supervisor, it is assumed that the two states named `stateA` and `stateB` can be detected. By using the same type of reasoning as presented

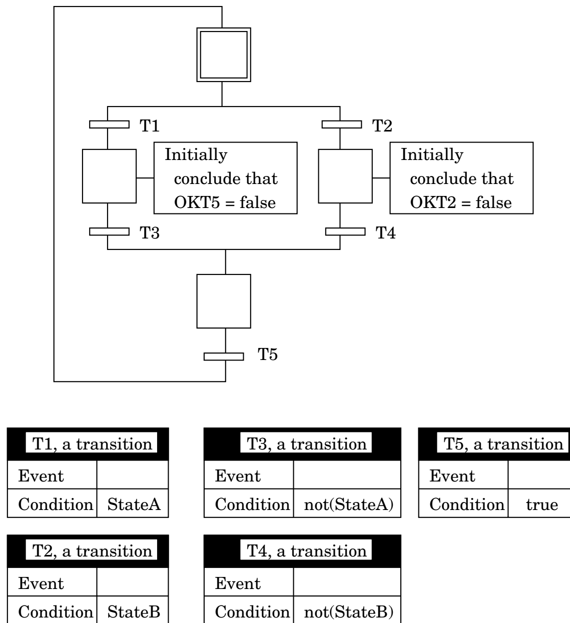


Figure 9.24 A reduced supervisor for the case in Figure 9.23.

in Figure 9.20 the real states that correspond to the two states in the reachability graph can be deduced, see the following table:

	Reachability graph state	Real state
stateA	$[0010 **** ***]^T$	((RecipeA mixer_empty AND recipeA $reactor_{heat}$_charge) OR recipeA $reactor_{heat}$_heat)
	$[**** 0100 ***]^T$	((RecipeB mixer_fillA AND recipeB mixer_fillB) OR recipeB mixer_agitate)
stateB	$[0100 **** ***]^T$	((RecipeA mixer_fillA AND recipeA mixer_fillB) OR recipeA mixer_agitate)
	$[**** 0010 ***]^T$	((RecipeB mixer_empty AND recipeB $reactor_{cool}$_charge) OR recipeB $reactor_{cool}$_cool)

StateA corresponds to the six upper rows in the previous table and stateB corresponds to the six lower rows. The receptivities in the reduced supervisor can now be rewritten using these conditions. The receptivity of transition T1 is shown in Figure 9.25.

T1, a transition	
Event	
Condition	((recipeA mixer_empty AND recipeA $reactor_{heat}$ _charge) OR recipeA $reactor_{heat}$ _heat) AND ((recipeB mixer_fillA AND recipeB mixer_fillB) OR recipeB mixer_agitate)

Figure 9.25 Receptivity of transition T1 in the reduced supervisor shown in Figure 9.24.

As can be seen the receptivities of the reduced supervisor becomes more complex than the receptivities of the "reachability graph based supervisory". However, the size of the supervisor is drastically reduced.

In Chapter 9.5 (Static Deadlock Prevention), a table was shown indicating the "risk-of-deadlocks" for different combinations of batches and plant configurations, see Figure 9.13. The most intricate combination is the one corresponding to the first column and the upper row in the table, i.e., 2 type A batches and two type B batches produced in a plant that consists of three mixers, 1 $reactor_{heat}$ and two $reactor_{cool}$. The reachability graph for this combination has 85 states. Constructing a supervisor that resembles the reachability graph would result in a large supervisor. However, a closer look at the reachability graph shows that there are only three states that are of interest, and the size of the supervisor can be reduced to five states.

9.7 Other Analysis Approaches

The analysis method presented earlier in this chapter is based on Petri nets. There do also exist other analysis methods that might be of interest also for batch recipe applications.

Banker's Algorithm

An algorithm commonly used for resource allocation in real-time systems is Banker's algorithm. The algorithm is used for deadlock-avoidance, [Silberschatz and Galvin, 1995]. When a new process enters the system it must declare the maximum number of instances of each resource type that

it might need. This number may not exceed the total number of resources in the system. When a process requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If so, the resources are allocated, otherwise, the process must wait until some other process releases enough resources.

Ideas from this algorithm might be useful for a deadlock-detection system within a recipe management system.

Modular Control by Tittus et al.

The two recipes, recipe-A and recipe-B, presented in this chapter run in the same plant. The plant configuration is given in Figure 9.1. Between the units valve batteries are placed, see Figure 7.10. Thanks to the matrix structure of the valve batteries there is a separate connection line, i.e., set of valves, pumps and pipes, between every two units. A connection line is reserved only when the two units it connects are reserved. This means that there will never be any problems allocating a connection line, neither must a connection line be left unreserved in order for another connection line to work properly. In a plant with such a generous connection line structure, a deadlock situation can never occur due to the connection lines and it is therefore enough to take only the units into account when doing deadlock analysis. However, this is not the case in all batch plants. Plants exist where both units and connection lines have to be reserved and unreserved while running the batch. This increases the complexity of the analysis methods. The methods developed by Tittus et al., [Åkesson and Tittus, 1998] and [Tittus, 1995] take this constraint into consideration.

The methods developed by Tittus et al. are based on Petri net theory. The resources in a plant are divided into two generic classes; processes, e.g., reactors and pumps, and transport devices, e.g., pumps and valves. A connection line is defined as the set of required transport devices in order to connect a source processor with a target processor. When the batch should be moved from one processor (unit) to another processor (unit), the two processors (units) as well as the connection line must be reserved. The task of the supervisor is to prevent deadlock situations from occurring. Deadlocks can occur when allocating processors or when allocating transport devices. The two problems are independent from each other which makes it possible to split the supervisor into two modules, one which restricts the booking of processors ($\mathcal{S}$-module), and one which restricts the booking of transport devices ($\mathcal{T}$-module), see Figure 9.26.

In the method developed by Tittus et al. each batch has a corresponding device dependent recipe which contains all possible execution path through the plant. The recipe is modeled by a Petri net where each operation is modeled by one place, and the transitions correspond to booking

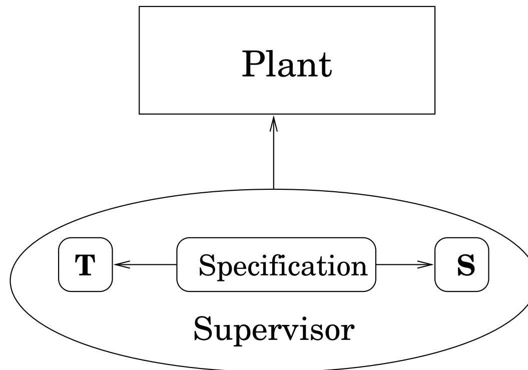


Figure 9.26 Supervisor structure according to Tittus et al..

or unbooking of processors or connection lines. The different recipes are interleaved with each other, i.e., they are run independently of each other. A specification is defined as all possible interleavings of the recipes. In addition to this each processor and transport device is modeled by a separate Petri net. The states indicate if the processor/transport device is unbooked or booked, i.e., if it is available or unavailable for a batch.

The $\mathcal{T}$-module removes the circular wait between resources in the transporting system by ordering all transport devices and guaranteeing that all transport devices in a connection line are booked in this order. The booking order is specified by a Petri net. This Petri net is synchronized with the recipes, i.e., transitions with the same label acts as if they were the same transition (i.e. they all fire simultaneously, but only if the input places in all Petri nets can be fired). By assuring that no circular wait can occur, the $\mathcal{T}$-module is deadlock free, and a deadlock can therefore not occur within the transporting system.

The $\mathcal{S}$-module is constructed in the following way. A Petri net describing the plant is constructed. All different recipes are combined into one large Petri net. The plant Petri net and the large recipe Petri net are synchronized, i.e., transitions with the same label are assumed to be the same transition. This results in a very large Petri net expressing all physically possible ways of executing the different operations in the plant. This Petri is now translated into automata representation. A discrete supervisor that coordinates and controls the execution of the different recipes is algorithmically synthesized (exhaustive search).

9.8 Resource Allocation and Scheduling

The methods presented in this chapter only serve to prevent resource allocation if it can lead to a deadlock situation. If there are several paths through the plant, it is not the task of these methods to choose the optimal one among them. The task of optimization, most often with respect to time, is instead assured by a scheduling algorithm, see Chapter 6.4. As was shown in Chapter 6.4 scheduling is a wide concept that can be broken down hierarchically, according to Figure 6.5. The bottom-most level in the scheduling hierarchy deals with resource allocation and arbitration.

9.9 Summary

A recipe structured with Grafchart can be transformed into a corresponding Petri net to which already existing Petri net analysis methods can be applied. Using, e.g., the reduction techniques and the reachability graphs, characteristics like deadlock-freeness can be investigated. As shown in this chapter it is only the topology of the Grafchart structured recipe that can be analyzed, influences of step actions or transition receptivities cannot be included in the analysis. The results of the deadlock analysis can be used in different ways. Static reasoning deals with the question of which combinations of number of batches and number of units that can be safely started in the plant at the same time. The answer to this question can, e.g., be stored in a table. Dynamic reasoning is about determining, depending on the status of the plant, which units that are unsafe to allocate and therefore not should be reserved by a batch, i.e., to find the unsafe states. To assure that the system never reaches an unsafe state a supervisor can be used. In the chapter it is shown how the supervisor can be constructed from the reachability graph and how the size of the supervisor can be reduced.

10

Conclusions

In this thesis a graphical language for sequential control is presented and its application to batch control is examined and discussed.

Graphical programming languages have many advantages compared to textual programming languages. They often allow programming in a style that closely mimics the style that people model problems. Graphical languages fit users' existing perception of a problem, making the application easier to build, debug, document, and maintain. An added benefit is the possibility to use color and animation to provide feedback as the program executes. Graphical programming languages have always been popular in the automatic control community, e.g., in the form of relay diagrams or function block diagrams. Sequential logic is commonly represented by SFC/Grafcet, a graphical language that has its roots in automata theory and Petri nets. The state machine and Petri net formalisms are also commonly used in the context of formal methods for analysis of discrete event systems.

Grafchart is the name of a graphical language for sequential control applications. The main aim of Grafchart is to show how SFC/Grafcet can be modified from a rather low-level graphical language into a high-level, object-oriented graphical language. Two versions of Grafchart exist; the basic version that is strongly related to SFC/Grafcet, and the high-level version that has more in common with high-level Petri nets. The main new concepts in the basic version of Grafchart are parameterization, methods and message passing and exception handling facilities. These concepts are common in textual languages, but not so often found in graphical languages. The features increase the possibilities to reuse programs from one application to another. In the high-level version the new features are the support for object tokens and multi-dimensional charts. These features have been inspired by concepts found in the object-oriented programming and high-level Petri net area. The features greatly increase the abstraction and structuring possibilities of the language. This makes it possible to

use Grafchart not only for simple local level control applications, but also for more demanding applications on the supervisory control level. Since Grafchart is based on Grafcet and Petri nets, it is possible to transform a Grafchart program into a Petri net and/or a Grafcet. Hence, it is possible to apply the formal analysis methods developed for Petri nets and Grafcet also to Grafchart.

Grafchart is a general-purpose language that has been used in several different application. The topic of this thesis is recipe-based batch control. In the thesis it has been shown how Grafchart can be used for recipe structuring and recipe analysis. By using the features of Grafchart in various ways, recipes can be given different structures with different advantages and disadvantages. All structures comply with the international batch standard IEC 61512-1 (also refered to as ISA S88.01). To test and evaluate the different recipe structures a real-time simulation model of a batch cell was used. The different features introduced in Grafchart have many advantages in the context of recipe-based batch control.

- The parameterization feature makes it easy to reuse existing recipe structures and facilitates modifications of already existing recipe structures.
- By utilizing the method and message feature, the linking between the procedural elements of the control recipe and the equipment, is easily represented and implemented.
- The object token and the multi-dimensional chart features make it possible to have a compact, but yet clear, description and visualization of the recipe execution and of the overall status in the plant.
- Batch recipes and resource allocation can be combined. The combined Grafchart network can be translated into an equivalent Petri net and analyzed in order to look for possible deadlock situations. The results can be used for static deadlock prevention or dynamic deadlock avoidance.

Several commercial batch control systems exist today. They all have a similar approach to recipe structuring, resource allocation and user presentation. The advantages of using Grafchart compared to some of these systems, regarding these features are presented in the thesis. Grafchart allows, e.g., the operator to have a graphical view of the overall status in the plant as well as of the resource allocation. Analysis of the recipes can be performed in order to look for possible deadlock situations caused by improper resource allocation. By using Grafchart it is also possible to implement all layers in the recipe procedure using the same language.

Grafchart is currently implemented in G2, an object-oriented real-time programming environment with a strong graphical appeal. Since G2 is used in industry the step towards industrial application of the results presented in this thesis is relatively small. However, the Grafchart-based batch control system lacks several important features that are needed in a real system, e.g., data logging and report generation. Hence, it should primarily be seen as a source of inspiration for the developer of future batch control systems. An example of this is that the results presented in this thesis has attracted a lot of interest from the ISA S88.01 standardization committee.

10.1 Future Research Directions

The current work can be continued in several directions. They can be divided into topics that concern the Grafchart model and implementation, and topics that concern the batch control application.

Implementation Improvements

The G2 implementation of Grafchart can still be viewed as a prototype. Several modifications are possible. It would be relatively easy to change the local interpretation algorithm into a global algorithm. With this modification the semantics of Grafchart would become closer to the Grafcet semantics. The syntax for step actions and receptivities needs improvements. New language elements could be considered. The main hierarchical construct of Grafchart, the macro step, is very procedural in nature. A macro step may only have one entry point, i.e., one enter step. A result of this is that it becomes cumbersome to represent truly hierarchical steps with multiple entry points. This is often practical to have when Grafchart is used for representing state machines. A possibility would be to define a new super step that may have multiple entry and exit points, similar to the super state of Statecharts.

Alternative Implementation Platforms

G2 has been a very convenient implementation platform for Grafchart. The main features that have been utilized are the object-orientation and the graphical support. However, G2 is still not widely spread. An alternative implementation platform that shares many of the advantages of G2 is Java. Java is object-oriented and has good graphics support. Furthermore, Java is platform-independent and supports distributed execution. The current strong trend towards using Java also in process automation makes it a very strong candidate as a new platform for Grafchart.

Batch Scheduling

The current recipe management system performs dynamic resource allocation. By applying analysis methods one can find out when and why deadlock situations can occur. Statical deadlock prevention or dynamical deadlock avoidance can be used in order not to get caught in an undesired deadlock situation. However, the task of allocating the resources in an optimal way, most often with respect to time, is not considered. This task should instead be assured by a scheduling algorithm.

Monitoring of Recipe-Based Batch Production

Monitoring of recipe-based batch control is an interesting area. In recipe-based batch control, the monitoring consists of monitoring of the equipment, similar to the situation in continuous processes, and monitoring of the recipe execution. The current recipe execution system contains no guarantees that ensure that a recipe is well-formulated in the sense that the sequence of operations and phases in the recipe is physically possible, considering the capabilities of the equipment units. For example, a recipe might start with a discharge phase, or perform a heating phase in a reactor before the reactor has been charged. This problem can be attacked in two ways. One possibility is to check the recipes off-line. This is perhaps the most realistic alternative. However, alternatively or in addition, one could also have an on-line check that tests that the equipment is in an allowed state before a new recipe phase is started. Also, the current recipe execution system contains no equipment monitoring. Faults may occur in an equipment unit both when it is used by a batch and when it is free. The equipment monitoring is normally taken care of by safety and alarm logic associated with the equipment. A fault can be seen as a change in state of the equipment. The safety logic responsible for detecting the fault can be seen as a conditional test that initiates a state change from a normal state to a faulty state.

The above forms of on-line monitoring can be combined if an internal model approach is taken. Each equipment unit contains a state machine that models the equipment state. The safety logic can be represented as transition conditions in this state machine. The state machine may be a single automaton or decomposed into several smaller automata. Associated with each equipment phase is a set of allowed pre-condition states. A phase may only be started if the current active state belongs to the precondition states of the phase. Executing a phase corresponds to moving from a initial state, through a number of intermediate states, to a final state. Grafchart, possibly extended with the super step construct, could be used also for representing the equipment state machine. This would provide an interesting combination where Grafchart is used both

to represent the recipe procedures, the equipment control logic, and the equipment monitoring system.

11

Bibliography

AGAOUA, S. (1987): *Spécification et commande des systèmes à événements discrets, le Grafcet coloré*. PhD thesis, Institut National polytechnique de Grenoble.

ÅKESSON, K. and M. TITTUS (1998): "Modular control for avoiding deadlock in batch processes." In *WBF'98 — World Batch Forum*.

ALSOP, N., L. CAMILLOCCI, A. SANCHEZ, and S. MACCHIETTO (1996): "Synthesis of procedural controllers - applications to a batch pilot plant." *Computers and Chemical Engineering*, **20:972**.

ARC (1996): "Batch process automation strategies." Industrial Automation Strategies for Executives. Automation Research Corporation, Memorandum.

ÅRZÉN, K.-E. (1991): "Sequential function charts for knowledge-based, real-time applications." In *Proc. Third IFAC Workshop on AI in Real-Time Control*. Rohnert Park, California.

ÅRZÉN, K.-E. (1993): "Grafcet for intelligent real-time systems." In *Preprints IFAC 12th World Congress*. Sydney, Australia.

ÅRZÉN, K.-E. (1994a): "Grafcet for intelligent supervisory control applications." *Automatica*, **30:10**, pp. 1513 – 1525.

ÅRZÉN, K.-E. (1994b): "Parameterized high-level Grafcet for structuring real-time KBS applications." In *Preprints of the 2nd IFAC Workshop on Computer Software Structures Integrating AI/KBS in Process Control Systems, Lund, Sweden.*

ÅRZÉN, K.-E. (1996): "A sequential function chart based approach to alarm filtering." In *Preprints 13th IFAC World Congress*, vol. M, pp. 235 – 241.

ÅRZÉN, K.-E. and C. JOHNSSON (1996a): "Object-oriented SFC and ISA-S88.01 recipes." In *WBF'96—World Batch Forum*. Toronto, Canada.

ÅRZÉN, K.-E. and C. JOHNSSON (1996b): "Object-oriented SFC and ISA-S88.01 recipes." *ISA Transactions*, **35**, pp. 237–244.

ÅRZÉN, K.-E. and C. JOHNSSON (1997): "Grafchart: a Petri net/Grafcet based graphical language for real-time sequential control applications." In *SNART'97—The Swedish National Real-Time Systems Conference, Lund.*

BASTIDE, R., C. SIBERTIN-BLANC, and P. PALANQUE (1993): "Cooperative Objects: A concurrent, Petri-net based, object-oriented language." In *Proceeding of IEEE Conference on Systems, Man and Cybernetics*. Le Touquet, France.

BATTISTON, E., F. DE CINDIO, and G. MAURI (1988): "OBJSA: A class of high level nets having objects as domains." In ROZENBERG, Ed., *Advances in Petri Nets 1988*, vol. 340 of *Lecture notes in computer science*, pp. 20–43. Springer Verlag, Berlin Heidelberg, New York.

BERTHELOT, G. (1983): *Transformation et analyse de réseaux de Petri: Application aux protocoles*. PhD thesis, Nice University.

BERTHELOT, G. (1986): "Checking properties of nets using transformations." In *Lecture notes in Computer Science*, vol. 222, pp. 19 – 40. Springer Verlag.

BRETTSCHMEIDER, H., H. GENRICH, and H. HANISCH (1996): "Verification and performance analysis of recipe based controllers by means of dynamic plant models." In *Second International Conference on Computer Integrated Manufacturing in the Process Industries*. Eindhoven, The Netherlands.

BURNS, A. and A. WELLINGS (1996): *Real-time systems and programming languages*. Addison-Wesley.

CASSANDRAS, C. (1993): *Discrete Event Systems: Modeling and Performance Analysis*. Irwin.

CHARBONNIER, F. (1996): *Commande supervisée des systèmes à événements discrets*. PhD thesis, Institut National Polytechnique de Grenoble.

CHARBONNIER, F., H. ALLA, and R. DAVID (1995): "The supervised control of discrete event dynamic systems: A new approach." In *34th Conference on Decision and Control*. New Orleans.

DAVID, R. (1995): "Grafcet: A powerful tool for specification of logic controllers." *IEEE Transactions on Control Systems Technology*, **3:3**, pp. 253–268.

DAVID, R. and H. ALLA (1992): *Petri nets and Grafcet: Tools for modelling discrete events systems*. Prentice-Hall International (UK) Ltd.

DE LOOR, P., J. ZAYTOON, and G. VILLERMAIN-LECOLIER (1997): "Abstractions and heuristics for the validation of Grafcet controlled system." *JESA – European Journal of Automation*, **31:3**, pp. 561–580.

DESROCHERS, A. A. and R. AL'JAAR (1995): "Applications of Petri nets in manufacturing systems: Modelling, control and performance analysis." *IEEE Press*.

DIJKSTRA, E. (1968): "Cooperating sequential processes." In GENUYS, Ed., *Programming languages*. Academic Press N.Y.

ENGELL, S. and K. WÖLLHAF (1994): "Dynamic simulation for improved operation of flexible batch plants." In *Proc. CIMPRO 94*, pp. 441–455. Rutgers University, New Jersey, USA.

FISHER, T. G. (1990): *Batch Control Systems: Design, Application, and Implementation*. Instrument Society of America, Research Park, NC.

FLEISCHHACK, H. and U. LICHTBLAU (1993): "MOBY - A tool for high level Petri nets with objects." In *Proceeding of IEEE Conference on Systems, Man and Cybernetics*. Le Touquet, France.

GAFFE, D. (1996): *Le modèle Grafcet: Réflextion et intégration dans une plate-forme multiformalisme synchrone*. PhD thesis, Université de Nice-Sophia Antipolis.

GENRICH, H. J., H.-M. HANISCH, and K. WÖLLHAF (1994): "Verification of recipe-based control procedures by means of predicate/transition nets." In *15th International Conference on Application and Theory of Petri nets, Zaragoza, Spain*.

GENSYM COORPORATION (1995): *G2 Reference Manual, Version 4.0*. Gensym Coorporation, 125 Cambridge Park Drive, Cambridge, MA 02140, USA.

GUNNARSSON, J. (1997): *Symbolic Methods and Tools for Discrete Event Dynamic Systems*. PhD thesis, Linköping University.

HANISCH, H.-M. and S. FLECK (1996): "A resource allocation scheme for flexible batch plants based on high-level Petri nets." In *IEEE SMC, CESA96*. Lille, France.

HARPER, R. (1986): "Introduction to standard ML." Technical Report. Dept. of Computer Science, University of Edinburgh. ECS-LFCS-86-14.

HO, Y. (1991): *Discrete Event Dynamic Systems: Analyzing Complexity and Performance in the Modern World*. IEEE Press, New York.

HOLLOWAY, L. and B. KROGH (1990): "Synthesis of feedback control logic for a class of controlled Petri nets." *IEEE Transactions on Automatic Control*, **35:5**, pp. 514–523.

HOLLOWAY, L. and B. KROGH (1994): "Controlled Petri nets: A tutorial survey." In *11th International Conference on Analysis and Optimization of Systems - Discrete Event Systems*. Springer-Verlag. number 199 in Lecture Notes in Control and Information Science, pages 158-168, DES94, Ecole des Mines de Paris, INRIA.

IEC (1988): "IEC 60848: Preparation of function charts for control systems." Technical Report. International Electrotechnical Commissio.

IEC (1993): "IEC 61131 programmable controllers - part 3: Programming languages." Technical Report. International Electrotechnical Commission.

IEC (1997): "IEC 61512-1: Batch control, part1, models and termonology." Technical Report. International Electrotechnical Commission.

ISA (1995): "ISA S88.01 batch control." Instrument Society of America.

JENSEN, K. (1981): "Coloured Petri Nets and the invariant method." *Theoretical Computer Science, North-Holland*, **14**, pp. 317–336.

JENSEN, K. (1990): "Coloured Petri nets: A high level language for system design and analysis." In ROZENBERG, Ed., *Advances in Petri Nets 1990*, vol. 483 of *Lecture notes in Computer Science*, pp. 342–416. Springer Verlag, Berlin Heidelberg, New York.

JENSEN, K. (1992): *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.*, vol. 1, Basic Concepts. Springer-Verlag.

JENSEN, K. (1995): *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.*, vol. 2, Analysis Methods. Springer-Verlag.

JENSEN, K. (1997): *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.*, vol. 3, Practical Use. Springer-Verlag.

JENSEN, K. and G. ROZENBERG (1991): *High-level Petri Nets*. Springer Verlag.

JOHNSSON, C. (1997): "Recipe-Based Batch Control Using High-Level Grafchart." Licentiate thesis ISRN LUTFD2/TFRT--3217--SE. Dept. of Automatic Control, Sweden. Available at http:://www.control.lth.se/~lotta/papers.html.

JOHNSSON, C. and K.-E. ÅRZÉN (1994): "High-level Grafcet and batch control." In *Symposium ADPM'94—Automation of Mixed Processes: Dynamical Hybrid Systems*. Brussels, Belgium.

JOHNSSON, C. and K.-E. ÅRZÉN (1996a): "Batch recipe structuring using high-level Grafchart." In *IFAC'96, Preprints 13th World Congress of IFAC*. San Francisco, California.

JOHNSSON, C. and K.-E. ÅRZÉN (1996b): "Object tokens in high-level Grafchart." In *CIMAT'96—Computer Integrated Manufacturing and Automation Technology*. Grenoble, France.

JOHNSSON, C. and K.-E. ÅRZÉN (1998a): "Grafchart and batch recipe structures." In *WBF'98 — World Batch Forum*. Baltimore, ML, USA.

JOHNSSON, C. and K.-E. ÅRZÉN (1998b): "Grafchart and its relations to Grafcet and Petri nets." In *Symposium INCOM'98 — Information Control Problems in Manufacturing*. Nancy, France.

JOHNSSON, C. and K.-E. ÅRZÉN (1998c): "Grafchart applications." In *GUS'98 — Gensym User Society meeting*. Newport, RI, USA.

JOHNSSON, C. and K.-E. ÅRZÉN (1998d): "Grafchart for recipe-based batch control." *Computers and Chemical Engineering*, **22:12**, pp. 1811 – 1828.

JOHNSSON, C. and K.-E. ÅRZÉN (1998e): "On batch recipe structures using High-Level Grafchart." In *Symposium ADPM'98 — Automation of Mixed Processes: Dynamical Hybrid Systems*. Reims, France.

JOHNSSON, C. and K.-E. ÅRZÉN (1998f): "On batch recipe structuring and analysis using Grafchart." *European Journal of Automation*. To appear.

JOHNSSON, C. and K.-E. ÅRZÉN (1998g): "Petri net analysis of batch recipes structured with Grafchart." In *FOCAPO'98—Foundations of Computer Aided Process Operations*.

JOHNSSON, C. and K.-E. ÅRZÉN (1999a): "Grafchart and batch recipe structures." In *Interphex'99*. New York, NY, USA.

JOHNSSON, C. and K.-E. ÅRZÉN (1999b): "Grafchart and Grafcet: A comparison between two graphical languages aimed for sequential control applications." In *Ifac'99*. Beijing, China.

KLUGE, W. and K. LAUTENBACH (1982): "The orderly resolution of memory access conflicts among competing channel processes." *IEEE Trans. Comp.*, **C-31:2**, pp. 194–207.

LAKOS, C. (1994): "Object Petri nets - definitions and relationship to coloured nets." Technical Report R94-3. Dept. of Computer Science, University of Tasmania.

LAKOS, C. and C. KEEN (1994): "Loopn++: A new language for object-oriented Petri nets." Technical Report R94-4. Dept. of Computer Science, University of Tasmania.

LÓPEZ GONZÁLEZ, J. M., J. I. LLORENTE GONZÁLEZ, J. M. SANTAMARÍA YUGUEROS, O. PEREDA MARTÍNEZ, and E. ALVAREZ DE LOS MOZOS (1994): "Graphical methods for flexible machining cell control using G2." In *Proc. of the Gensym European User Society Meeting, Edinburgh, October.*

MEALY, G. (1955): "A method for synthesizing sequential circuits." *Bell System Technical Journal*, **5:34**, pp. 1045–1079.

META SOFTWARE CORPORATION, Cambridge, MA,USA (1993): *Design/CPN Tutorial for X-Windows.* version 2.0.

MOALLA, M. (1985): "Réseaux de Petri interprétés et Grafcet." *Technique et Science Informatique*, **14:1**.

MOLLOY, M. (1982): "Performance analysis using stochastic Petri nets." *IEEE Trans. Computers*, **C-31:9**, pp. 913–917.

MOORE, E. (1956): "Gedanken experiments on sequential machines." *Automata Studies*, pp. 129–153. Princeton University Press.

MURATA (1989): "Petri nets: Properties, analysis and applications." *Proceedings of the IEEE*, **77:4**, pp. 541 – 580.

MURATA, T. and J. KOH (1980): "Reduction and expansion of live and safe marked graphs." *IEEE Transactions on Circuit Systems*, **27:1**, pp. 68 – 70.

MURATA, T., N. KOMODA, K. MATSUMOTO, and K. HARUNA (1986): "A Petri net-based controller for flexible and maintainable sequence control and its application in factory automation." *IEEE Transaction Ind. Electron.*, **33:1**, pp. 1–8.

NAMUR (1992): *NAMUR-Emphehlung: Anforderungen an Systeme zur Rezeptfaheweise (Requirements for Batch Control Systems). NAMUR AK 2.3 Funktionen der Betriebs- und Produktionsleitebene.*

NILSSON, B. (1991): "En on-linesimulator för operatörsstöd," (An on-line simulator for operator support). Report TFRT-3209. Department of Automatic Control, Lund Institute of Technology.

OZSU, M. (1985): "Modelling and analysis of distributed database concurrency control algorithms using an extended Petri net formalism." *IEEE Transaction Software Engineering*, **SE-11:10**, pp. 1225–1240.

PETERSON, J. (1981): *Petri net theory and the modeling of systems*. Prentice-hall.

PETRI, C. A. (1962): "Kommunikation mit automaten." Technical Report. Institut fur Instrumentelle Mathematik, Universitat Bonn. Schriften des IIM Nr. 3. Also in English translation, Communication with Automata, New York: Griffiss Air Force Base. Tech. Rep. RADC-TR-65-377, vol. 1, Suppl. 1, 1966.

RAMADGE, P. and W. WONHAM (1989): "The control of discrete event systems." In *Proceedings of the IEEE*, vol. 77, pp. 81–98.

ROSENHOF, H. P. and A. GHOSH (1987): *Batch Process Automation, Theory and Practice*. Van Nostrand Reinhold.

ROUSSEL, J.-M. and J.-J. LESAGE (1996): "Validation and verification of Grafcets using state machine." In *CESA'96 IMACS/IEEE Symposium on Discrete Events and Manufacturing Systems*, pp. 765–770. Lille, France.

SANCHEZ, A., G. ROTSTEIN, and S. MACCHIETTO (1995): "Synthesis of procedural controllers for batch chemical processes." In *Proc. of 4th IFAC Symposium on Dynamics and Control of Chemical Reactors, Distillation Columns and Batch Processes (DYCORD+'95), Denmark*.

SHAW, W. (1982): *Computer Control of Batch Processes*. EMC Controls Inc, Cockeysville, MD.

SIFAKIS, J. (1978): "Use of Petri nets for performance evaluation." *Acta cybernet.*, **4:2**, pp. 185–202.

SILBERSCHATZ, A. and P. GALVIN (1995): *Operating System concepts*. Addison-Wesly.

SILVA, M. and E. TERUEL (1996): "Petri nets for design and operation of manufacturing systems: An overview." In *First International Workshop on Manufacturing and Petri Nets, Osaka Japan, International Conferences on Application and Theory of Petri Nets (ICATPN96)*. Universidad de Zaragoza, Spain.

SILVA, M. and S. VELILLA (1982): "Programmable logic controllers and Petri nets: A comparative study." In *IFAC Software for Computer Control, Madrid, Spain*, pp. 83–88. Ferrate, G. and Puente, E.A., Pergamont Press, Oxford England.

SIMENSEN, J., C. JOHNSSON, and K.-E. ÅRZÉN (1996): "A framework for batch plant information models." Report ISRN LUTFD2/TFRT--7553--SE. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

SIMENSEN, J., C. JOHNSSON, and K.-E. ÅRZÉN (1997a): “A multiple-view batch plant information model.” In *PSE’97/ESCAPE-7*. Trondheim, Norway.

SIMENSEN, J., C. JOHNSSON, and K.-E. ÅRZÉN (1997b): “A multiple-view batch plant information model.” *Computers and Chemical Engineering*, **21:1001**, pp. 1209 – 1214.

SONNENSCHEIN, M. (1993): “An introduction to Gina.” In *Proceding of IEEE Conference on Systems, Man and Cybernetics*. Le Touquet, France.

SUAU, D. (1989): *Grafcet coloré: Conception et realisation d’un outil de generation de simulation et de commande temps réel*. PhD thesis, Universite de Monpellier.

TITTUS, M. (1995): *Control Synthesis for Batch Processes*. PhD thesis, Chalmers University of Technology.

TURBAK, F., D. GIFFORD, and B. REISTAD (1995): “Applied semantics of programming languages.”. Unpublished Draft.

VALETTE, R., M. COURVOISIER, J. BEGOU, and J. ALBUKERQUE (1983): “Petri net based programmable logic controllers.” In *1st Int. IFIP conference: Comp. appl. in production and engineering*, pp. 103–116.

VISWANATHAN, S., C. JOHNSSON, R. SRINIVASAN, V. VENKATASUBRAMANIAN, and K.-E. ÅRZÉN (1998a): “Automating operating procedure synthesis for batch processes: Part 1. Knowledge representation and planning framework.” *Computers and Chemical Engineering*, **22:11**, pp. 1673 – 1685.

VISWANATHAN, S., C. JOHNSSON, R. SRINIVASAN, V. VENKATASUBRAMANIAN, and K.-E. ÅRZÉN (1998b): “Automating operating procedure synthesis for batch processes: Part 2. Implementation and application.” *Computers and Chemical Engineering*, **22:11**, pp. 1687 – 1698.

VISWANATHAN, S., V. VENKATASUBRAMANIAN, C. JOHNSSON, and K.-E. ÅRZÉN (1996): “Knowledge representation and planning strategies for batch plant operating procedure synthesis.” In *AIChE annual meeting, Chicago, USA*.

WILLIAMS, T. (1988): “A reference model for Computer Integrated Manufacturing (CIM).” In *International Purdue Workshop on Industrial Computer Systems, ISA*.

YAMALIDOU, E. C. and J. C. KANTOR (1991): “Modeling and optimal control of discrete-event chemical processes using Petri nets.” *Computers Chem. Engng*, **15**, pp. 503–519.

YOELI, M. (1987): “Specification and verification of asynchronous circuits using marked graphs.” In *Concurrency and Nets*, pp. 605–622.

A

Abbreviations

AFCET	Association Française pour la Cybernétique Economique et Technique
AFNOR	Association Française de NORmalisation
ANSI	American National Standards Institute
DCS	Distributed Control System
DEDS	Discrete Event Dynamic System
DES	Discrete Event System
ERP	Enterprise Resource Planning
GMP	Good Manufacturing Practive
IEC	International Electrotechnical Commission
ISA	Instrument Society of America
MES	Manufacturing Execution System
MRD	Material Resource Planning
OCS	Open Control Systems
PCT	Procedural Control Theory
PFC	Procedural Function Charts
PLC	Programmable Logic Controller
PN	Petri Nets
SCT	Supervisory Control Theory
SFC	Sequential Function Charts
SPC	Statistical Process Control

B

Petri Net Reduction Methods

Four reduction rules exist that preserve the properties: live, bounded, safe, deadlock-free, with home state and conservative, [Murata and Koh, 1980].

1. Reduction R_1: Substitution of a place
2. Reduction R_2: Implicit place
3. Reduction R_3: Neutral transition
4. Reduction R_4: Identical transitions

Two reduction methods exist that preserve the invariants, [Berthelot, 1986].

1. Reduction R_a: Self loop transitions
2. Reduction R_b: Pure transitions

A presentation of the reduction methods can be found in [David and Alla, 1992].

In this appendix, only the reduction methods R_1, R_2, R_3 and R_4 are presented. The reduction methods are presented for ordinary PN but they can be applied to generalized PN as well, assumed some adaptations are made. The properties have been proved by [Berthelot, 1983].

Reduction R_1*:* Substitution of a place
A place P_i can be substituted if it satisfies the following three conditions:

1. The output transitions of P_i have no other input places than P_i.
2. Place P_i is pure.
3. At least one output transition of P_i is not a sink transition.

Figure B.1 shows a simple case of place substitution. It is clear from the figure that if transition T_1 is fired then, sooner or later, transition T_2 will be fired and therefore place P_2 can be removed. If place P_2 is marked then this token should be put in place P_3 after substitution. If there are several output places of transition T_2, all the possible cases must be considered and several different nets must be studied from now on.

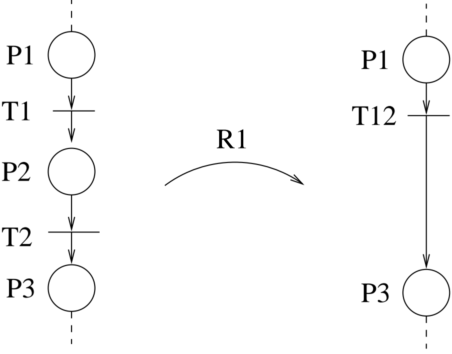


Figure B.1 Reduction R_1.

Reduction R_2: Implicit place
A place is implicit if it satisfies the following conditions.

1. The marking of this place never forms an obstacle to the firing of its output transitions.
2. Its marking can be deduced from the marking of the other places by the relation $M(P_i) = \sum_{k \neq i}(\alpha_k M(P_k)) + \beta$, where α_k is a rational number ≥ 0 and β is a rational number.

An example of reduction rule R_2 is shown in Figure B.2.

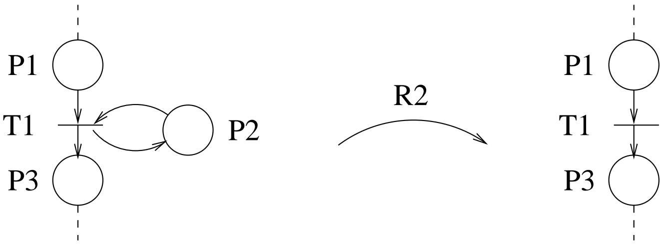


Figure B.2 Reduction R_2.

Reduction R_3: Neutral transition
A transition T_j is neutral if and only if the set of its input places is identical to the set of its output transitions, i.e., $^\circ T_j = T_j^\circ$. A neutral transition can be suppressed if and only if a transition $T_k \neq T_j$ exists such that $Post(P_i) \geq Pre(P_i, T_j)$ for every place $P_i \in {^\circ T_j}$. An example of reduction rule R_3 is shown in Figure B.3.

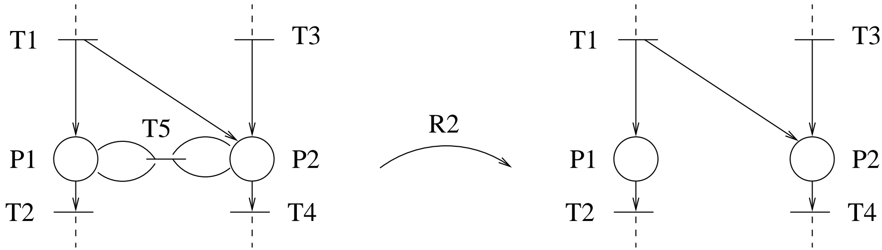


Figure B.3 Reduction R_3.

Reduction R_4: Identical transitions
Two transitions T_j and T_k are identical if they have the same set of input places and the same set of output places, i.e., $^\circ T_j = {^\circ T_j}$ and $T_j^\circ = T_k^\circ$. An example of reduction rule R_4 is shown in Figure B.4.

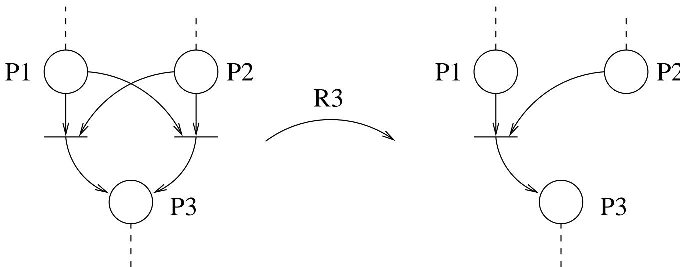


Figure B.4 Reduction R_4.

C

Formal Definition of Grafchart

A programming language is fully defined by its syntax, semantics and pragmatics. The syntax defines the form of the languages, the semantics defines the meaning of the languages and the pragmatics defines the implementation of the languages, [Turbak *et al.*, 1995]. If the pragmatics is omitted the programming language is reduced to a model.

In this chapter the syntax of Grafchart is formally presented.

- Syntax
  The syntax focuses on the notations used to encode a programming language. Consider a program, that indicates that the variable $C = 1$ should be increased by 2 if $x < 10$ and increased by 5 if $x \geq 10$. Such a program can be written in many different notations:

  - as a Pascal program

```
program Increase-C (C,x)
begin
var C= integer;
    x = real;
begin
  C:= 1;
  if x < 10 then
    C := C+2
  else
    C := C+5;
  end;
end
```

– or as a Grafchart function chart, see Figure C.1.

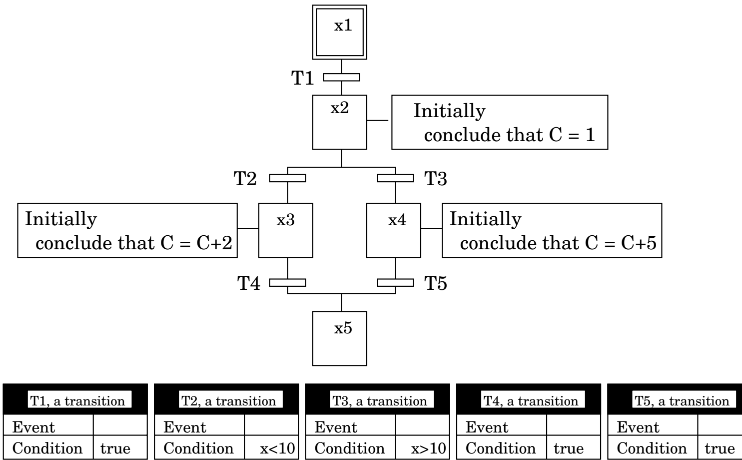


Figure C.1 A Grafchart function chart.

Although the notation used in the two programs are very different they are both a description of the abstract phrase: increase the variable $C = 1$ by 2 if $x < 10$ and by 5 if $x \geq 10$. The syntax describes the relationship between a notation and an abstract phrase.

If the language is a graphical language the notation is mainly graphical. The graphical elements together with the notation for step actions and transition receptivities constitute the syntax for Grafchart.

- Semantics
  The semantics specifies the mapping between the syntax of a programming language and what the syntax means. A programming language syntax has no meaning without an interpretation algorithm used for interpreting the syntax. Consider for example the Grafchart function chart shown in Figure C.1. The result of the program will be totally different depending on if e.g., the execution order is top-to-bottom or bottom-to-top, i.e., if step `x1` or step `x5` is considered the initialstep.

  The same programming language syntax can have many possible meanings. The semantics describes the relationship between the programming syntax and its meaning.

- Pragmatics
  The pragmatics focuses on how the meaning is computed, i.e, how is the syntax and semantics implemented.

This chapter contains a presentation of the formal definition of Grafchart. Section C.1 contains a presentation of notations that will be used later in the chapter. Grafchart exists in two versions: Grafchart-BV and Grafchart-HLV. However, Section C.2 introduces yet another version of Grafchart: Grafchart – *the Grafcet version*. This is a version of Grafchart that is very close to Grafcet, i.e., there is no support for any advanced features. The introduction of this version, and the presentation of its formal definition, facilitates the understanding of the formal definitions for Grafchart-BV, Section C.3, and Grafchart-HLV, Section C.4.

C.1 Notation

Notation useful later in this chapter are presented in this section. In order to illustrate certain notations an object with three attributes and two object methods is considered.
Attributes:

$$
\begin{aligned}
\mathcal{A}ttr &= \{size, mixer, reactor\} \\
attr_1 &= size \\
attr_2 &= mixer \\
attr_3 &= reactor
\end{aligned}
$$

The attributes are given the values

$$
\begin{aligned}
attr_1 &= 25 \\
attr_2 &= m1 \\
attr_3 &= sup.reactor \\
&\underbrace{\Rightarrow}_{during\ execution}\ r2
\end{aligned}
$$

Methods:

$$
\begin{aligned}
O\mathcal{M} &= \{agitate, heat\} \\
OM_1 &= agitate \\
OM_2 &= heat
\end{aligned}
$$

The methods are given the values

$$\begin{array}{rcl} OM_1 & = & gp2 \\ OM_2 & = & sup.reactor\ heat \\ & \underbrace{\Rightarrow}_{during\ execution} & r2\ heat \end{array}$$

Useful functions

- $Expr(b)$ gives the value of b.

$$\begin{array}{rcl} Expr(attr_1) & = & 25 \\ Expr(attr_2) & = & m1 \\ Expr(attr_3) & = & sup.reactor \end{array}$$

$$\begin{array}{rcl} Expr(OM_1) & = & gp2 \\ Expr(OM_2) & = & sup.reactor\ heat \end{array}$$

- $Eval(b)$ evaluates an expression.

$$\begin{array}{rcl} Eval(sup.reactor) & = & r2 \\ Eval(Expr(attr_3)) & = & r2 \end{array}$$

$$\begin{array}{rcl} Eval(sup.reactor\ heat) & = & r2\ heat \\ Eval(Expr(OM_2)) & = & r2\ heat \end{array}$$

- $Type(b)$ gives the type of the parameter b.

$$\begin{array}{rcl} Type(Expr(attr_1)) & = & Type(25) = integer \\ Type(Expr(attr_2)) & = & Type(m1) = mixer \\ Type(Eval(Expr(attr_3))) & = & Type(Eval(sup.reactor)) \\ & = & Type(r2) = reactor \end{array}$$

$$\begin{array}{rcl} Type(Expr(OM_1)) & = & Type(gp2) = Grafchart\ procedure \\ Type(Eval(Expr(OM_2))) & = & Type(Eval(sup.reactor\ heat) \\ & = & Type(r2\ heat) = Object\ method \end{array}$$

- $Var(b)$ gives the variables used in the expression b.

$$\begin{array}{rcl} Var(conclude\ that\ x = 10) & = & x \\ Type(Var(conclude\ that\ x = 10)) & = & Type(x) = integer \end{array}$$

C.2 Grafchart – *the Grafcet version*

The most primitive version of Grafchart is achieved if all advanced features are disregarded. Grafchart then becomes similar to Grafcet. In this section, the formal definition of Grafchart - *a Grafcet version* is presented.

A Grafchart function chart can be defined as a 9-tuple

$$G = < \mathcal{V}_r, \mathcal{V}_a, \mathcal{X}, \mathcal{T}, \mathcal{M}, \mathcal{P}_{procedure}, \mathcal{P}_{process}, \mathcal{GP}, I >$$

where:

- $\mathcal{V}_r$ is the set of receptivities, $\mathcal{V}_r = \{R_1, R_2, \cdots\}$.
  The set can be divided into two disjoint subsets, $\mathcal{V}_{ext}$ and $\mathcal{V}_{int}$. $\mathcal{V}_{ext}$ is the set of receptivities containing variables originating from the plant. $\mathcal{V}_{int}$ is the set of receptivities containing variables corresponding to the internal states of the Grafchart.
  $\mathcal{V}_r$ can also be divided into three, not necessarily disjoint, subsets.

  $$\mathcal{V}_r = \{\mathcal{V}_{r\mathcal{X}} \cup \mathcal{V}_{r\mathcal{M}} \cup \mathcal{V}_{r\mathcal{GP}}\}$$

  - $\mathcal{V}_{r\mathcal{X}}$ are the receptivities associated with the transitions not included in a macro step or a Grafchart procedure.
  - $\mathcal{V}_{r\mathcal{M}}$ are the receptivities associated with the transitions included in a macro step.
  - $\mathcal{V}_{r\mathcal{GP}}$ are the receptivities associated with the transitions included in a Grafchart procedure.
- $\mathcal{V}_a$ is the set of actions, $\mathcal{V}_a = \{A_1, A_2, \cdots\}$.
  $\mathcal{V}_a$ can be divided into three, not necessarily disjoint, subsets.

  $$\mathcal{V}_a = \{\mathcal{V}_{a\mathcal{X}} \cup \mathcal{V}_{a\mathcal{M}} \cup \mathcal{V}_{a\mathcal{GP}}\}$$

  - $\mathcal{V}_{a\mathcal{X}}$ are the actions associated with the steps not included in a macro step or a Grafchart procedure.
  - $\mathcal{V}_{a\mathcal{M}}$ are the actions associated with the steps included in a macro step.
  - $\mathcal{V}_{a\mathcal{GP}}$ are the actions associated with the steps included in a Grafchart procedure.

- $\mathcal{X}$ is the set of steps, $\mathcal{X} = \{x_1, x_2, x_3, ...\}$.
  A step $x \in \mathcal{X}$ is defined by

$$\{action\}$$

  - $action(x)$ is the actions associated with x, $action(x) \in \mathcal{V}_{a\mathcal{X}}$.

- $\mathcal{T}$ is the set of transitions, $\mathcal{T} = \{t_1, t_2, t_3, ...\}$.
  A transition $t \in \mathcal{T}$ is defined by the 3-tuple:

$$\{X_{PR}, X_{FO}, \varphi\}$$

  - $X_{PR}(t)$ is the set of previous steps of t, $X_{PR}(t) \in \{\mathcal{X} \cup \mathcal{M} \cup \mathcal{P}_{procedure} \cup \mathcal{P}_{process}\}$
  - $X_{FO}(t)$ is the set of the following steps of t, $X_{FO}(t) \in \{\mathcal{X} \cup \mathcal{M} \cup \mathcal{P}_{procedure} \cup \mathcal{P}_{process}\}$
  - $\varphi(t)$ is the receptivity associated with t, $\varphi(t) \in \mathcal{V}_{r\mathcal{X}}$.

- $\mathcal{M}$ is the set of macro steps, $\mathcal{M} = \{M_1, M_2, \cdots\}$.

  - $\mathcal{V}_{r\mathcal{M}}$ is the set of receptivities associated with the transitions in $\mathcal{M}$
  - $\mathcal{V}_{a\mathcal{M}}$ is the set of actions associated with the macro steps and the steps in $\mathcal{M}$.

- $\mathcal{P}_{procedure}$ is the set of procedure steps, $\mathcal{P} = \{P_1, P_2, \cdots\}$.
  The name of the procedure that should be called from the procedure step is given by $Procedure(P_i) \in \mathcal{GP}$.

- $\mathcal{P}_{process}$ is the set of process steps, $\mathcal{P} = \{P_1, P_2, \cdots\}$.
  The name of the procedure that should be called from the process step is given by $Process(P_i) \in \mathcal{GP}$.

- $\mathcal{GP}$ is the set of Grafchart Procedures, $\mathcal{GP} = \{GP_1, GP_2, \cdots\}$ that can be called from the Grafchart.

  - $\mathcal{V}_{r\mathcal{GP}}$ is the set of receptivities associated with the transitions in $\mathcal{GP}$
  - $\mathcal{V}_{a\mathcal{GP}}$ is the set of actions associated with the Grafchart procedure and the steps in $\mathcal{GP}$.

- I is the set of initial steps, $I \subseteq \mathcal{X}$.

In order for the formal definition of Grafchart – *a Grafcet version* to be complete, a formal definition of macro steps, Grafchart procedures, receptivities and actions has to be given.

(1) A macro step M_i is defined as a 10-tuple

$$M_i = < \mathcal{V}_{ri}, \mathcal{V}_{ai}, \mathcal{X}_i, \mathcal{T}_i, \mathcal{M}_i, \mathcal{P}_{procedure,i}, \mathcal{P}_{process,i}, \mathcal{GP}_i, Enter_i, Exit_i >$$

where:

- $\mathcal{V}_{ri}, \mathcal{V}_{ai}, \mathcal{X}_i, \mathcal{T}_i$ are defined as $\mathcal{V}_r, \mathcal{V}_a, \mathcal{X}, \mathcal{T}$, for a Grafchart.
- $\mathcal{M}_i$ is the finite set of macro steps, $\mathcal{M}_i = \{M_{i,1}, M_{i,2}, \cdots\}$. The macro steps are not allowed to be infinitely recursive. The actions associated with a macro step $M_{i,j} \in \mathcal{M}_i$ are given by the set $action(M_{i,j}) \in \mathcal{V}_{a\mathcal{M}_i}$.
  - $\mathcal{V}_{r\mathcal{M}_i} = \{\mathcal{V}_{ri1} \cup \mathcal{V}_{ri2} \cup \cdots\}$.
  - $\mathcal{V}_{a\mathcal{M}_i} = \{\mathcal{V}_{ai1} \cup \mathcal{V}_{ai2} \cup \cdots \cup action(M_{i1}) \cup action(M_{i2}) \cup \cdots\}$.
- $\mathcal{P}_{procedure,i}$ is the finite set of Procedure steps, $\mathcal{P}_{procedure} = \{P_1, P_2 \cdots\}$.
- $\mathcal{P}_{process,i}$ is the finite set of Process steps, $\mathcal{P}_{process} = \{P_1, P_2 \cdots\}$.
- $\mathcal{GP}_i$ are the set of Grafchart procedures that can be called from M_i.
- $Enter_i$ is the enter step of the macro step, $Enter_i \subset \mathcal{X}_i$.
- $Exit_i$ is the exit step of the macro step, $Exit_i \subset \mathcal{X}_i$.

(2) A Grafchart procedure, GP_i is defined by the 10-tuple

$$GP_i = < \mathcal{V}_{ri}, \mathcal{V}_{ai}, \mathcal{X}_i, \mathcal{T}_i, \mathcal{M}_i, \mathcal{P}_{procedure,i}, \mathcal{P}_{process,i}, \mathcal{GP}_i, Enter_i, Exit_i >$$

where:

- $\mathcal{V}_{ri}, \mathcal{V}_{ai}, \mathcal{X}_i, \mathcal{T}_i$ are defined as $\mathcal{V}_r, \mathcal{V}_a, \mathcal{X}, \mathcal{T}$ for a Grafchart.
- $\mathcal{M}_i$ is the finite set of macro steps, $\mathcal{M}_i$ is defined as for a Grafchart.

- $\mathcal{P}_{procedure,i}$ is the finite set of Procedure steps, $\mathcal{P}_{procedure} = \{P_1, P_2 \cdots\}$.
- $\mathcal{P}_{process,i}$ is the finite set of Process steps, $\mathcal{P}_{process} = \{P_1, P_2 \cdots\}$.
- $\mathcal{GP}_i$ are the finite set of Grafchart procedures.
  $\mathcal{GP}_i = \{GP_{i,1}, GP_{i,2}, \ \cdots\}$. The calls to Grafchart procedures are not allowed to be infinitely recursive.
  The actions associated with a Grafchart procedure $GP_{i,j} \in \mathcal{GP}_i$ are given by the set $action(GP_{i,j}) \in \mathcal{V}_{a\mathcal{GP}_i}$.
  - $\mathcal{V}_{r\mathcal{GP}_i} = \{\mathcal{V}_{ri1} \cup \mathcal{V}_{ri2} \cup \cdots\}$.
  - $\mathcal{V}_{a\mathcal{GP}_i} = \{\mathcal{V}_{ai1} \cup \mathcal{V}_{ai2} \cup \cdots \cup action(GP_{i,1}) \cup action(GP_{i,2}) \cup \cdots\}$.
- $Enter_i$ is the enter step of the Grafchart procedure, $Enter_i \subset \mathcal{X}_i$.
- $Exit_i$ is the exit step of the Grafchart procedure, $Exit_i \subset \mathcal{X}_i$.

(3) A receptivity, R_i is defined by the pair

$$R_i = \{event, condition\}$$

where: *event* and *condition* are functions such that

$$\begin{array}{rcl} event(R_i) & \in & any\ G2\ event\ expression \\ condition(R_i) & \in & any\ boolean\ G2\ expression \end{array}$$

(4) An action, A_i is defined by the pair

$$A_i = \{actiontype, actiontext\}$$

where: *actiontype* and *actiontext* are functions such that

$$\begin{array}{rcl} actiontype(A_i) & \in & \{initially, finally, always, abortive\} \\ actiontext(A_i) & \in & any\ G2\ rule\ action\ statement \end{array}$$

EXAMPLE C.2.1
In order to illustrate how the formal definition can be used an example is presented. Figure C.2 shows a Grafchart for which the formal definition is given in this example.

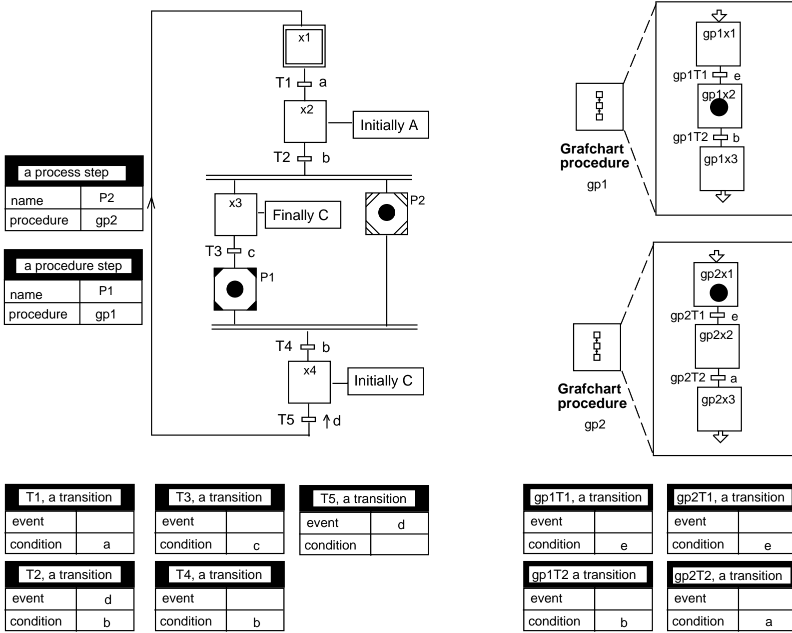


Figure C.2 A Grafchart function chart.

The Grafchart function chart in Figure C.2 is given by the tuple

$$G =< \mathcal{V}_r, \mathcal{V}_a, \mathcal{X}, \mathcal{T}, M, \mathcal{P}_{procedure}, \mathcal{P}_{process}, \mathcal{GP}, I >$$

where:

$$\begin{aligned}
\mathcal{V}_r \quad &= \quad \{\mathcal{V}_{rX} \cup \mathcal{V}_{r\mathcal{M}} \cup \mathcal{V}_{r\mathcal{GP}}\} \\
&\quad\quad \mathcal{V}_{rX} = \{R_1, R_2, R_3, R_4, R_5\} \\
&\quad\quad event(R_1) = - \quad condition(R_1) = a \\
&\quad\quad event(R_2) = d \quad condition(R_1) = b \\
&\quad\quad event(R_3) = - \quad condition(R_1) = c \\
&\quad\quad event(R_4) = - \quad condition(R_1) = b \\
&\quad\quad event(R_5) = d \quad condition(R_1) = -
\end{aligned}$$

$$
\begin{aligned}
\mathcal{V}_a &= \{\mathcal{V}_{aX} \cup \mathcal{V}_{a\mathcal{M}} \cup \mathcal{V}_{a\mathcal{GP}}\} \\
&\quad \mathcal{V}_{aX} = \{A_1,\ A_2,\ A_3\} \\
&\quad\ \ actiontype(A_1) = Initially \quad actiontext(A_1) = A \\
&\quad\ \ actiontype(A_2) = Finally \quad actiontext(A_2) = C \\
&\quad\ \ actiontype(A_3) = Initially \quad actiontext(A_3) = C
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{X} &= \{x1, x2, x3, x4\} \\
&\quad\ \ action(x2) = A_1 \qquad action(x3) = A_2 \\
&\quad\ \ action(x4) = A_3
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{T} &= \{T1, T2, T3, T4, T5\} \\
&\quad X_{PR}(T1) = x1 \qquad X_{FO}(T1) = x2 \qquad \phi(T1) = R_1 \\
&\quad X_{PR}(T2) = x2 \qquad X_{FO}(T2) = \{x3, P2\} \qquad \phi(T2) = R_2 \\
&\quad X_{PR}(T3) = x3 \qquad X_{FO}(T2) = P1 \qquad \phi(T3) = R_3 \\
&\quad X_{PR}(T4) = \{P1, P2\} \qquad X_{FO}(T4) = x4 \qquad \phi(T4) = R_4 \\
&\quad X_{PR}(T5) = x4 \qquad X_{FO}(T5) = x1 \qquad \phi(T5) = R_5
\end{aligned}
$$

$$
\mathcal{M} = \emptyset
$$

$$
\begin{aligned}
\mathcal{P}_{procedure} &= \{P1\} \\
&\quad Procedure(P1) = gp1
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{P}_{process} &= \{P2\} \\
&\quad Process(P2) = gp2
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{GP} &= \{gp1, gp2\} \\
&\quad \mathcal{V}_{r\mathcal{GP}} = \{\mathcal{V}_{r\ gp1} \cup \mathcal{V}_{r\ gp2}\} \\
&\quad \mathcal{V}_{a\mathcal{GP}} = \{\mathcal{V}_{a\ gp1} \cup \mathcal{V}_{a\ gp2}\ \}
\end{aligned}
$$

$$
I = x1
$$

In order for the formal definition for the Grafchart function chart shown in Figure C.2 to be complete, also the two Grafchart procedures named *gp1* and *gp2* have to be formally defined.

- The Grafchart Procedure $gp1$ in Figure C.2 is given by the tuple

$$gp1 = < \mathcal{V}_{r\ gp1}, \mathcal{V}_{a\ gp1}, \mathcal{X}, \mathcal{T}, \mathcal{M}, \mathcal{P}_{procedure}, \mathcal{P}_{process}, \mathcal{GP}, I >$$

where:

$$\begin{aligned}
\mathcal{V}_{r\ gp1} &= \{R_{1\ gp1}, R_{2\ gp1}\} \\
&\quad event(R_{1\ gp1}) = -, \quad condition(R_{1\ gp1}) = e \\
&\quad event(R_{2\ gp1}) = -, \quad condition(R_{2\ gp1}) = b \\
\mathcal{V}_{a\ gp1} &= \emptyset \\
\mathcal{X} &= \{gp1x1, gp1x2, gp1x3\} \\
\mathcal{T} &= \{gp1T1, gp1T2\} \\
&\quad X_{PR}(gp1T1) = gp1x1 \quad X_{PR}(gp1T2) = gp1x2 \\
&\quad X_{FO}(gp1T1) = gp1x2 \quad X_{FO}(gp1T2) = gp1x3 \\
&\quad \phi(gp1T1) = R_{1\ gp1} \quad \phi(gp1T2) = R_{2\ gp1} \\
\mathcal{M} &= \emptyset \\
\mathcal{P}_{procedure} &= \emptyset \\
\mathcal{P}_{process} &= \emptyset \\
\mathcal{GP} &= \emptyset \\
Enter &= gp1x1 \\
Exit &= gp1x3
\end{aligned}$$

- The Grafchart Procedure $gp2$ in Figure C.2 is given by the tuple

$$gp2 = < \mathcal{V}_{r\ gp2}, \mathcal{V}_{a\ gp2}, \mathcal{X}, \mathcal{T}, \mathcal{M}, \mathcal{P}_{procedure}, \mathcal{P}_{process}, \mathcal{GP}, I >$$

where:

$$\begin{aligned}
\mathcal{V}_{r\ gp2} &= \{R_{1\ gp1}, R_{2\ gp1}\} \\
&\quad event(R_{1\ gp2}) = -, \quad condition(R_{1\ gp2}) = e \\
&\quad event(R_{2\ gp2}) = -, \quad condition(R_{2\ gp2}) = a
\end{aligned}$$

$$
\begin{aligned}
\mathcal{V}_{a\ gp2} &= \emptyset \\
\mathcal{X} &= \{gp2x1, gp2x2, gp2x3\} \\
\mathcal{T} &= \{gp2T1, gp2T2\}
\end{aligned}
$$

$$
\begin{array}{ll}
X_{PR}(gp2T1) = gp1x1 & X_{PR}(gp2T2) = gp1x2 \\
X_{FO}(gp2T1) = gp1x2 & X_{FO}(gp2T2) = gp1x3 \\
\phi(gp2T1) = R_{1\ gp2} & \phi(gp2T2) = R_{2\ gp2}
\end{array}
$$

$$
\begin{aligned}
\mathcal{M} &= \emptyset \\
\mathcal{P}_{procedure} &= \emptyset \\
\mathcal{P}_{process} &= \emptyset \\
\mathcal{GP} &= \emptyset \\
Enter &= gp2x1 \\
Exit &= gp2x3
\end{aligned}
$$

#

The formal definition for Grafchart - *the grafcet version*, is similar to the formal definition for Grafcet presented in Chapter 3.3.

A Grafcet can be defined as a 5-tuple

$$G = < \mathcal{V}_r, \mathcal{V}_a, \mathcal{X}, \mathcal{T}, I >$$

A Grafchart can be defined as an 9-tuple

$$G = < \mathcal{V}_r, \mathcal{V}_a, \mathcal{X}, \mathcal{T}, M, \mathcal{P}_{procedure}, \mathcal{P}_{process}, \mathcal{GP}, I >$$

What differs in the two definitions are the macro steps M, the procedure steps $\mathcal{P}_{procedure}$, the process steps $\mathcal{P}_{process}$, and the Grafchart procedures $\mathcal{GP}$, included in the Grafchart definition.

Another difference is the meaning of $\mathcal{V}_r$ and $\mathcal{V}_a$. In Grafcet they are variables specifying the receptivities and the actions respectively. In Grafchart they are the receptivities and the actions, i.e., they are the corresponding G2 statements.

C.3 Grafchart – *the basic version*

The formal definition for Grafchart – *the basic version* also takes into consideration the parameterization and methods and message passing features. The formal definition now concerns a complete Grafchart application and not only a single function chart.

A Grafchart-BV application can be defined as a 4-tuple:

$$Appl = <\Sigma, G, \mathcal{GP}, O>$$

where:

- Σ defines all the types that are allowed in the application.
- G is the main function chart implemented by a Grafchart process.
- $\mathcal{GP}$ is the finite set of stand alone procedures implemented by Grafchart Procedures, $\mathcal{GP} = \{GP_1, GP_2, \cdots\}$.
- O is the finite set of objects, $O = \{O_1, O_2, \cdots\}$.

The formal definitions of a Grafchart process G, a Grafchart procedure GP, and an object O are now presented.

(1). A Grafchart process, G, can be defined as a tuple:

$$G = <\Sigma, \mathcal{V}_r, \mathcal{V}_a, \mathcal{X}, \mathcal{T}, \mathcal{M}, \mathcal{P}_{procedure}, \mathcal{P}_{process}, I, \mathcal{A}ttr>$$

where:

- Σ is a set of types. It defines the different types that are allowed in the Grafchart process.
- $\mathcal{V}_r$ is the set of receptivities associated with the transitions. $\mathcal{V}_r$ can be divided into two, not necessarily disjoint, subsets.

$$\mathcal{V}_r = \{\mathcal{V}_{r\mathcal{X}} \cup \mathcal{V}_{r\mathcal{M}}\}$$

- - $\mathcal{V}_{r\mathcal{X}}$ are the receptivities associated with the transitions not included in a macro step.
  - $\mathcal{V}_{r\mathcal{M}}$ are the receptivities associated with the transitions included in the macro steps.
- $\mathcal{V}_a$ is the set of actions issued to the plant. $\mathcal{V}_a$ can be divided into three, not necessarily disjoint, subsets.

$$\mathcal{V}_a = \{\mathcal{V}_{a\mathcal{X}} \cup \mathcal{V}_{a\mathcal{M}} \cup \mathcal{V}_{aG}\}$$

  - $\mathcal{V}_{a\mathcal{X}}$ are the actions associated with the steps not included in a macro step.
  - $\mathcal{V}_{a\mathcal{M}}$ are the actions associated with the macro steps.
  - $\mathcal{V}_{aG}$ are the actions associated with the Grafchart process G.
- $\mathcal{X}$ is the set of steps, $\mathcal{X} = \{x_1, x_2, x_3, \cdots\}$.
- $\mathcal{T}$ is the set of transitions, $\mathcal{T} = \{t_1, t_2, t_3, \cdots\}$. $\mathcal{T}$ can be divided into two disjoint subsets.

$$\mathcal{T} = \{\mathcal{T}_o \cup \mathcal{T}_{exception}\}$$

  - $\mathcal{T}_o$ is the set of ordinary transitions, $\mathcal{T}_o = \mathcal{T} \setminus \mathcal{T}_{exception}$
  - $\mathcal{T}_{exception}$ is the set of exception transitions.
- $\mathcal{M}$ is the set of macro steps, $\mathcal{M} = \{M_1, M_2, \cdots\}$.
  - $\mathcal{V}_{r\mathcal{M}} = \{\mathcal{V}_{rM_1} \cup \mathcal{V}_{rM_2} \cup \cdots\}$.
  - $\mathcal{V}_{a\mathcal{M}} = \{\mathcal{V}_{aM_1} \cup \mathcal{V}_{aM_2} \cup \cdots\}$.
- $\mathcal{P}_{procedure}$ is the set of procedure steps, $P = \{P_1, P_2, \cdots\}$.
- $\mathcal{P}_{process}$ is the set of process steps, $P = \{P_1, P_2, \cdots\}$.
- I is the set of initial steps, $I \subseteq \mathcal{X}$.
- $\mathcal{A}ttr$ defines the attributes associated with the Grafchart process.
  - $\forall attr_i \in \mathcal{A}ttr : [\{Type(Eval(Expr(attr_i))) \vee Type(Expr(attr_i))\} \in \Sigma]$

(2). A Grafchart procedure, GP_i is defined by the tuple:

$$GP_i = < \Sigma, \mathcal{V}_{ri}, \mathcal{V}_{ai}, \mathcal{X}_i, \mathcal{T}_i, \mathcal{M}_i, \mathcal{P}_{procedure,i}, \mathcal{P}_{process,i}, \mathcal{GP}_i, Enter_i, Exit_i, \mathcal{A}ttr >$$

where:

- Σ is the set of types allowed in the Grafchart procedure.
- $\mathcal{V}_{ri}$ is the set of receptivities associated with the transitions. $\mathcal{V}_{ri}$ can be divided into three, not necessarily disjoint, subsets.

$$\mathcal{V}_{ri} = \{\mathcal{V}_{r\mathcal{X}i} \cup \mathcal{V}_{r\mathcal{M}i} \cup \mathcal{V}_{r\mathcal{GP}i}\}$$

  - $\mathcal{V}_{r\mathcal{X}i}$ are the receptivities associated with the transitions not included in a macro step or a Grafchart procedure.
  - $\mathcal{V}_{r\mathcal{M}i}$ are the receptivities associated with the transitions included in a macro step.
  - $\mathcal{V}_{r\mathcal{GP}i}$ are the receptivities associated with the transitions included in a Grafchart procedure.
- $\mathcal{V}_{ai}$ is the set of actions issued to the plant. $\mathcal{V}_{ai}$ can be divided into three, not necessarily disjoint, subsets

$$\mathcal{V}_{ai} = \{\mathcal{V}_{a\mathcal{X}i} \cup \mathcal{V}_{a\mathcal{M}i} \cup \mathcal{V}_{a\mathcal{GP}i} \cup \mathcal{V}_{aGPi}\}$$

  - $\mathcal{V}_{a\mathcal{X}i}$ are the actions associated with the steps not included in a macro step or a Grafchart procedure.
  - $\mathcal{V}_{a\mathcal{M}i}$ are the actions associated with the steps included in a macro step.
  - $\mathcal{V}_{a\mathcal{GP}i}$ are the actions associated with the steps included in a Grafchart procedure.
  - $\mathcal{V}_{aGPi}$ are the actions associated with the Grafchart procedure GP.
- $\mathcal{X}_i, \mathcal{T}_i$ are defined as $\mathcal{X}, \mathcal{T}$ for a Grafchart process.
- $\mathcal{M}_i$ is the set of macro steps, $\mathcal{M}_i = \{M_{i,1}, M_{i,2} \cdots\}$.
  - $\mathcal{V}_{r\mathcal{M}_i} = \{\mathcal{V}_{rM_{i,1}} \cup \mathcal{V}_{rM_{i,2}} \cup \cdots\}$.
  - $\mathcal{V}_{a\mathcal{M}_i} = \{\mathcal{V}_{aM_{i,1}} \cup \mathcal{V}_{aM_{i,2}} \cup \cdots\}$.

- $\mathcal{P}_{procedure,i}$ and $\mathcal{P}_{process,i}$ are defined as $\mathcal{P}_{procedure}$ and $\mathcal{P}_{process}$ for a Grafchart process.
- $\mathcal{GP}_i$ is the set of Grafchart procedures, $\mathcal{GP}_i = \{GP_{i,1}, GP_{i,2}, \cdots\}$, that can be called from the Grafchart procedure.
  - $\mathcal{V}_{r\mathcal{GP}_i} = \{\mathcal{V}_{rGP_{i,1}} \cup \mathcal{V}_{rGP_{i,2}} \cup \cdots\}$.
  - $\mathcal{V}_{a\mathcal{GP}_i} = \{\mathcal{V}_{aGP_{i,1}} \cup \mathcal{V}_{aGP_{i,2}} \cup \cdots\}$.
- $Enter_i$ is the enter step of the Grafchart procedure, $Enter_i \subset \mathcal{X}_i$.
- $Exit_i$ is the exit step of the Grafchart procedure, $Exit_i \subset \mathcal{X}_i$.
- $\mathcal{A}ttr$ is the set of attributes associated with the Grafchart procedure.
  - $\forall attr_i \in \mathcal{A}ttr : [\{Type(Eval(Expr(attr_i))) \vee Type(Expr(attr_i))\} \in \Sigma]$

(3) An object, O is defined by the 3-tuple

$$O =< \Sigma, \mathcal{A}ttr, \mathcal{OM} >$$

where:

- Σ is the set of types allowed for the object.
- $\mathcal{A}ttr$ is the set of attributes associated with the object.
  - $\forall attr_i \in \mathcal{A}ttr : [\{Type(Eval(Expr(attr_i) \vee Type(Expr(attr_i))\} \in \Sigma]$
- $\mathcal{OM}$ is the set of object methods associated with the object, $\mathcal{OM} = \{OM_1, OM_2, \cdots\}$.
  - An object method, OM_i, is defined in the same way as a Grafchart procedure.

The formal definitions of a step x, a transition t, a macro step M, a procedure step $P_{procedure}$, a process step $P_{process}$, a receptivity and an action are now presented.

(a) A step x is defined by:

$$\{action\}$$

- $action$ is a function defined from $\mathcal{X}$ into expressions such that:

$$\forall x \in \mathcal{X} : [action(x) \in \mathcal{V}_{a\mathcal{X}}]$$

  i.e., $action(x)$ defines the actions associated with the step x.

(b) A transition t is defined by the 3-tuple:

$$\{X_{PR}, X_{FO}, \varphi\}$$

- X_{PR} is a function defined from $\mathcal{T}$ into expressions such that:

$$\forall t \in \mathcal{T} : [X_{PR}(t) \in \{\mathcal{X} \cup \mathcal{M} \cup \mathcal{P}_{procedure} \cup \mathcal{P}_{process}\}]$$

  i.e., $X_{PR}(t)$ is the set of previous steps.

- X_{FO} is a function defined from $\mathcal{T}$ into expressions such that:

$$\forall t \in \mathcal{T} : [X_{FO}(t) \in \{\mathcal{X} \cup \mathcal{M} \cup \mathcal{P}_{procedure} \cup \mathcal{P}_{process}\}]$$

  i.e., $X_{FO}(t)$ is the set of following steps.

- φ a function defined from $\mathcal{T}$ into expressions such that:

$$\forall t \in \mathcal{T} : [\varphi(t) \in \mathcal{V}_{r\mathcal{X}}]$$

  i.e., $\varphi(t)$ defines the receptivity associated with the transition t.

(c) A macro step M_i is defined as a tuple:

$$M_i = < \Sigma, \mathcal{V}_{ri}, \mathcal{V}_{ai}, \mathcal{X}_i, \mathcal{T}_i, \mathcal{M}_i, \mathcal{P}_{procedure,i}, \mathcal{P}_{process,i}, \mathcal{GP}_i, Enter_i, Exit_i, \mathcal{A}ttr >$$

where:

- Σ is the set of types allowed in the macro step.
- $\mathcal{V}_{ri}$ is the set of receptivities associated with the transitions. $\mathcal{V}_{ri}$ can be divided into two, not necessarily disjoint, subsets.

$$\mathcal{V}_{ri} = \{\mathcal{V}_{r\mathcal{X}i} \cup \mathcal{V}_{r\mathcal{M}i} \cup \mathcal{V}_{r\mathcal{GP}i}\}$$

  - $\mathcal{V}_{r\mathcal{X}i}$, $\mathcal{V}_{r\mathcal{M}i}$, and $\mathcal{V}_{r\mathcal{GP}i}$ are defined as $\mathcal{V}_{r\mathcal{X}}$, $\mathcal{V}_{r\mathcal{M}}$, and $\mathcal{V}_{r\mathcal{GP}}$ for a Grafchart procedure.
- $\mathcal{V}_{ai}$ is the set of actions issued to the plant. $\mathcal{V}_{ai}$ can be divided into three, not necessarily disjoint, subsets

$$\mathcal{V}_{ai} = \{\mathcal{V}_{a\mathcal{X}i} \cup \mathcal{V}_{a\mathcal{M}i} \cup \mathcal{V}_{a\mathcal{GP}i} \cup \mathcal{V}_{aMi}\}$$

  - $\mathcal{V}_{a\mathcal{X}i}$, $\mathcal{V}_{a\mathcal{M}i}$, and $\mathcal{V}_{a\mathcal{GP}i}$ are defined as $\mathcal{V}_{a\mathcal{X}}$, $\mathcal{V}_{a\mathcal{M}}$, and $\mathcal{V}_{a\mathcal{GP}}$ for a Grafchart procedure.
  - $\mathcal{V}_{aMi}$ are the actions associated with the macro step M_i.
- $\mathcal{X}_i, \mathcal{T}_i$ are defined as $\mathcal{X}, \mathcal{T}$, for a Grafchart process.
- $\mathcal{M}_i$ is defined as $\mathcal{M}$ for a Grafchart process.
- $\mathcal{P}_{procedure,i}$ and $\mathcal{P}_{process,i}$ is defined as $\mathcal{P}_{procedure}$ and $\mathcal{P}_{process}$ for a Grafchart process.
- $\mathcal{GP}_i$ is defined as $\mathcal{GP}$ for a Grafchart procedure.
- $Enter_i$ is the enter step of the macro step, $Enter_i \subset \mathcal{X}_i$.
- $Exit_i$ is the exit step of the macro step, $Exit_i \subset \mathcal{X}_i$.
- $\mathcal{A}ttr$ is the set of attributes associated with the macro step.
  - $\forall attr_i \in \mathcal{A}ttr : [\{Type(Eval(Expr(attr_i))) \vee Type(Expr(attr_i))\} \in \Sigma]$

(d) A procedure step P_i is defined by the pair:

$$\{\mathcal{A}ttr, Proc\}$$

- $\mathcal{A}ttr$ is the set of attributes associated with the procedure step.
  - $\forall attr_i \in \mathcal{A}ttr : [\{Type(Eval(Expr(attr_i))) \vee Type(Expr(attr_i))\} \in \Sigma]$
- $Proc$ is the Grafchart procedure or the object method that should be called from the Procedure step.
  - $\{Type(Eval(Proc)) \vee Type(Proc)\} \in \{\mathcal{GP} \cup \mathcal{OM}\}$

(e) A process step P_i is defined by the pair:

$$\{\mathcal{A}ttr, Proc\}$$

- $\mathcal{A}ttr$ is the set of attributes associated with the process step.
  - $\forall attr_i \in \mathcal{A}ttr : [\{Type(Eval(Expr(attr_i))) \vee Type(Expr(attr_i))\} \in \Sigma]$
- $Proc$ is the Grafchart procedure or the object method that should be called from the Process step.
  - $\{Type(Eval(Proc)) \vee Type(Proc)\} \in \{\mathcal{GP} \cup \mathcal{OM}\}$

(f) A receptivity, R_i is defined by the pair

$$R_i = \{event, condition\}$$

where: *event* and *condition* are functions such that

$$\begin{aligned} event(R_i) &\in \text{any G2 event expression} \\ condition(R_i) &\in \text{any boolean G2 expression} \end{aligned}$$

(g) An action, A_i is defined by the pair

$$A_i = \{actiontype, actiontext\}$$

where: *actiontype* and *actiontext* are functions such that

$$\begin{aligned} actiontype(A_i) &\in \{initially, finally, always, abortive\} \\ actiontext(A_i) &\in any\ G2\ rule\ action\ statement \end{aligned}$$

EXAMPLE C.3.1

In order to illustrate how the formal definition can be used a Grafchart-BV application is presented and its definition is given. The Grafchart application is a batch recipe, see Figure C.3. The batch recipe is supposed to be executed in the batch cell shown in Figure 7.6.

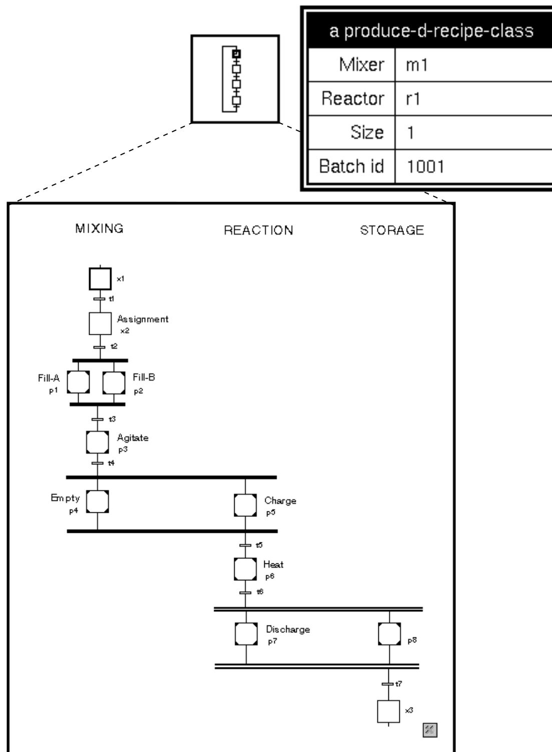


Figure C.3 A Grafchart-BV application consisting of a batch recipe.

The batch recipe application is described by the 4-tuple:

$$Appl = <\Sigma, G, \mathcal{GP}, O>$$

- Σ is the set of all types allowed in the application.

$$\Sigma = \{\text{integer, boolean, storage-tank, mixer, reactor, product-tank, on-off-valve, control-valve, 2in-valve, level-sensor, temperature-sensor, pressure-sensor, flow-sensor, pump, agitator-switch, pid-controller}\}$$

- G is a Grafchart process describing the recipe.
- $\mathcal{GP}$ is the set of stand alone procedures.

$$\mathcal{GP} = \emptyset$$

- O is the set of objects.

$$O = \{stA, stB, m1, r1, ptD\}$$

The definitions of the Grafchart process G and the objects stA, stB, $m1$, $r1$, and ptD are:

(1) The Grafchart process G, see Figure C.3, is described by the tuple:

$$G = <\Sigma, \mathcal{V}_r, \mathcal{V}_a, \mathcal{X}, \mathcal{T}, \mathcal{M}, \mathcal{P}_{procedure}, \mathcal{P}_{process}, I, \mathcal{A}ttr>$$

where

- $\Sigma = \{$ integer, boolean, storage-tank, mixer, reactor, product-tank, on-off-valve $\}$
- $\mathcal{V}_r = \{\mathcal{V}_{r\mathcal{X}} \cup \mathcal{V}_{r\mathcal{M}}$
  - $\mathcal{V}_{r\mathcal{M}} = \emptyset$
  - $\mathcal{V}_{r\mathcal{X}} = \{R_1, R_2, R_3, R_4, R_5, R_6, R_7\}$

  The receptivities, $\mathcal{V}_{r\mathcal{X}}$ are shown in Figure C.4.

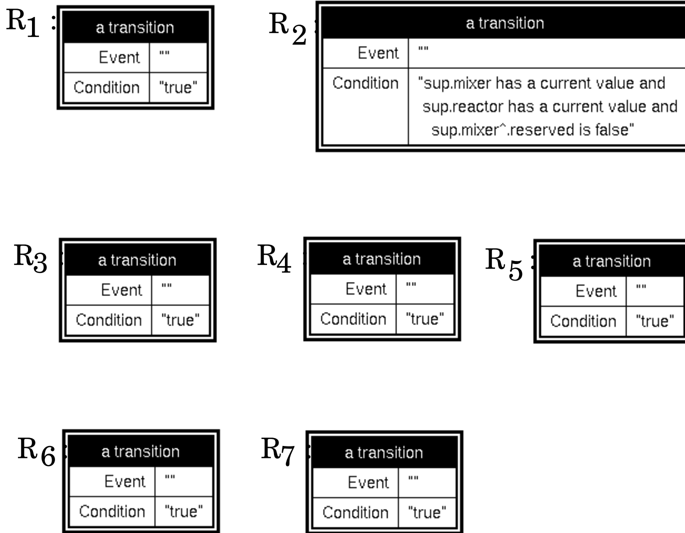


Figure C.4 The set of receptivities for the Grafchart process G.

$event(R_1) = -$ $\quad$ $condition(R_1) = true$

$event(R_2) = -$ $\quad$ $condition(R_2) =$"sup.mixer has a current value and sup.reactor has a current value and sup.mixer^.reserved is false"

$event(R_3) = -$ $\quad$ $condition(R_3) = true$

$event(R_4) = -$ $\quad$ $condition(R_4) = true$

$event(R_5) = -$ $\quad$ $condition(R_5) = true$

$event(R_6) = -$ $\quad$ $condition(R_6) = true$

$event(R_7) = -$ $\quad$ $condition(R_7) = true$

- $\mathcal{V}_a = \{\mathcal{V}_{a\mathcal{X}} \cup \mathcal{V}_{a\mathcal{M}} \cup \mathcal{V}_{aG}\}$
  - $\mathcal{V}_{aG} = \emptyset$
  - $\mathcal{V}_{a\mathcal{M}} = \emptyset$
  - $\mathcal{V}_{a\mathcal{X}} = \{A_1, A_2\}$

  The actions, $\mathcal{V}_{a\mathcal{X}}$, is shown in Figure C.5.

$$
\begin{aligned}
actiontype(A_1) &= Initially\\
actiontext(A_1) &= \text{"If not(sup.mixer has a current value) then}\\
&\quad\text{inform the operator that "No mixer has}\\
&\quad\text{been assigned to the batch""}
\end{aligned}
$$

$$
\begin{aligned}
actiontype(A_2) &= Initially\\
actiontext(A_2) &= \text{"If not(sup.reactor has a current value) then}\\
&\quad\text{inform the operator that "No mixer has}\\
&\quad\text{been assigned to the batch"}
\end{aligned}
$$

- $\mathcal{X} = \{x_1, x_2, x_3\}$
  - $action(x_1) = \emptyset$
  - $action(x_2) = \{A_1, A_2\}$
  - $action(x_3) = \emptyset$

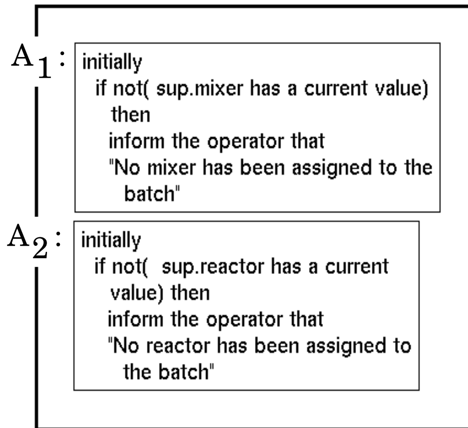


Figure C.5 The set of actions for the Grafchart process G.

- $\mathcal{T} = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7\}$
  - $\mathcal{T}_o = \mathcal{T}$
  - $\mathcal{T}_{exception} = \varnothing$

$$
\begin{array}{lll}
X_{PR}(t_1) = x_1 & X_{FO}(t_1) = x_2 & \varphi(t_1) = R_1 \\
X_{PR}(t_2) = x_2 & X_{FO}(t_2) = \{p_1, p_2\} & \varphi(t_2) = R_2 \\
X_{PR}(t_3) = \{p_1, p_2\} & X_{FO}(t_3) = p_3 & \varphi(t_3) = R_3 \\
X_{PR}(t_4) = p_3 & X_{FO}(t_4) = \{p_4, p_5\} & \varphi(t_4) = R_4 \\
X_{PR}(t_5) = \{p_4, p_5\} & X_{FO}(t_5) = p_6 & \varphi(t_5) = R_5 \\
X_{PR}(t_6) = p_6 & X_{FO}(t_6) = \{p_7, p_8\} & \varphi(t_6) = R_6 \\
X_{PR}(t_7) = \{p_7, p_8\} & X_{FO}(t_7) = x_3 & \varphi(t_7) = R_7
\end{array}
$$

- $\mathcal{M} = \varnothing$
- $\mathcal{P}_{procedure} = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$
  The set of procedure steps is shown in Figure C.6.

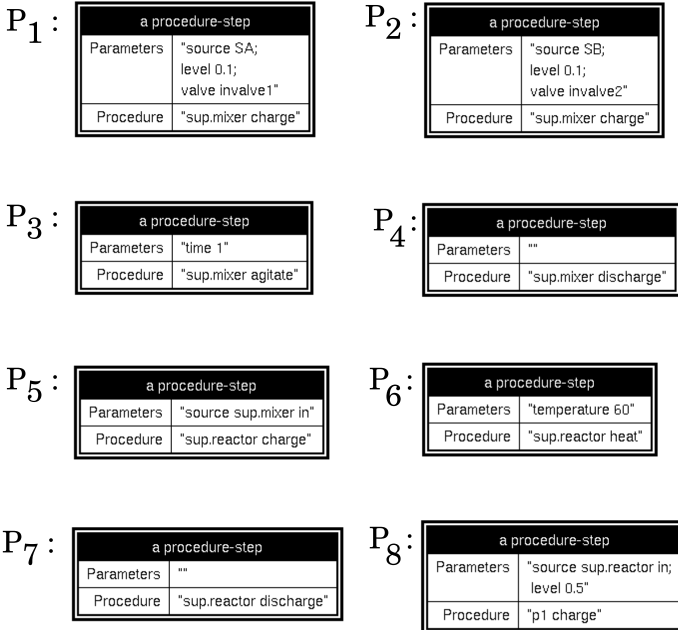


Figure C.6 The set of procedure steps for the Grafchart process G.

$$\mathcal{A}ttr(p_1) = \{source, level, valve\}$$
$$Expr(source) = SA, Expr(level) = 0.1, Expr(valve) = invalve1$$
$$Proc(p_1) = \{sup.mixer\ charge\}$$

$$\mathcal{A}ttr(p_2) = \{source, level, valve\}$$
$$Expr(source) = SB, Expr(level) = 0.1, Expr(valve) = invalve2$$
$$Proc(p_2) = \{sup.mixer\ charge\}$$

$$\mathcal{A}ttr(p_3) = \{time\}$$
$$Expr(time) = 1$$
$$Proc(p_3) = \{sup.mixer\ agitate\}$$

$$\mathcal{A}ttr(p_4) = \{\}$$
$$Proc(p_4) = \{sup.mixer\ discharge\}$$

$$\mathcal{A}ttr(p_5) = \{source\}$$
$$Eval(Expr(source)) = Eval(sup.mixer\ in) = m1$$
$$Proc(p_5) = \{sup.reactor\ charge\}$$

$$\mathcal{A}ttr(p_6) = \{temperature\}$$
$$Expr(temperature) = 60$$
$$Proc(p_6) = \{sup.reactor\ heat\}$$

$$\mathcal{A}ttr(p_7) = \{\}$$
$$Proc(p_7) = \{sup.reactor\ discharge\}$$

$$\mathcal{A}ttr(p_8) = \{source, level\}$$
$$Eval(Expr(source)) = Eval(sup.reactor\ in) = r1, Expr(level) = 0.5$$
$$Proc(p_8) = \{p1\ charge\}$$

- $\mathcal{P}_{process} = \emptyset$
- $I = \{x_1\}$
- $\mathcal{A}ttr = \{mixer, reactor, size, batchid\}$

(2) The object $m1$, see Figure C.7 is defined by the 3-tuple:

$$M1 = \{\Sigma, \mathcal{A}ttr, \mathcal{OM}\}$$

where:

- Σ = { boolean, on-off-valve, pump, level-sensor, temperature-sensor, pressure-sensor, agitator-switch }
- $\mathcal{A}ttr$ = { reserved, invalve1, invalve2, outvalve, pump, level-sensor, temperature-sensor, pressure-sensor, agitator }
- $\mathcal{OM}$ = { Charge, Agitate, Discharge }

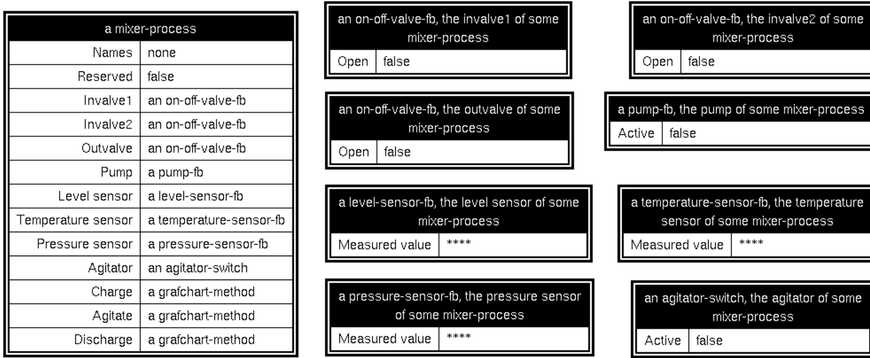


Figure C.7 The mixer $m1$.

(3) The object $R1$, see Figure C.8, is defined by the 3-tuple:

$$R1 = \{\Sigma, \mathcal{A}ttr, \mathcal{OM}\}$$

where:

- Σ = { boolean, integer, control-valve, on-off-valve, pump, level-sensor, temperature-sensor, pressure-sensor, flow-sensor, pid-controller, 2in-valve, agitator-switch }
- $\mathcal{A}ttr$ = { reserved, invalve1, invalve2, outvalve, flowvalve, pump, level-sensor, temperature-sensor, pressure-sensor, flow-sensor, level-PID, temperature-PID, flow-PID, 2-in, agitiator }
- $\mathcal{OM}$ = { Charge, Load, Agitate, Heat, Cool, Discharge }

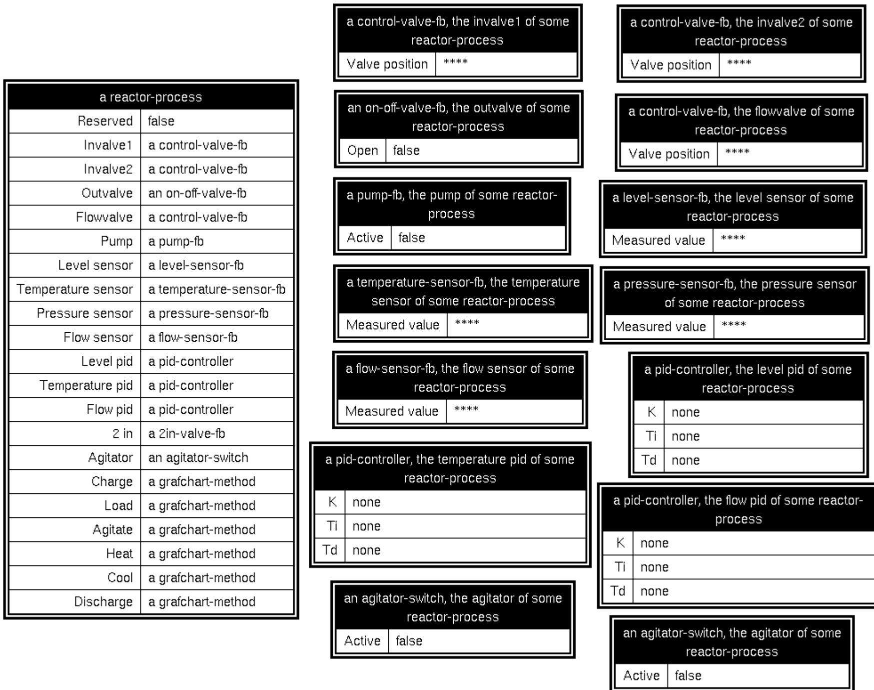


Figure C.8 The reactor $r1$.

The definitions of stA, stB, and ptD are similar to the ones for $m1$ and $r1$.

All methods belonging to the mixer or to the reactor are defined in a similar manner. Their definitions are identical to that of a Grafchart procedure. Here, only the definition of the method `Agitate` for mixer $m1$ is presented. For $m1$ the following is true:

$$Expr\{agitate\} = \texttt{mixer_agitate_method}$$

`mixer_a_m` will be use as a shorter name for `mixer_agitate_method`.

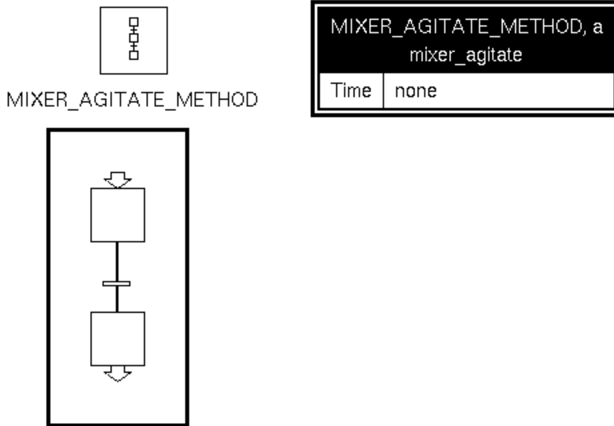


Figure C.9 The object method `mixer_agitate_method` belonging to the mixer $m1$.

The object method `mixer_agitate_method`, see Figure C.9, is defined by:

`mixer_a_m` $=< \Sigma, \mathcal{V}_r, \mathcal{V}_a, \mathcal{X}, \mathcal{T}, \mathcal{M}, \mathcal{P}_{procedure}, \mathcal{P}_{process}, \mathcal{GP}, Enter, Exit, \mathcal{A}ttr >$

where:

- $\Sigma = \{integer\}$
- $\mathcal{V}_r = \{\mathcal{V}_{r\mathcal{X}} \cup \mathcal{V}_{r\mathcal{M}} \cup \mathcal{V}_{r\mathcal{GP}}\}$
  - $\mathcal{V}_{r\mathcal{GP}} = \emptyset$
  - $\mathcal{V}_{r\mathcal{M}} = \emptyset$
  - $\mathcal{V}_{r\mathcal{X}} = \{R_1\}$
    The receptivity is shown in Figure C.10.

  $event(R_1) = -$
  $condition(R_1) =$ "active-time-larger-than(this workspace,sup.time)"

a transition	
Event	""
Condition	"active-time-larger-than(this workspace, sup.time)"

Figure C.10 The receptivity for the mixer method `agitate`.

- $\mathcal{V}_a = \{\mathcal{V}_{a\mathcal{X}} \cup \mathcal{V}_{a\mathcal{M}} \cup \mathcal{V}_{a\mathcal{GP}} \cup \mathcal{V}_{a,\texttt{mixer_a_m}}\}$
  - $\mathcal{V}_{a,\texttt{mixer_a_m}} = \emptyset \quad \mathcal{V}_{a\mathcal{GP}} = \emptyset, \quad \mathcal{V}_{a\mathcal{M}} = \emptyset$
  - $\mathcal{V}_{a\mathcal{X}} = \{A_1, A_2\}$
    The actions are shown in Figure C.11.

$actiontype(A_1) = Initially$

$actiontext(A_1) =$ conclude that self.agitator.active = true

$actiontype(A_2) = Finally$

$actiontext(A_2) =$ conclude that self.agitator.active = false

- $\mathcal{X} = \{x_1, x_2\}$
  - $action(x_1) = \{A_1, A_2\}$
  - $action(x_2) = \emptyset$
- $\mathcal{T} = \{t_1\}$
  - $\mathcal{T}_o = \mathcal{T}, \quad \mathcal{T}_{exception} = \emptyset$

$X_{PR}(t_1) = x_1 \quad X_{FO}(t_1) = x_2 \quad \varphi(t_1) = R_1$

- $\mathcal{M} = \emptyset$
- $\mathcal{P}_{procedure} = \emptyset$
- $\mathcal{P}_{process} = \emptyset$
- $\mathcal{GP} = \emptyset$
- $Enter = \{x_1\}$
- $Exit = \{x_2\}$
- $\mathcal{A}ttr = \{time\}$

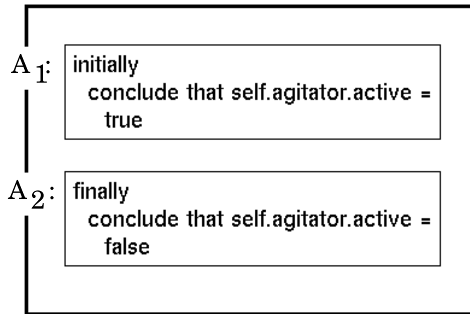


Figure C.11 The action for the mixer method agitate.

C.4 Grafchart – *the high-level version*

The formal definition of Grafchart – *the high-level version* takes into account all four features; parameterization, methods and message passing, object tokens, and multi-dimensional charts. The formal definition concerns a complete Grafchart application.

The definition of Grafchart – *the high-level version* resembles the definition of Grafchart – *the basic version*. Therefore only the differences are pointed out.

A High-Level Grafchart application can be defined as a 5-tuple:

$$Appl = <\Sigma, G, \mathcal{GP}, \mathcal{O}, \mathcal{TC}>$$

where:

- Σ defines the types allowed in the application.
- G is the main function chart implemented by a Grafchart Process.
- $\mathcal{GP}$ is the finite set of stand alone procedures implemented by Grafchart Procedures, $\mathcal{GP} = \{GP_1, GP_2, \cdots\}$.
- $\mathcal{O}$ is the finite set of objects, $\mathcal{O} = \{O_1, O_2, \cdots\}$.
- $\mathcal{TC}$ is the finite set of token colors, $\mathcal{TC} = \{TC_1, TC_2, \cdots\}$.

The formal definitions of a Grafchart process G, a Grafchart procedure GP, an object O and a token color TC are now presented.

(1) A Grafchart process, G, can be defined as a tuple:

$$G = <\Sigma, \mathcal{V}_r, \mathcal{V}_a, \mathcal{X}, \mathcal{T}, \mathcal{M}, \mathcal{P}_{procedure}, \mathcal{P}_{process}, I, \mathcal{TC}, \mathcal{A}ttr>$$

where:

- Σ, $\mathcal{V}_r$, $\mathcal{V}_a$, $\mathcal{X}$, $\mathcal{T}$, $\mathcal{M}$, $\mathcal{P}_{procedure}$, $\mathcal{P}_{process}$, I, and $\mathcal{A}ttr$ are defined in the same way as for a Grafchart process G in Grafchart-BV.
- $\mathcal{TC}$ is the set of token colors allowed in the Grafchart process.

(2) A Grafchart procedure, GP_i is defined by the tuple:

$$GP_i = < \Sigma, \mathcal{V}_{ri}, \mathcal{V}_{ai}, \mathcal{X}_i, \mathcal{T}_i, \mathcal{M}_i, \mathcal{P}_{procedure,i}, \mathcal{P}_{process,i}, \mathcal{GP}_i, Enter_i, Exit_i, \mathcal{A}ttr >$$

where:

- Σ, $\mathcal{V}_{ri}$, $\mathcal{V}_{ai}$, $\mathcal{X}_i$, $\mathcal{T}_i$, $\mathcal{M}_i$, $\mathcal{P}_{procedure,i}$, $\mathcal{P}_{process,i}$, $\mathcal{GP}_i$, $Enter_i$, $Exit_i$ and $\mathcal{A}ttr$ are defined in the same way as for a Grafchart procedure GP_i in Grafchart-BV.

(3) An object, O_i is defined by the 3-tuple

$$O_i = < \Sigma, \mathcal{A}ttr, \mathcal{OM} >$$

where:

- Σ, $\mathcal{A}ttr$, and $\mathcal{OM}$ are defined in the same way as for an object, O, in Grafchart-BV.

(4) A token color, TC_i is defined by the three-tuple

$$TC_i = < \Sigma, \mathcal{A}ttr, \mathcal{OM} >$$

where:

- Σ is the set of types allowed within the token color.
- $\mathcal{A}ttr$ is the set of attributes associated with the token color.
  - $\forall attr_i \in \mathcal{A}ttr : [\{Type(Eval(Expr(attr_i) \vee Type(Expr(attr_i))\} \in \Sigma]$

  There are default values associated with the attributes.
- $\mathcal{OM}$ is the set of object methods associated with the token color, $\mathcal{OM} = \{OM_1, OM_2, \cdots\}$.
  - An object method, OM_i, is defined in the same way as a Grafchart procedure.

The formal definitions of a step x, a transition t, a macro step M, a procedure step $P_{procedure}$, and a process step $P_{process}$ are now presented.

(a) A step x is defined by:

$$\{action\}$$

- $action$ is a function defined from $\mathcal{X}$ into expressions such that:

$$\forall x \in \mathcal{X} : [action(x)) \in \mathcal{V}_{aX}]$$

  i.e., $action(x)$ defines the actions associated with the step x.

(b) A transition t is defined by the 3-tuple:

$$\{X_{PR}, X_{FO}, \varphi\}$$

- X_{PR} is a function defined from $\mathcal{T}$ into expressions such that:

$$\forall t \in \mathcal{T} : [X_{PR}(t) \in \{\mathcal{X} \cup \mathcal{M} \cup \mathcal{P}_{procedure} \cup \mathcal{P}_{process}\}]$$

  i.e., $X_{PR}(t)$ is the set of previous steps.

- X_{FO} is a function defined from $\mathcal{T}$ into expressions such that:

$$\forall t \in \mathcal{T} : [X_{FO}(t) \in \{\mathcal{X} \cup \mathcal{M} \cup \mathcal{P}_{procedure} \cup \mathcal{P}_{process}\}]$$

  i.e., $X_{FO}(t)$ is the set of following steps.

- φ a function defined from $\mathcal{T}$ into expressions such that:

$$\forall t \in \mathcal{T} : [\varphi(t) \in \mathcal{V}_{rX}]$$

  i.e., $\varphi(t)$ defines the receptivity associated with the transition t.

(c) A macro step M_i is defined as a tuple:

$$M_i = <\Sigma, \mathcal{V}_{ri}, \mathcal{V}_{ai}, \mathcal{X}_i, \mathcal{T}_i, \mathcal{M}_i, \mathcal{P}_{procedure,i}, \mathcal{P}_{process,i}, Enter_i, Exit_i, \mathcal{T}C_i, \mathcal{A}ttr>$$

where:

- Σ, $\mathcal{V}_{ri}$, $\mathcal{V}_{ai}$, $\mathcal{X}_i$, $\mathcal{T}_i$, $\mathcal{M}_i$, $\mathcal{P}_{procedure,i}$, $\mathcal{P}_{process,i}$, $Enter_i$, $Exit_i$, and $\mathcal{A}ttr$ are defined as for a macro step M_i for Grafchart-BV.
- $\mathcal{T}C_i$ is the set of token colors defined for the macro step.

(d) A procedure step P_i is defined by:

$$\{ProcCall\}$$

- $ProcCall$ is a function defined from $\mathcal{P}_{procedure}$ into expressions such that:

$$\forall P_i \in \mathcal{P}_{procedure} : [ProcCall(P_i) \in \mathcal{P}C]$$

i.e., $ProcCall(P_i)$ defines the procedure calls associated with the procedure step P_i. $\mathcal{P}C = \{PC_1, PC_2, \cdots\}$.

(e) A process step P_i is defined by:

$$\{ProcCall\}$$

- $ProcCall$ is a function defined from $\mathcal{P}_{process}$ into expressions such that:

$$\forall P_i \in \mathcal{P}_{process} : [ProcCall(P_i) \in \mathcal{P}C]$$

i.e., $ProcCall(P_i)$ defines the procedure calls associated with the process step P_i. $\mathcal{P}C = \{PC_1, PC_2, \cdots\}$.

The formal definitions of an action A, a receptivity R and a procedure call PC are now presented.

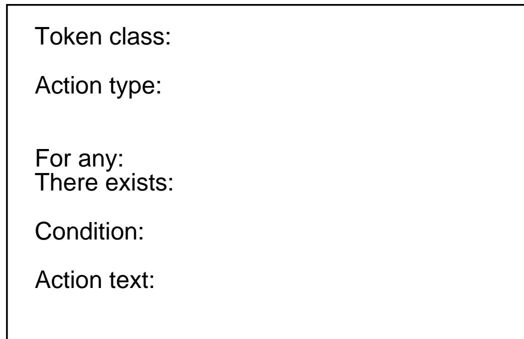


Figure C.12 A Grafchart-HLV action.

(f) An action A_i, see Figure C.12 is defined by:

$$A_i = \{\Sigma, \texttt{Token class}, \texttt{Action Type}, \texttt{For any}, \texttt{There exists}, \texttt{Condition}, \texttt{Action text}\}$$

- Σ defines the types allowed in the action.
- `Token class` is a function defined from $\mathcal{V}_a$ into expression such that:
  - $\forall A_i \in \mathcal{V}_a : [Token\ class(A_i) \in \Sigma]$

  `Token class` specifies the class of the token that activates the action. Here, only the token class name is given, e.g., "blue-token".
- `Action Type` is a function defined from $\mathcal{V}_a$ into expressions such that:
  - $\forall A_i \in \mathcal{V}_a : [ActionType(A_i) = \{Initially, Finally, Always, Abortive\}]$

  `Action Type` specifies the type of the action, i.e., Initially, Finally, Always or Abortive.

- `For any` is a function defined from $\mathcal{V}_a$ into expression such that:
  - $Type(Var(There\ exists(A_i))) \in \Sigma$
  - $\exists v \in Var(For\ any(A_i)) : [Type(v) = Tokenclass(A_i)]$

  `For any` defines the temporary-token-name of the tokens involved in the action. A name of the token mentioned in the `token class` must be declared. E.g., "For any: b:blue-token" or "For any: b1: blue-token, b2:blue-token". All names must be unique.

- `There exists` is a function defined from $\mathcal{V}_a$ into expressions such that:
  - $Type(Var(There\ exists(A_i))) \in \Sigma$

  `There exists` can be used to check the presence of other tokens than the one activating the action. E.g., "There exists: r:red-token" or "There exists: g1: green-token, g2:green-token". All names must be unique.

- `Condition` specifies the conditions that must be fulfilled for the action to be executed, e.g., "b.y = 10", "not(b1.y=20)".
- `Action text` specifies what should be done when the action is executed, e.g., "b.y = r.v", "b.on = false".

(g) A receptivity R, see Figure C.13, is defined by:

- $R_i = \{\Sigma, \texttt{Token class}, \texttt{For any}, \texttt{There exists}, \texttt{Event}, \texttt{Condition}, \texttt{Action}\}$
- Σ defines the types allowed in the receptivity.
- `Token class` is a function defined from $\mathcal{V}_r$ into expression such that:
  - $\forall R \in \mathcal{V}_r : [Type(Token\ class(R_i)) \in \Sigma]$

  `Token class` specifies the class of the token that enables the receptivity. Here, only the tokenclass name is given, e.g., "blue-token".

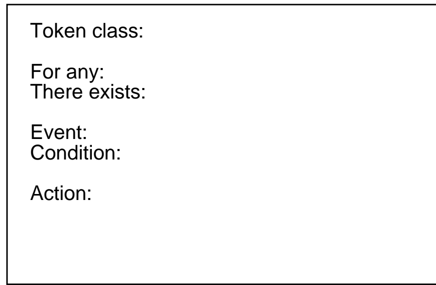


Figure C.13 A Grafchart-HLV receptivity.

- `For any` is a function defined from $\mathcal{V}_r$ into expressions such that:
  - $Type(Var(There\ exists(R_i))) \in \Sigma$
  - $\exists v \in Var(For\ any(A_i)) : [Type(v) = Tokenclass(A_i)]$

  `For any` defines the temporary-token-name of the tokens involved in the receptivity. A name of the token mentioned in the `token class` must be declared. E.g., "For any: b:blue-token" or "For any: b1: blue-token, b2:blue-token". All names must be unique.

- `There exists` is a function defined from $\mathcal{V}_r$ into expressions such that:
  - $Type(Var(There\ exists(R_i))) \in \Sigma$

  `There exists` can be used to check the presence of other tokens than the one enabling the receptivity. E.g., "There exists: r:red-token" or "There exists: g1: green-token, g2:green-token". All names must be unique.

- `Event` is a function defined from $\mathcal{V}_r$ into expressions such that:
  - $\forall R \in \mathcal{V}_r : [Type(Event(R_i)) = Boolean]$
  - $\forall R \in \mathcal{V}_r : [Type(Var(Event(R_i))) \in \Sigma]$

  `Event` specifies the event that has to occur for the enabled transition to fire. E.g.,"^b.y=10".

- `Condition` is a function defined from $\mathcal{V}_r$ into expressions such that:
  - $\forall R \in V_r : [Type(Condition(R_i)) = Boolean]$
  - $\forall R \in V_r : [Type(Var(Condition(R_i))) \in \Sigma]$

  `Condition` specifies the conditions that must be fulfilled in order for the enabled transition to fire. E.g., "r.on = true".

- `Action` is a function defined from $\mathcal{V}_r$ into expressions such that:
  - $Action(R_i) = \{Move, Create, Delete, Change\}$
  - $Type(Var(Action(R_i))) \in \Sigma$

  `Action` specifies what should happen when the transition is fired. Possible options
  - `move`: The action `move(b)`, moves the token refered to as `b` from the preceding step to the succeeding step.
  - `delete`: The action `delete(b)`, deletes the token refered to as `b`, this token must reside in the preceding step.
  - `create`: The action `create(b:blue-token)`, creates a new token and gives it the temporary-name `b`. The new token is placed in the succeeding step. By combining this action with the `change` action, the attributes of the new token can be given any value. If this action is not combined with the `change` action, the attributes of the newly created token will have their default values.
  - `change` The action `change(b.y=12, r.on=false)`, changes the y attribute of the token refered to as b and changes the on attribute of the token refered to as r.

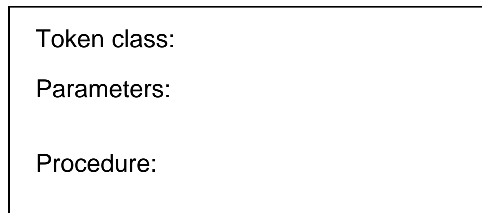


Figure C.14 A Grafchart-HLV procedure call.

(h) A procedure call PC, see Figure C.14, is defined by the 4-tuple:

$$\{\Sigma, \texttt{Token class}, \mathcal{A}ttr, \texttt{Proc}\}$$

- Σ defines the types allowed in the procedure call.
- `Token class` is a function defined from $\mathcal{PC}$ into expressions such that:
  - $\forall PC_i \in \mathcal{PC} : [Tokenclass(PC_i) \in \Sigma]$

  `Token class` specifies the class of the token that enables the procedure call. Here, only the tokenclass name is given, e.g., “blue-token”.
- $\mathcal{A}ttr$ is the set of attributes associated with the procedure-call.
  - $\forall attr_i \in \mathcal{A}ttr : [\{Type(Eval(Expr(attr_i))) \vee Type(Expr(attr_i))\} \in \Sigma]$

  $\mathcal{A}ttr$ specifies the attributes that should be sent to the Grafchart procedure.
- `Proc` defines the Grafchart procedure or the object method that should be called from the procedure call.
  - $\{Type(Eval(Expr(Proc))) \vee Type(Eval(Proc))\} \in \{\mathcal{GP} \cup \mathcal{OM}\}$

  `Proc` specifies the name of the Grafchart procedure that should be called. The name could be given either direct or indirect via a reference. The Grafchart procedure that should be call can either be a stand alone procedure or a method belonging to an object.

C.5 Remarks

The Grafchart syntax definition is based upon tuples. This kind of syntax definition is suitable for graphical languages, e.g., Grafcet and Petri nets. Grafchart mainly has a graphical syntax and a tuple-based definition of the language has therefore been chosen. However, Grafchart does also contain textual parts, e.g., the actions and the receptivities. In order to be consistent, these are also defined using tuples. For textual languages it is, however, more common to use the extended Backus-Naur form (EBNF).

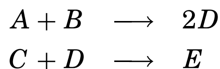
D

Dynamic Simulator

The dynamic simulator is used as a substitute for a real plant. It simulates the mass balances, energy balances and the chemical reactions. All simulations are done in real-time.

A table over the notation and the constants used in the dynamic simulator is given in the end of the appendix, Chapter D.7.

Two reactions can be simulated in the scenario:



$$A + B \longrightarrow 2D$$
$$C + D \longrightarrow E$$

The total mass balance, $m(t)$, the component mass balances, $x(t)$, and the energy balance, $T(t)$, see Figure D.1, for the reactions are calculated.

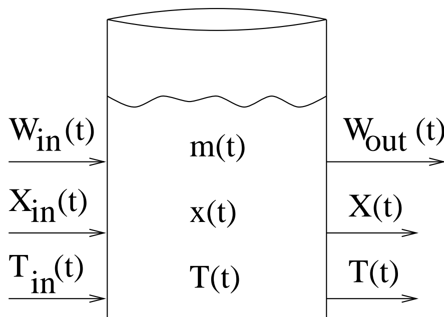


Figure D.1 Mass balance, component mass balance and energy balance.

D.1 Total Mass Balance

The total mass balance of a unit is given by a differential equation together with an initial condition.

$$\begin{aligned} \frac{dm(t)}{dt} &= \underbrace{W_{in}}_{incoming\ massflow} - \underbrace{W_{out}}_{outgoing\ massflow} \\ m(0) &= m_o \end{aligned}$$

The equation comes from the principle of the conservation of mass. $m(t)$ is the weight of the content in the unit, i.e. the mass. The mass is increased if there is an inflow to the unit and it is decreased if there is an outflow of the unit. W_{in} is the mass flowing in to the unit per time unit and W_{out} is the mass flowing of the unit per time unit.

D.2 Component Mass Balance

The component mass balance tells how much there is of a specific substance within a unit, it is given in percentage of the total mass. Unlike mass, chemical components are not conserved. If a reaction occurs inside a system, the mass of an individual component will increase if it is a product of the reaction or decrease if it is a reactant. The mass fraction, $x(t)$, changes continuously and is therefor given by a differential equation together with an initial condition.

$$\begin{aligned} \frac{dx(t)}{dt} &= \underbrace{\frac{W_{in}}{m}(x_{in} - x)}_{in-out} + \underbrace{\frac{M \cdot V}{m} r}_{produced} \\ x(0) &= x_o \end{aligned}$$

where M is the molweight of the substance, V is the volume and r is the reaction rate.

D.3 Energy Balance

The energy balance gives a differential equation for the temperature in the unit, $T(t)$. The equation is based on the first law of thermodynamics,

the conservation of energy.

$$\begin{aligned} \frac{dT(t)}{dt} &= \frac{W_{in}}{m}(T_{in} - T) + \frac{1}{c_p \cdot m} Q_p - \frac{1}{c_p \cdot m} Q_c \\ T(0) &= t_o \end{aligned}$$

Q_p is the energy produced during the reaction, Q_c is the energy that is removed by, e.g., a cooling jacket, and c_p is the heat capacity.

Energy Balance for the Reactor Jacket

The equations given above, apply to all units in the batch scenario. The reactor, however, needs one more equation that describes its jacket.

Produced reaction rate, Q_p:

$$Q_p = \Delta H_{reac} \cdot V \cdot r$$

If $Q_p \leq 0$ the reaction consumes energy, i.e., it is an endotherm reaction. If $Q_p \geq 0$ the reaction generates energy, i.e. it is an exotherm reaction.

Cooling heat, Q_c:

$$\begin{aligned} Q_c &= \mathcal{H} \cdot A_c \cdot (T - T_c) \\ A_c &= k_w \frac{m}{\rho \cdot A} \end{aligned}$$

The cooling heat is the energy that is removed due to, e.g., a cooling jacket. $\mathcal{H}$ is the heat transfer, A is the area of the reactor, k_w is the wall coefficient for the reactor and ρ is the density of the content in the reactor.

Energy balance for the jacket of the reactor:

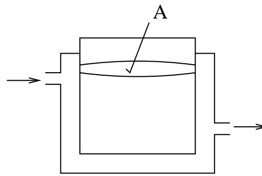


Figure D.2 Reactor with a jacket.

$$\frac{dT_c(t)}{dt} = \frac{W_{ic}}{m_c}(T_{ic} - T_c) + \frac{1}{c_p \cdot m_c} Q_c$$

D.4 Level and Volume

Equations for calculating the level and the volume for all the units are also part of the dynamic simulator.

D.5 Reaction Rates

The reaction rates, r_1 and r_2, of the reactions can be calculated by using the Arrenius equation:

$$A + B \longrightarrow 2D \tag{D.1}$$

$$C + D \longrightarrow E \tag{D.2}$$

Eq. D.1 gives:

$$\begin{aligned} r_A &= r_1 \\ r_B &= r_1 \\ r_D &= -2r_1 \end{aligned}$$

Eq. D.2 gives:

$$\begin{aligned} r_C &= r_2 \\ r_D &= r_2 \\ r_E &= -r_2 \end{aligned}$$

The reaction rate for substance D can now be calculated:

$$r_D = -2r_1 + r_2$$

The constants r_1 and r_2 are given by:

$$\begin{aligned} r_1 &= -K_1 \cdot e^{-\frac{E_1}{R(T+273)}} \cdot c_A \cdot c_B \\ &= -K_1 \cdot e^{-\frac{E}{R(T+273)}} \cdot \frac{\rho \cdot x_A}{M_A} \cdot \frac{\rho \cdot x_B}{M_B} \end{aligned}$$

$$\begin{aligned} r_2 &= -K_2 \cdot e^{-\frac{E_2}{R(T+273)}} \cdot c_C \cdot c_D \\ &= -K_2 \cdot e^{-\frac{E}{R(T+273)}} \cdot \frac{\rho \cdot x_C}{M_C} \cdot \frac{\rho \cdot x_D}{M_D} \end{aligned}$$

K_1 and K_2 are reaction constants, E_1 and E_2 are the activation energies, R is the gas-law constant, T is the temperature in $^\circ C$, c_i, x_i and M_i are the concentration the mass fraction and the mole weight of substance i, ρ is the density.

D.6 Implementation

The objects are defined in a simulation class hierarchy, see Figure D.3. The top class, volume, has two subclasses, 'tube-sim' and 'non-tube-sim'. 'tube-sim' corresponds to the pipes in the cell whereas 'non-tube-sim' corresponds to the units. The class names are extended with '-sim' to indicate that these classes are used in the dynamic simulator.

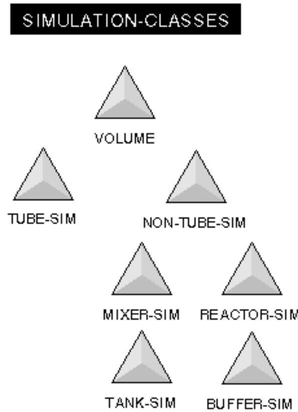


Figure D.3 Simulation classes.

On the subworkspace of each class the equations specific for this class are placed. Equations that are common to all units are placed on the subworkspace of 'non-tube-sim', these are e.g. the equations for the volume, the level, the component-mass-balances, the energy-balances, the total mass balance and the reaction-rates. In Figure D.4 two of the eight equations associated with the class 'mixer-sim' are shown.

To each class a number of attributes are associated. The attributes common to all classes are defined in the class 'volume', whereas attributes specific to only one class are defined within the class itself. Attributes shared by the classes are e.g. the name, the mass fraction of each substance, the volume and the level, the reaction rates of the two reactions and the temperature. Attributes specific to a class are, e.g., the number of inpipes.

When the program is started the equations are calculated and the differential equations are updated periodically. The results are shown as values of the attributes to the objects. In Figure D.5 the simulation view of the

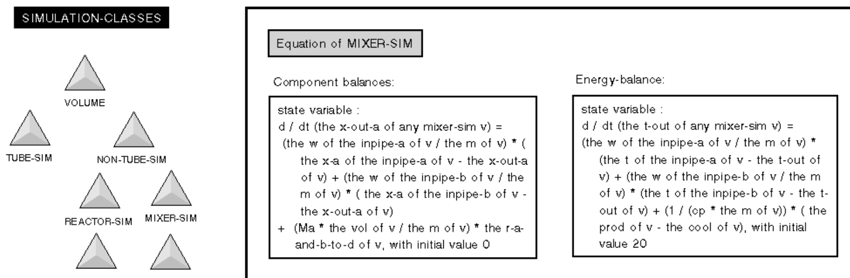
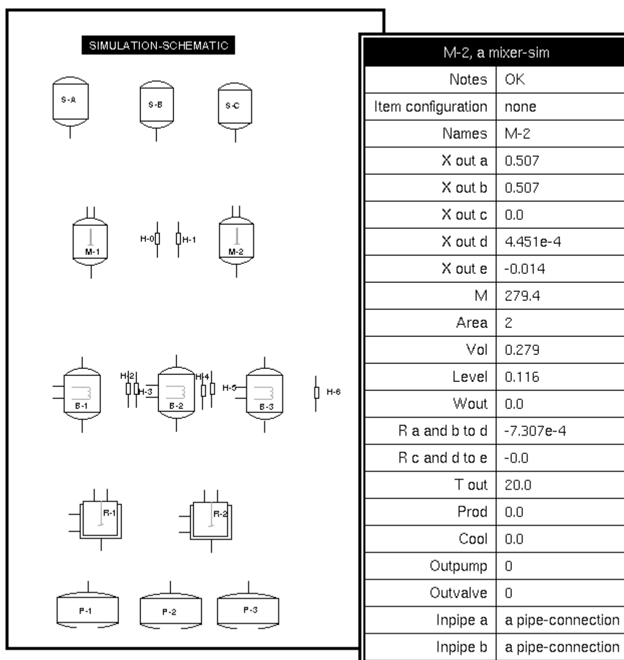


Figure D.4 Equations associated with the class "mixer".

batch scenario is shown together with the attribute table of one of the mixers. The table contains the values of the simulated parameters.



M-2, a mixer-sim	
Notes	OK
Item configuration	none
Names	M-2
X out a	0.507
X out b	0.507
X out c	0.0
X out d	4.451e-4
X out e	-0.014
M	279.4
Area	2
Vol	0.279
Level	0.116
Wout	0.0
R a and b to d	-7.307e-4
R c and d to e	-0.0
T out	20.0
Prod	0.0
Cool	0.0
Outpump	0
Outvalve	0
Inpipe a	a pipe-connection
Inpipe b	a pipe-connection

Figure D.5 Attribute table of the mixer-sim named "M-2".

D.7 Notation and Constants

Notation	Unit		Notation used in Batch scenario
V	m^3	Volume	Vol
W	kg/sek	mass flow	Wout
X	$(kg/total\ kg)$	mass fraction	x-out-(a,b,c,d,e)
M	$kg/kmol$	mole weight	MA,MB,MC,MD,ME
m	kg	mass	M
r	$kmol/(sek \cdot m^3)$	reaction rate	r-a-and-b-to-e etc.
c	$kmol/m^3$	concentration	$(C_A = \frac{\rho x_A}{M_a})$
ρ	kg/m^3	density	density
E	$J/kmol$	activation energy	e-a-and-b-to-d etc
K	$m^3/(kmol \cdot sek)$	reaction-constant	k-a-and-b-to-d etc
R	$J/kmol \cdot K$	gas law constant	r
T	$^\circ C$	temperature	t out
h	J/kg	enthalpy	$(h = c_p \cdot T)$
u	J	internal energy	$(u \approx H = c_p \cdot T)$
Q_p	J/sek	produced reaction rate	Prod
Q_c	J/sek	cooling heat	Cool
ΔH_{reac}	$J/kmol$	—	h-a-and-b-to-d etc
$\mathcal{H}$	$J/(sek \cdot m^2 \cdot K)$	heat transfer	cooling-coefficient
k_w	m	wall coefficient	wall-area-coefficient
c_p	$J/(kg \cdot K)$	heat capacity	c_p

Constant	
c_p	$4180\ J/(kg \cdot K)$
density	$1000\ kg/m^3$
wall-area-coef.	$40\ m$
R	$8314\ J/(kmol \cdot K)$
mol-weights:	
Ma	$50\ kg/kmol$
Mb	$60\ kg/kmol$
Mc	$40\ kg/kmol$
Md	$50\ kg/kmol$
Me	$40\ kg/kmol$
reaction-constants:	
k-a-and-b-to-d	$2 \cdot 10^7\ m^3/(kmol \cdot sek)$
k-c-and-d-to-e	$7 \cdot 10^{10}\ m^3/(kmol \cdot sek)$
activation-energy:	
e-a-and-b-to-d	$69.418 \cdot 10^6\ J/kmol$
e-c-and-d-to-e	$75.00 \cdot 10^6\ J/kmol$
reaction-energy:	
h-a-and-b-to-d	$-6.99 \cdot 10^7\ J/kmol$
h-c-and-d-to-e	$-75900\ J/kmol$

E

An Introduction to G2

G2, developed by Gensym Corporation in USA, was originally developed as a real-time expert system. It has however evolved into a very powerful object oriented programming environment with strong graphical features. G2 is written in Common LISP which is automatically translated into C. However, all user programming is done in G2's built in programming language using either rules, functions or procedures. G2 runs on a variety of UNIX and Windows platforms.

Classes and objects The programming language of G2 is strictly object oriented. This means that objects are implemented in classes defined in a class hierarchy. Each class has an icon and a number of specific attributes. The icon can be defined either textually with the text editor or graphically using the icon editor. The attributes contain the data associated with a class. Subclasses inherit properties from its superclasses, multiple inheritance is allowed.

An application is built up by placing objects (instances of a class) on a workspace. By connecting the objects their relationships are shown. Associated with each object is its attribute table. The table is automatically created from the definition of the object's class.

Composite objects are objects that have an internal structure. As other objects, composite objects are represented by an icon and have an attribute table. The values of these attributes may be other objects. It is, however, not possible to have a graphical representation of the composite object and its internal objects at the same time. If such a representation is desired this can be implemented using the subworkspace concept. In G2 all objects may have an associated subworkspace and on this subworkspace other objects may be positioned, i.e., the internal structure of an object can be represented on its subworkspace.

Rules and procedures G2-rules can be used to indicate how to respond to and what to conclude from changing conditions within the application. There are five different types of rules; if-, when- initially- whenever- and unconditionally rules. Rules can be scanned or invoked in a number of ways.

Procedures are written in a Pascal like language. The roles of procedures are dual, either they can be used as ordinary procedures or they can function as processes. Procedures are called from rules, from other procedures or from user actions, e.g., from buttons. Procedures can have input parameters and they can return one or several values. When a procedure operates as a process each invocation of the procedure execute as a separate task. Since the language of G2 is object oriented it also contains methods. A method is a procedure that implements an operation for an object of a particular class.

Simulator G2 has a built in simulator which can provide simulated values for variables. The simulator allows the simulated expressions to be algebraic equations, difference equations , or first-order differential equations.

G2 provides a powerful graphical interface and easy ways to manipulate and reason about objects. It is particularly well fitted for applications which need graphical representations like the batch scenario. For more information about G2 see [Gensym Coorporation, 1995].

LUND INSTITUTE OF TECHNOLOGY
Lund University

Department of Automatic Control

ISSN 0280–5316
ISRN LUTFD2/TFRT--1051--SE