Fortschritt-Berichte VDI

VDI

Reihe 8

Mess-,
Steuerungs- und
Regelungstechnik

Nr. 1248

Dipl.-Ing. Liyong Yu,
Tianjin (China)

A Reference Model for the Integration of Agent Orientation in the Operative Environment of Automation Systems

ACPLT
AACHENER
PROZESSLEITTECHNIK

Lehrstuhl für
Prozessleittechnik
der RWTH Aachen

A Reference Model for the Integration of Agent Orientation in the Operative Environment of Automation Systems

Von der Fakultät für Georessourcen und Materialtechnik
der Rheinisch-Westfälischen Technischen Hochschule Aachen

zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften

genehmigte Dissertation

vorgelegt von **Dipl.-Ing.**

Liyong Yu

aus Tianjin, China

Berichter: Univ.-Prof. Dr.-Ing. Ulrich Epple
Univ.-Prof. Dr.-Ing. Dr. h. c. Peter Göhner

Tag der mündlichen Prüfung: 14. Dezember 2015

Fortschritt-Berichte VDI

Reihe 8

Mess-, Steuerungs- und Regelungstechnik

Dipl.-Ing. Liyong Yu,
Tianjin (China)

Nr. 1248

A Reference Model for the Integration of Agent Orientation in the Operative Environment of Automation Systems

Lehrstuhl für
Prozessleittechnik
der RWTH Aachen

Yu, Liyong
A Reference Model for the Integration of Agent Orientation in the Operative Environment of Automation Systems
Fortschr.-Ber. VDI Reihe 8 Nr. 1248. Düsseldorf: VDI Verlag 2016.
146 Seiten, 60 Bilder, 2 Tabellen.
ISBN 978-3-18-524808-5, ISSN 0178-9546,
€ 57,00/VDI-Mitgliederpreis € 51,30.

Keywords: Agent Systems – Procedure Description – Service Orientation – IEC 61131-3 – Sequential Function Chart – Process Control – Engineering – Agentensysteme, Prozedurbeschreibung, Dienstorientierung

This work introduces a reference model for automation agents that can be seamlessly integrated in existing process control systems. This model combines the advantages of function block technology, service orientation and agent orientation. Concept decisions about modularity, service-oriented interaction and procedure description provide a base for the development of process control- diagnosis-, model management-, project management- and other agents. The model is implemented in a platform-neutral development environment as a proof of concept and tested with an industrial plant.

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

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

D82 (Diss. RWTH Aachen University, 2015)

Als Manuskript gedruckt. Printed in Germany.
ISSN 0178-9546
ISBN 978-3-18-524808-5

Prefacet

The present thesis emerged from my work at the Chair of Process Control Engineering of RWTH Aachen University during the period from 2008 to 2013.

I would like to express my deep and sincere gratitude to Univ.-Prof. Dr.-Ing. Ulrich Epple for his mentoring, his encouragement and his support during my work as a doctoral student and research assistant at his chair. I also wish to express my warm and sincere thanks to Univ.-Prof. Dr.-Ing. h. c. Peter Göhner for his kindness in taking on the function of second advisor for this work. Furthermore, I am very grateful to Univ.-Prof. Dr.-Ing. Gerhard Hirt for presiding over my doctoral examination. Finally, I sincerely thank Margarete Milescu for her invaluable assistance.

Many people have contributed to the research that is presented in this thesis with their comments and thoughts. For this I thank Lars Evertz, Dr. Reinhard Fuchs, Sten Grüner, Holger Jeromin, Dr. Reiner Jorewitz, David Kampert, Roland König, Dr. Kai Krüning, Sebastian Maurell-Lopez, Dr. Henning Mersch, Tina Mersch, Dr. Martin Mertens, Dr. Gustavo Quirós, Markus Schlütter, Dr. Stefan Schmitz, Andreas Schüller, Sabrina von Styp, Constantin Wagner. I also thank Ursula Bey, Christopher Fleischacker, Ting Guo, Christopher Hense, Huijing Jie, Xinye Li, Tobias Lietke, Vihn Pham, Gregor Rohbogner, Ilya, Schapovalov, Semjon Spitzglus, Martina Uecker for their collaboration at the Chair of Process Control Engineering.

Finally, I owe my loving thanks to my parents Yizeng and Yuanheng, my parents-in-law Heping and Honghui for their love, understanding and constant support throughout my graduate. My special gratitude is also due to my wife Yifei and my Children Haoting and Langting for their continuous loving support.

Aachen, December 2015 *Liyong Yu*

(Ge Wu Zhi Zhi)

To study the phenomena of nature in order to acquire knowledge; to study the nature of things.

from “The Book of Rites - The Great Learning”
Zeng Shen (China, 505-434 B.C.)

Contents

Kurzfassung

Der zunehmende Funktionsumfang der Automatisierungssysteme sowie die steigende Komplexität der Automatisierungsfunktionen stellen dem Systementwickler die Herausforderung, die Modularität, Flexibilität sowie die Autonomie auch der Prozessleitsysteme fortlaufend zu verbessern. Die agenten-orientierte Automatisierung hat ein groes Potential beleuchtet, diese Herausforderungen zu bewältigen und gleichzeitig den Engineeringaufwand zu reduzieren. Klassische Agentensysteme aus der Informationstechnik isolieren in der Regel die Endanwender vom Engineering der Agenten. Zudem sind die Engineeringumgebung, die Laufzeitumgebung sowie die Beschreibungsmitteln für Agenten normalerweise inkompatibel mit den bestehenden Automatisierungssystemen. Die vorliegende Arbeit stellt ein Referenzmodell für Automatisierungsagenten vor, welches eine nahtlose Integration der Agenten in IEC 61131-3 basierende Prozessleitsysteme und ein anwender-zentriertes Engineering ermöglicht.

Das Referenzmodell dient als ein generisches Muster für die Entwicklung von verschiedenen Agenten, z.B. für Prozessführung, Diagnose, Modellverwaltung, Projektierung usw. Je nach der Aufgabenstellung können die autonomen Agenten miteinander interagieren. Auf diese Weise können die Fähigkeiten einzelner Agenten für die Lösung komplexer Aufgaben kombiniert werden. Das Referenzmodell definiert den Ausführungsrahmen der Agenten, ihre Kommunikationsschnittstelle sowie die Beschreibungsmittel für agenten-interne kontinuierliche Funktionen und Prozeduren anwendungs- und leitsystemneutral. Dadurch wird der Engineeringaufwand gering gehalten, während die Interoperabilität und die Wiederverwendbarkeit der Funktionsmodule (d.h. Agenten) des Automatisierungssystems erhöht werden.

Das Referenzmodell ist in einer plattformneutralen Entwicklungsumgebung umgesetzt. Seine Anwendung in einem industriellen Projekt wird vorgestellt. In dem Projekt sind Agenten u.a. zuständig für die operative Prozessführung sowie die automatische Erstellung der Prozessführung und Anlagensimulation. Da in diesem Modell IEC 61131-3 kompatible Ausführungssemantiken und Beschreibungsmittel verwendet werden, können die Endanwender die Agenten eigenständig projektieren. Diese Tätigkeiten können bei Verwendung klassischer agentenorientierter Methoden nur von Experten mit spezieller Programmierschulung und Erfahrungen im Bereich Software Engineering durchgeführt werden.

Aufgrund der Reduzierung des Engineeringaufwands und der Kompatibilität mit den bestehenden Automatisierungssystemen kann das Referenzmodell als Basis für die Integration von Agentensystemen in Prozessleitsystem genutzt werden. Konzeptentscheidungen über dienstorientierte Interaktion, modulare Kapselung von Funktionen und generische Beschreibung der Automatisierungsprozeduren können ebenfalls bei der Entwicklung von Funktionen unterstützen, die nicht von Agenten ausgeführt werden sollen. Auch in diesen Fällen wird erwartet, dass die Engineering-Kosten reduziert und die Flexibilität sowie die Interoperabilität der Automatisierungsfunktionen erhöht werden.

Abstract

The increasing functional range of automation systems and the increasing complexity of automation functions challenge system developers to continuously improve the modularity, flexibility and the autonomy of process control systems. The agent-oriented automation has shown great potential in addressing these challenges while reducing the engineering effort at the same time. Classic agent systems in the field of information technology usually isolate the end users from the engineering of agents. In addition, the applied engineering environment, runtime systems and description methods are normally incompatible with the existing automation systems. The present work presents a reference model for automation agents that can be seamlessly integrated in process control systems based on the IEC 61131-3 standard. Moreover, a user-centralized engineering is allowed.

The reference model serves as a generic model for the development of process control-, diagnosis-, model management-, project management-, and other agents. Depending on the automation task, autonomous agents can also interact with each other. By this means, the capabilities of individual agents can be combined for solving complex tasks. The reference model generally defines the execution frame of automation agents, their communication interfaces and the description methods for continuous and procedural functions within the agents. These aspects are application- and system-neutral. Thus, the engineering effort is kept low, while the interoperability and the reusability of functional modules (i.e. agents) within the automation system is increased.

The reference model is implemented in a platform-neutral development environment. Its applications in an industrial project will be presented. In this project, agents are developed i.e. for the operational process control as well as the automatic creation of process control and plant simulation. Because IEC 61131-3 compatible execution semantics and description methods are applied, the end-user can configure the agents by him-/herself. By using classic agent-oriented methodologies, however, these activities can only be carried out by experts with special training and programming experience in software engineering.

Due to the reduction of engineering effort and the compatibility with existing automation systems, the reference model can be used as a base for the integration of agent systems in process control systems. Concept decisions about service-oriented interaction, modular encapsulation of functions and general description of automation procedures can also assist in the development of functions that are not going to be performed by agents. In these cases, the reduction of the engineering cost and the increment of flexibility and interoperability of elementary modules of the automation system are also to be expected.

1 Introduction

1.1 Motivation

A process automation system is the hardware and software environment for the control and observation of production processes, typically in the chemical and metallurgical industry. Along with the rapid development of automation technologies and computer science, the process industry is facing new challenges at present. Firstly, the present global market challenges the manufacturer to shorten the life cycle and raise the individuation of their product. Automation solutions should be flexibly structured, so that they can be easily adapted for new production tasks. Secondly, the quantity and the complexity of automation functions has increased significantly. It becomes more and more difficult to define all possible production situations in advance. Autonomous functions (e.g. adaptation, learning ability etc.) are necessary. Furthermore, technical personals is changing more frequently than in 1970s. Automation systems should be well-structured and easily understandable, so that the knowledge inheritance during a personnel change and generation change can be simplified. The system intuition can be improved through encapsulation and abstraction of automation functions (e.g. as services).

Under these circumstances, the engineering principle of agent orientation appears to have great potential for coping with the aforementioned challenges. By means of agent orientation, automation functions can be encapsulated in individual agents which are in charge of process control, data archiving, mode management or diagnosis. Automation agents are modular entities and behave autonomously. They can, for instance, recognize situations, sense abnormal changes in the environment and chose operation strategies autonomously. Agents provide services to the user and ensure the achievement of service objectives. The user does not need to master all implementation details. Agents can cooperate among each other and solve complex automation tasks. With their help, the autonomy, flexibility and scalability of automation systems can be significantly improved. Bright practical perspectives can be expected through the further development of automation agents.

The agent-oriented software engineering is one important research area in computer science. Research achievements (e.g. situation recognition, adaptivity and learning ability etc.) and their practical application offer a good basis for the development of autonomous agents in process automation. However, the classic software agents can-

not be directly applied in automation systems. One main reason for this is that classic agents are normally developed for runtime systems and implemented in programming languages (typically JAVA) which are not compatible with automation systems. There is still no appropriate approach for the construction of automation agents and for their integration in automation systems.

The process industry is a rather conservative industry and has special requirements on software applications. During the application of a new technology, existing hardware characteristics, runtime behaviors, programming languages (e.g. according to the IEC 61131-3 standard) should be regarded. Existing best practices (e.g. communication technology, engineering processes etc.) are also worthy of being utilized.

One main goal of the present dissertation is to contribute a reference model for automation agents that can be harmonically integrated into existing automation systems. The aforementioned characteristics of automation systems will be carefully considered. The following engineering aspects will be discussed: encapsulation form and execution frame, communication with the environment, internal structure, registration of services, runtime behaviors and execution models in automation systems.

One central engineering aspect of the reference model, which is also the second goal of the present dissertation, is the selection of appropriate description methods for describing continuous functions and procedural functions. The existing description methods in process automation and in computer science will be evaluated. Their compatibility with automation system and their utility in the introduced reference model of agents will also be analyzed.

1.2 Structure of this Work

The present dissertation is structured as follows: Chapter 2 introduces basics of process automation, model-based software engineering, service orientation and agent orientation. Chapter 3 introduces a reference model which serves as a guideline for the engineering of agents and for their implementation in automation systems. Chapter 4 evaluates existing procedure description methods. Chapter 5 specifies a general procedure description method which combines advantages of the analyzed methods and can be applied as a standard approach for describing most procedures in agents. Chapter 6 presents prototypes of the agent model and of the general procedure description method. Chapter 7 introduces results of case studies in which the models for agents and procedure description were tested. And finally, Chapter 8 provides a conclusion and highlights research directions for future research work.

2 Basics of Process Automation

The present work builds on theory and practice developed in the area of process control engineering. In order to bring relevant aspects into perspective, this chapter gives an overview of the basics of process automation and model-based engineering. Additionally, the idea of service orientation and agent orientation from the software engineering field will be introduced.

2.1 Process Automation System

A process control system (also decentralized control system (DCS)) is a decentralized distributed network consists of various computer stations. Automation functionalities installed in the network can be classified into different hierarchical levels and together compose a so-called *Automation Pyramid* [1, 2], cf. Figure 2.1

2.1.1 Overview

The root of the pyramid is the production process and the plant which are controlled by the upper levels. The *field level* contains components for measurement and control. Typical field devices are sensors and actors. The *control level* (or *supervisory level*) contains Programmable Logic Controllers (PLC) for the control of field devices, and PC stations for engineering, observation and operation. The level of *Manufacturing Execution System (MES)* is composed of various systems and software packages for data management (i.e. Process Information Management System PIMS), product management, material tracking & tracing etc. The *Enterprise Resource Planning (ERP)* level contains a suite of software applications (e.g. SAP) which are applied to optimize business processes and resource usage (e.g. capital, personal).

Classic field devices are connected via remote I/O to the PLC on the control level. PLCs can also be applied on the field level and serve as bus slaves. They can control complex devices such as exhaust handling. Field devices with field bus interface and PLC slaves can be connected to the PLC masters on the control level.

Control logics (e.g. valve control, PID loop) are executed on the PLC. The graphical visualization for observation, operation and engineering is installed on industrial PCs (IPC)(e.g. engineering station and operator station in Figure 2.1). These devices are

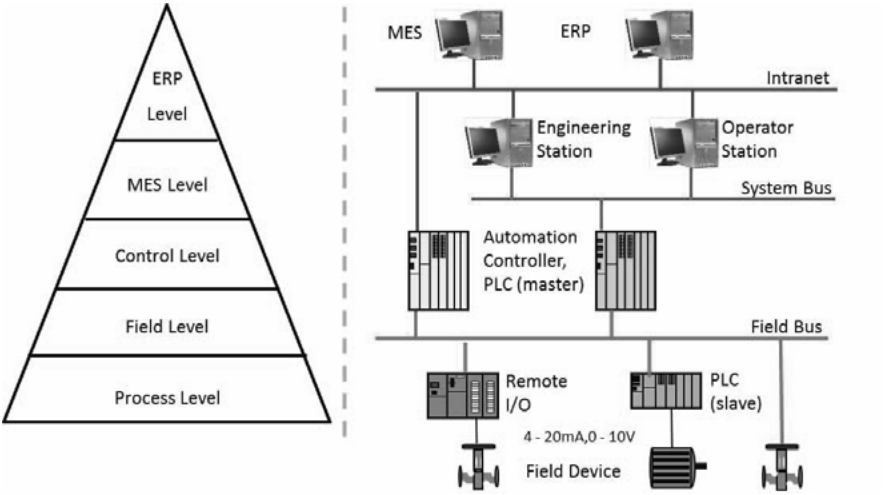


Figure 2.1: Automation pyramid and a schematic Decentralized Control System (DCS)

connected with the PLC masters via system bus. PC-based devices are also applied on the MES level. ERP software is hosted and administrated on the enterprise level and is not always coupled with the DCS network.

The DCS shown in Figure 2.1 is a simplified illustration. The network structure of a real production plant is more complex. Computer stations and sub-networks for high-level functionalities (e.g. advanced process control, backup & restore) are often applied. Additionally, hardware firewalls and interface PCs for the data exchange between two automation levels are often necessary.

2.1.2 Hardware and Software Environment

The pyramid's levels differ not only in terms of functionalities, but also in terms of hardware and software characteristics.

The device number on the bottom of the pyramid is much larger than on the top. A typical chemical plant has thousands of sensors and actors which are controlled by multiple PLCs and PC stations on the control level and MES level. ERP functionalities are managed on the enterprise level, typically in the computer center. It can be assumed for a specific plant, that the ERP applications are installed on one central remote server.

From a computing capacity point of view, devices on lower levels have relatively limited processor performance and data storage. Programs and algorithms on the lower levels are simpler but processed faster compared to the upper levels. Fields devices can be

regarded as embedded systems. They process at millisecond even microsecond level, and have very limited storage for algorithms or process data. Process controllers are typically single-core devices with a typical cycle time of $1sec.$ and possess higher CPU capacity and storage. Operation and engineering stations on the control level are PC-based and apply universal operating systems (e.g. Windows, Linux or Unix). The MES level and ERP level process larger data volumes (e.g. for long time archiving) and apply work stations and servers whose performance and storage volume are theoretically unlimited.

Various runtime systems and programming languages are applied on different pyramid levels. The runtime of field devices and process controllers is characterized by deterministic and cyclic-processing context. The cycle time is normally constant, which is an important prerequisite for most digital controllers (e.g. PID) and signal filters.

Programs on the upper levels have lower requirements on real-time execution and are normally executed periodically. The time span between two program iterations is not always strictly constant. Field devices mostly use a vendor-specific execution context and hardware-near programming languages. PLCs normally support the well-established international standard IEC 61131 [3–5] which specifies among other things:

- PLC runtime model (part 3),
- communication model (part 5),
- two textual programming languages (part 3): *Instruction List (IL)* and *Structured Text (ST)*,
- two graphical programming languages *Ladder Diagram (LD)* and *Function Block Diagramm (FBD)*, and
- one programming languages for procedures: *Sequential Function Chart (SFC)*. SFC can be implemented textually or graphically.

Software applications on the MES level and ERP level are normally vendor-specifically designed and programmed in high-level programming languages such as JAVA and C#.

Further characteristics of hardware and software in process automation can be found in [1, 2].

2.1.3 Trend toward Integration and Standardization

In classic automation systems, the pyramid's levels are strictly separated. Along with the rapid development of process automation and computer technology, the differences and borders among the levels are getting more and more blurred in modern process automation.

Firstly, the hardware differentiation among the levels is gradually eliminated. The performance of embedded systems, microcontrollers and PLCs grows by orders of mag-

nitude. Modern PLCs can nowadays also process complex algorithms which was only possible on hardware that was used on the upper levels. Additionally, the dependency of software on hardware is continuously decreasing. Devices with universal operating systems (e.g. soft-PLC with Windows or Linux) are continuously spread. Modern computer technologies like virtualization and cloud computing are applied or being intensively discussed in communities. Due to the increasing hardware performance and independency, even more functionalities and hardware will no longer be exclusive to a certain level of the pyramid. For example, process control logics which are traditionally executed on the control level, can nowadays also be distributed into field devices with the Fundation Fieldbus technology.

Secondly, the requirements on integration and interoperability of automation systems has increased significantly in the last years. Automation functionalities on different systems and pyramid's levels are required to be interconnected more closely. One example for this trend is the batch control. In classic DCS, production requirements (amount, quality, time etc.) are given by ERP tools, e.g. SAP; batch recipes are mostly manually - parameterized and executed in DCS; production data are achieved on the MES level. The call for continuous improvements in production efficiency has led to more intense requests for a further integration of ERP-, MES- and DCS-functionailities in recent years.

2.2 Modelling

Along with the increasing complexity of automation functionalities, a software engineering principles was required which can help developers and users to gain a better understanding of the automation systems and applications. Under this circumstance, model-driven development of automation functionalities has grown to an important research area in industrial automation. In the following sections, the basics of model-driven software engineering in process automation will be introduced. Chosen example models will be shown which provide a basis for the development of the agent model in Chapter 3 and for the procedure description method in Chapter 5.

2.2.1 Basics

Systems (e.g. DCS, production plant) or processes (e.g. chemical reaction, engineering process) in the real world often need to be described from different perspectives. For instance, the representation, attributes and behaviors of the same production plant are normally different in different application context (e.g. simulation, control, functional description, alarm monitoring etc.). The description from a certain perspective (or aspect) can be regarded as a model of the system or process.

According to [6, 7], the term model is formally defined as *a depiction of a system or process in another conceptual or concrete system which is obtained on the basis of the application of known legitimacies, an identification or assumptions and which displays the system of the process with sufficient accuracy with respect of the selected questions*. As the definition implies, models are conceptual, generalized, theoretic, not representational and always involve a specific context or aspect. The process towards creating a model is called *modelling* which can be regarded as synonymous with *abstraction*. Typical models in process automation are:

- mathematical models for process dynamics (e.g. chemical reaction),
- Piping and Instrumentation diagrams (P&ID [8]) for the representation of the interconnection of installed equipment of a plant,
- hierarchical models for recipes, procedures and processes; these models describe batch production according to the IEC 61512 [9] and
- hierarchical structure model for plants and instruments [9, 10].

A model can be described in different ways, e.g. mathematical equations, natural languages or formal description methods (e.g. class diagram according to the UML specification [11]).

Aside from the depiction with models, models and modelling concepts themselves are also subjects of the research works. In software engineering, the following terms are important for the construction and implementation of models:

Meta Modelling is an abstraction of model. The product of a meta-modelling process is called meta model which is a “model of a model”. A meta model specifies elementary components, associations and rules of a model. According to the MOF specification [12], models can be classified into four abstraction layers: instance (e.g. a valve control unit “Y014”), model (e.g. “on/off valve”), Meta model (e.g. class, operation, attribute) and Meta-meta model (e.g. meta-class, meta-attribute, meta-operation).

The **Core Model** describes fundamental facts and circumstances of a system or process. A core model is generic, domain-neutral and unique. It identifies the generally accepted “truth” that is sustainable and has no more valuable alternative variant. For instance, basic construction and runtime behaviors of process automation applications are generically and formally defined in a core model [13].

Reference Model: As the name implies, a reference model is a reference or design pattern for developing specific models for an application area or domain. In comparison with application- or domain-specific models, a reference model is neutral and is to be designed as the most appropriate model variant or design pattern. It gives a guideline for the construction and development of frames, rules as well as constraints for the treatment of certain tasks in different application areas. In contrast to the unique core model, a reference model may have alternative versions and even counter examples.

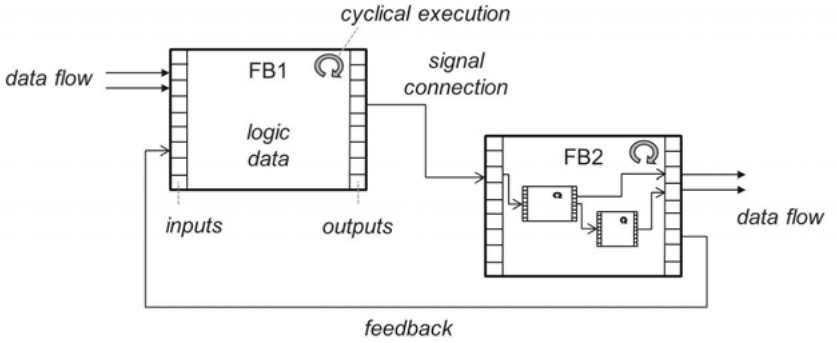


Figure 2.2: Function blocks according to the IEC 61131-3

2.2.2 Function Block

The function block is a central modeling principle in industrial automation. The history of the function block technology can be tracked back to the 1960s where control circuit elements (e.g. relay, timer and PID-controller) were gradually replaced by software modules on Programmable Logic Controllers (PLC). Every function block encapsulates internal logics and possesses input variables and output variables. Function blocks can be connected to each other via signal connections which map electrical wires and model the data flow. The modular and software-based solution on PLCs allows a flexible manipulation and extension of control logics without hardware changes.

Function blocks are also well-established in further application areas, e.g. dynamical modelling or simulation. Typical automation tools and modelling environments that apply function blocks are MATLAB/Simulink, LabVIEW, Modellica, SIEMENS/SIMIT, WinMOD etc. A meta-model of function blocks will be introduced in Section 6.1.1. In the following sections, the application of the function block technology in two automation standards will be introduced.

Application in IEC 61131-3

The function block is defined as an elementary *Program Organization Unit* (POU) in the IEC 61131-3 [3] (see also Section 2.1). The standard has specified among other data formats, communication interfaces and runtime behaviors of function blocks. Additionally, a set of standard function blocks (e.g. addition and multiplication) have also be defined as standards.

Function blocks (FB) can be linked to each other via signal connections and compose a Function Block Diagram (FBD). FBDs may nest subordinated FBDs. One example is given in Figure 2.2

In process automation systems, FBDs are usually implemented in the form of Continuous Function Chart (CFC) which is an extended variant of FBD. In difference to FBDs, CFCs allow a flexible positioning of function blocks. Moreover, the execution priority of function blocks within a CFC can be freely defined, whereas an FBD has to follow the priority rule “from top to bottom, from left to right”. Furthermore, CFCs allow feedback connections (cf. Figure 2.2) which are not supported in classic FBDs. These three differences between FBD and CFC are not essential and supported by FBD editors of all major vendors of process control system. As a result, these two graphical languages are usually regarded as synonymous in practical use.

Application in IEC 61499

The IEC61499 [14] defines the model and usage of function blocks for distributed control logics. As shown in Figure 2.3, IEC 61499 function blocks support *signal flow* and *event flow*. Function blocks can be distributed on different devices in the network and communicate to each other via events.

In contrast to the cyclical execution according to the IEC 61131-3, the execution of IEC 61499 function blocks is driven by events. An active IEC 61131 function block will be permanently executed, although its inputs do not change in value. On the contrary, an IEC 61499 function block starts to execute, only when an event input is received. Every function block has an internal execution control chart (ECC) which controls the execution of the internal algorithms, the read/write of inputs/outputs and the generation of output events for further function blocks.

IEC 61499 is intensively discussed in the academic community. Among others, distribution, reusability, probability and interoperability are identified as the main advantages [15]. However, this standard is still mainly being promoted by academics but not well accepted by industrial users. One main reason is that the ability of distribution is not required by the most applications, but requires higher complexity of the runtime system. Additionally, the last three addressed advantages can also be largely fulfilled by classic IEC 61131-3 function blocks. Furthermore, there is still no generally accepted event model. Although the IEC 61499 is not sustainable for replacing IEC 61131, the event-driven execution concept is worthy of being regarded in the further development of function block technology [15].

2.2.3 Runtime System Model

The description of runtime systems is important for the development of models and applications in process automation. [16] has introduced a unified model that can be universally applied for describing and modelling runtime systems on the field level, control level and the MES level. Due to the application-neutral and vendor-neutral nature, this model is taken as the terminology basis for the further discussions in the present work.

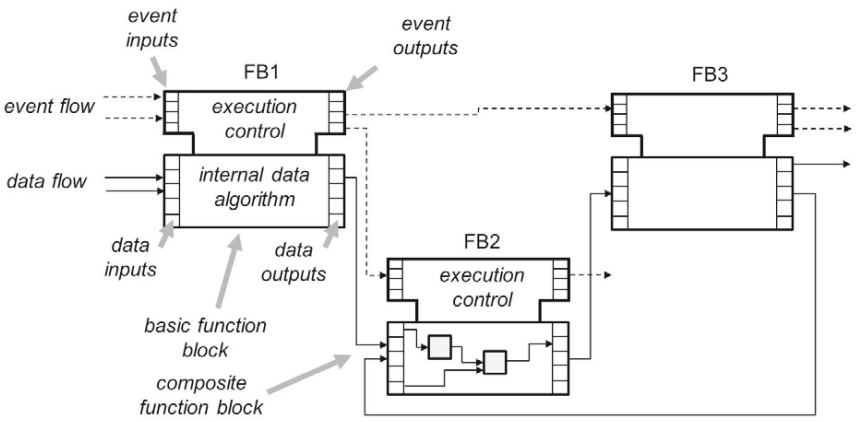


Figure 2.3: Function blocks according to the IEC 61499

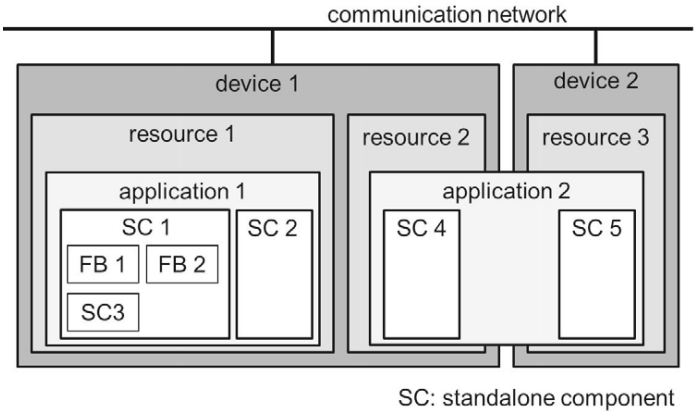


Figure 2.4: Model components of the unified runtime system (Figure according to [16])

An example system according to the unified runtime model is given in Figure 2.4. Key elements of the model will be introduced below:

- *Function Block* is the smallest and elementary unit for organizing program execution. In this model function blocks are generically defined and do not restrict the execution behavior in runtime. A function block can be executed cyclically (according to the IEC 61131-3) or event-driven (according to the IEC61499, cf. Section 2.2.2).

unified runtime system [16]	related Terminology in standards	
	IEC 61499	IEC 61131
device	device	resource
resource	resource	task
application	application	program
standalone component (SC)	–	
function block (FB)	function block	function block
function	function	function

Table 2.1: Overview of Terminology for runtime systems

- *Standalone component (SC)* is a further component for organizing programs. SCs may contain function blocks and SCs. SCs possess communication interfaces (e.g. signal inputs and ouputs) for the information exchange with further SCs. As the name implies, an SC can theoretically be executed, irrespective of the execution of further SCs. To ensure the safety of the production, a strictly deterministic execution within a Standalone Component (SC) is worthwhile.
- *Application* provides a logical name space for a set of standalone components. An application can be distributed on one or more resources that are not strictly present on the same physical device.
- *Resource* (or *Server*) represents a thread-safe execution environment for a set of standalone components. Resources are independent to each other and theoretically run simultaneously. A resource cannot be distributed on different physical devices. A standalone component can be located only on one resource.
- *Device* represents the entire hardware- and virtual world on a node in the automation network. A device may contain more than one resources.

In practical use, the terms *device* and *server* are often regarded as synonyms. Due to the widespread “server-client-model” from computer science, many designations like “engineering server”, “operation server”, and “batch server” also involve the host hardware, on which resources and applications are installed. In the present work, the term *server* only involves *resource* but not *device* unless otherwise noted.

A comparison of the unified runtime model with related definitions in the IEC 61131 and IEC 61499 is given in Table 2.1. Detailed discussions on the model elements and their relationship can be found in [16, 17].

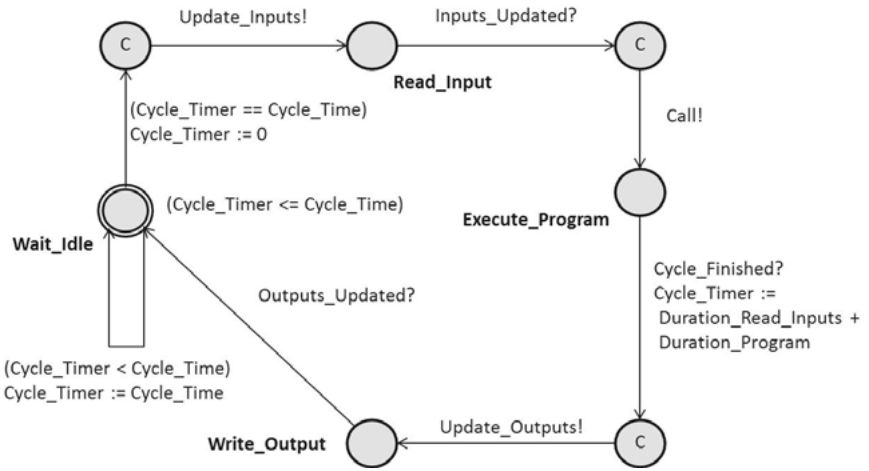


Figure 2.5: Cyclic execution context on an Programmable Logic Controller (PLC) or on an Standalone Component (SC) according to [16]. (Figure according to [18, 19])

2.2.4 Time Model of Cyclic Execution Environment

The cyclic-processing runtime of automation systems can be modeled as automata according to UPPAAL [18, 19]. UPPAAL is a software environment which allows an intuitive graphical representation of runtime behaviors, relationship and interactions among system components. Additionally, algorithms for model validation and verification can be conveniently developed with the UPPAAL's automata.

In UPPAAL, a runtime system is modeled as a network of automata. Every system element (e.g. function block, server, device etc.) may have an exclusive automaton. All automata describe behaviors in continuous time. They are standalone and theoretically run in parallel. However, in an execution context without parallelism, e.g. on a single threaded Programmable Logic Controller (PLC), only one automaton can be executed at a certain time. The other automata should wait, until the runtime system is idle again.

Figure 2.5 shows the UPPAAL automation of a cyclic execution context according to the IEC61131-3 [3]. This model can be applied to describe the execution of a PLC or a Standalone Component (SC) introduced in Section 2.2.3. UPPAAL automations communicate among each other via binary communication channels. An outgoing signal from one automaton on a channel is labeled $channelname!$. An incoming signal from a channel is labeled $channelname?$, respectively.

UPPAAL automata are composed of states and transitions. Transitions may have guards (expressions in brackets) and may perform actions (expressions without brackets). The initial state is marked with a double-lined border. Theoretically, committed

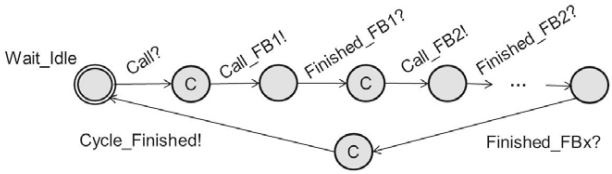


Figure 2.6: Root task of an Standalone Component (SC, compare Figure 2.4)

states (state with the label C) and transitions have no duration, whereas non-committed states (empty cycle) have time consumption.

The execution order of all internal components of an SC is defined by a *Root Task*. All function blocks on the SC are defined as task's children of the root task. The root task will be executed cyclically.

In case the automaton in Figure 2.5 is in its initial state (i.e. the idle state $Wait\_Idle$) and the time span reaches the pre-defined cycle time, the automaton updates inputs; emits the signal $Call!$; waits for the committed signal $Cycle\_Finished!$; updates outputs of the PLC afterwards; and returns to the idle state at the end. As the $Call!$ signal is emitted in a fixed time rate, the root task (i.e. the task automata in Figure 2.6) is triggered cyclically.

Figure 2.6 shows the UPPAAL automaton for a root task of an SC. In case the root task receives a $Call?$ signal from the SC (compare also Figure 2.5), the execution of the task is triggered. $Call$ signals are emitted to the task's children that will be executed sequentially. In the example root task, x function blocks (FB) are invoked as task's children. After a task's child is executed for one iteration, a $finished$ signal will be sent back to the root task. When all task's children are finished with their execution, the root task emits a $Cycle\_Finished!$ signal and returns to the idle state.

2.2.5 Model of Operational Resource and Operational Measure

Process control is the central operation of process control systems. [1] has introduced a *Resource Measure Model* which describes hierarchical levels of heterogonous process control functionalities and the relationship among process control modules. As shown in Figure 2.7, control modules can be classified into three groups:

- *Single Control Units (SCU)* are software representatives for individual actors (e.g. valve, ventilator etc.). Every unit is in charge of the downward interaction with the real device in the field level, and for the upward interaction with superordinate control modules and human users.
- *Group Control Units (GCU)* coordinate groups of control units (SCUs or GCUs). Thus, a hierarchical control structure can be built. For instance, a pump control

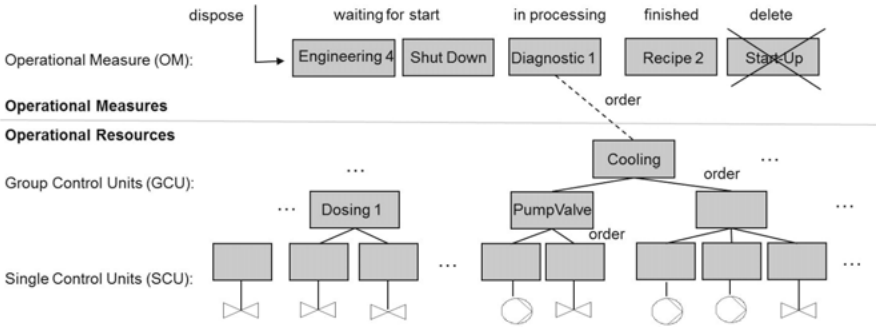


Figure 2.7: Resource measure model for operative process control

unit and a valve control unit can be controlled by a GCU "pump-valve group" which coordinates the two SCUs and realizes a flow control. The GCU can be controlled by another GCU which controls the whole water cooling system.

- *Operational Measures (OM)* represent specific procedures for production, start-up, shutdown, diagnosis etc.

SCUs and GCUs set up the level of *operational resources*. They represent the intrinsic process control functions of a plant, exist in the control system permanently and are theoretically always active. The allocation relationship among SCUs and GCUs is usually fixedly defined.

Operational measures (OM) realize production tasks by using operational resources. OMs can be created and disposed dynamically. They allocate operational resources temporarily and set them free, when the execution is finished. All operational resources that are to be assigned are specified during the disposition. All OMs have the same life cycle: disposed, waiting to start, processing, finished and deleted. The allocation relationship between operational resources and operational measures is normally built up dynamically.

2.2.6 Component Model for Hierarchical Process Control

The resource-measure-model in Section 2.2.5 specifies the general structure of process control functionalities. For the construction and engineering of individual process control units, a design concept named ACPLT/PF[1] has been proposed in [20]. ACPLT/PF defines among other things a reference model for control modules and the interaction behavior among them. This work has been followed by different research works [20–24] of the Chair of Process Control Engineering. Relevant discussions will be summarized and introduced in condensed form in the following paragraphs.

[1] Prozessführung: German expression of process control

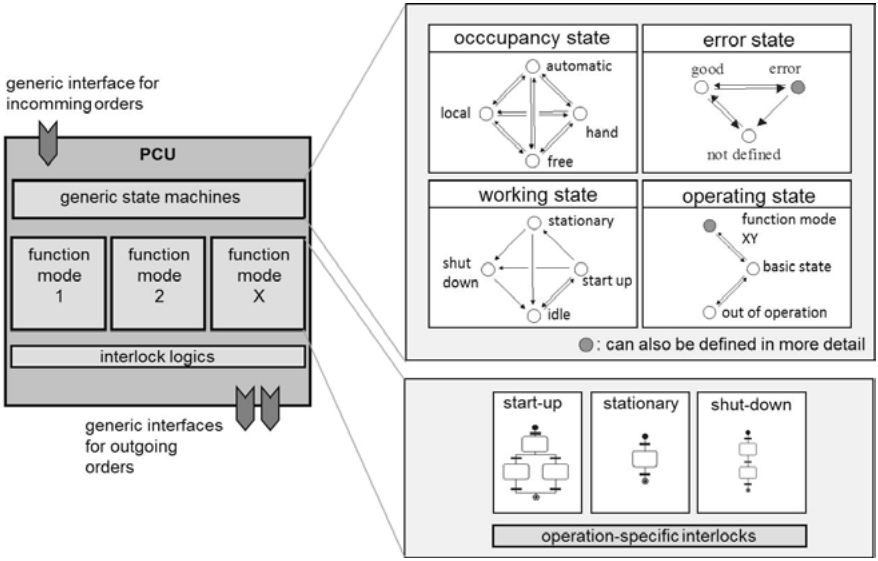


Figure 2.8: Component model of Process Control Unit (PCU)

According to ACPLT/PF, all Process Control Units can be designed with the standard frame shown in Figure 2.8. Every PCU encapsulates *Function Modes* (FMs), and has four standard state machines for controlling and monitoring the execution of function modes (cf. Figure 2.8).

Every PCU can be dynamically occupied by another PCU (occupancy state automatic) or the operator (occupancy state $hand$ or $local$). Execution orders from requesters other than the currently valid occupier are rejected.

Within the PCU frame, only one function mode can be active at any given time. Every function mode is composed of three procedures: start-up, stationary and shut-down.

To enable and standardize the interaction among PCUs, as well as the interaction between the operator and PCUs, two generic order interfaces have been defined. The incoming interface is responsible for receiving process control order from superordinate PCU or the operator, whereas the outgoing interface sends orders to further PCUs. Order interfaces are designed to interpret process control orders with the following standard form:

order sender; order receiver; order; parameter

The first three entries are obligatory, whereas the parameter can be optionally defined according to the specific process control order.

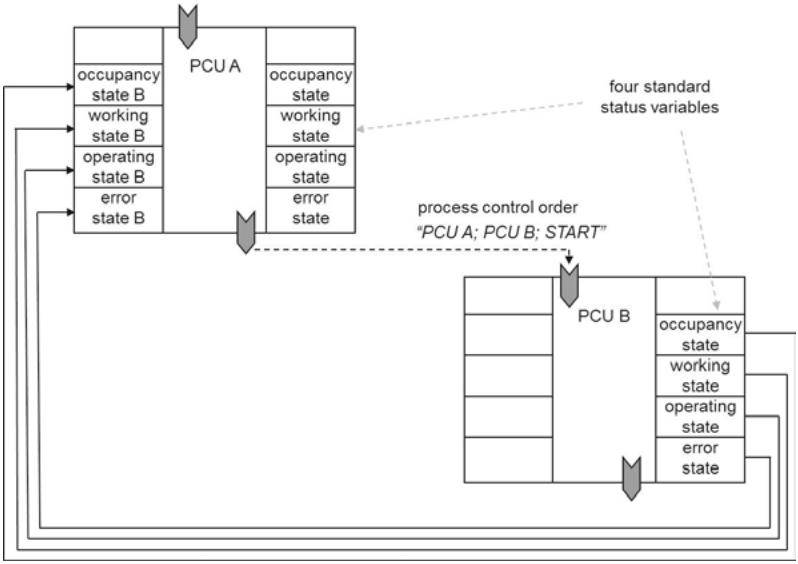


Figure 2.9: Communication between two Process Control Units (PCUs)

Similar to traditional signal connections, the order communication between two PCUs is also unidirectional. As shown in Figure 2.9, the superordinate PCU A sends a process control order to its subordinate PCU B, whereas PCU B does not send back any reply. Information about the occupancy and the progress of the execution is controlled by the four generic state machines in PCU B and saved in the form of four standard output variables. In order to inform about the current state information, PCU A can read the four variables via signal connections.

In comparison with conventional process control solutions, ACPLT/PF provides two main improvements. Firstly, a standard model for designing process control units has been defined. It formulates a guideline for designing encapsulation frame, order interfaces, state machines and variables representing execution states. Multiple signal interfaces for receiving control instructions are replaced by a general order interface. Secondly, the loosely-linked order interfaces allow a flexible interaction between PCUs. Every PCU can be loosely coupled in the system and be dynamically occupied by an execution requester which is not defined in advance. Moreover, the standard order format can significantly simplify the engineering. Users can concentrate on the order content and do not need to implementation details, such as, which variables of the individual PCUs should be set and which syntax and semantics should be considered.

ACPLT/PF is generally and platform-independently defined. A prototype is implemented in the development environment ACPLT/OV which will be introduced in Section 6.1.1.

2.3 Service Orientation

Service orientation is an architectural paradigm for software engineering and has received extensive attention in process automation (cf. [25, 26]). According to OASIS [27], service orientation is used for *organizing and utilizing distributed capabilities that may be under the control of different ownership domains*.

The basic idea of service orientation was inspired by the management of business processes among various departments within a company. Elementary services (e.g. procurement, logistics, IT technology support) are provided by departments and can be utilized as part of more complex services. The service user or service caller is the *service requester*, whereas the supplier plays the role of *service provider*. A service requester defines the functionality and entrusts one or more service providers with the realization.

In technical fields, service orientation is suited to abstract implementation details and to standardize communications among heterogeneous and loosely-coupled systems. As discussed in [25, 26], distributed automation functionalities can be provided as services which possess clearly defined interfaces and support system-neutral communication over networks. As shown in Figure 2.10, automation functionalities can be abstracted as services (ellipses) in a virtual service system. Services can be flexibly accessed and invoked. One service can use further services in the system.

For instance, data and configurations of automation systems often need to be archived. In traditional automation systems, various servers or software packages are installed on different systems to realize the functionality “archive”. Heterogeneous interfaces have been developed for accessing these archives. The effort required for engineering and maintenance is usually very high for these specific solutions. By means of service orientation, a service “archive” with an abstraction of implementation details and a well-descripted interface can be defined and utilized by service requesters, irrespective of their system and pyramid’s level. With this design, the usability and reusability of automation functionalities can be significantly improved. [26] has presented a core model of service orientation for process automation. The relationship between service, provider, requester and service description is explicitly defined (cf. Figure 2.11)

The call mechanism of services is very similar to the function call in traditional programming languages (e.g. ANSI C), but has some essential differences. Firstly, traditional functions are performed mostly on a local server, whereas service requests can be sent across networks and can be fulfilled on a remote server. Secondly, the execution

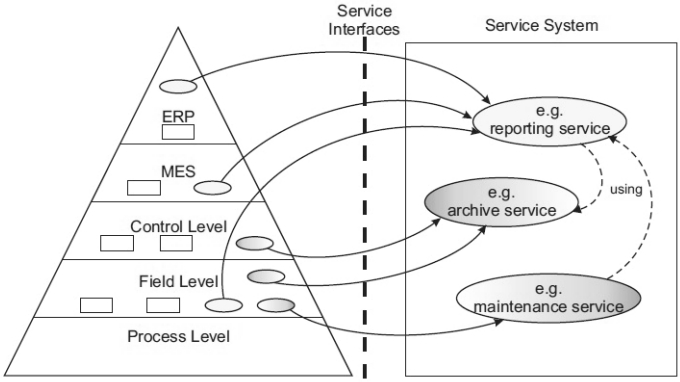


Figure 2.10: Service-orientation for process automation. (Figure 2 in [28])

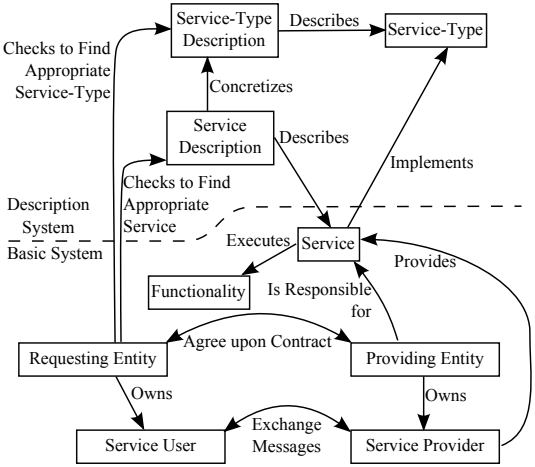


Figure 2.11: A reference mode for service-orientation in process automation. (Figure 4 in [26])

of a traditional function caller is normally blocked until the desired function is completely finished, whereas service requester and provider are designed to execute independently from each other. In other words, a service requester does not need to interrupt and block its execution until the execution result is send back. Furthermore, services should be able to describe themselves and can be explored. As introduced in [26, 29], every service should possess a data model which contains information about service characteristics, interfaces, data, as well as contracts and policies for using. These contents should be structured and machine-readable.

In classic signal-oriented communication, a signal normally only contains the data that is to be transmitted (e.g. value $TRUE$ for the control signal $VALVE\_14\_OPEN$). However, the content of service requests is more comprehensive. A service request contains at least the requester address, receiver's address and service identity. Further parameters specifying execution details and results are also required.

One central application of the service orientation in process automation is the OPC UA which is specified in the IEC 62541 standard [30]. OPC UA forms a concept of unified communication among various automation systems. The standard has defined among other things a communication model and two example services: read variable and write variable.

OPC UA is still not widespread in process automation. One reason for this is that the definition in the IEC62541 standard is conceptual and abstract. For various application areas in the automation system, standard services for specific application areas, vendor-neutral data models and data formats for service-oriented interactions are still to be defined. Another main reason is that DCS vendors address the integration of own systems and software solutions with vendor-specific communication technology (i.e. compare Siemens Simatic IT, ABB Enterprise Connectivity). Specific use cases are to be identified by users, so that vendors can be convinced to open their system and support vendor-neutral communication.

On the basis of OPC UA, the recommendation NAMUR NE 141 [31] has specified typical services (e.g. read history, browse attributes of a batch recipe etc.) for the communication between batch packages on the control level and MES level. However, this recommendation is still not implemented in commercial DCS.

2.4 Messages

A message is a discrete unit of information sent from one communication party to another. It can transmit data (e.g. temperature value, execution state) or represent a service request.

Messages can be described in different ways. One possible implementation approach is *Value-List*, in which all message contents are arranged in series and separated with special symbols. For instance, the process control command according to ACPLT/PF (cf. Section 2.2.6) is a Value-List message which is realized as a composed string. Message contents, e.g. sender and receiver, are separated by semicolons. Value-List is a compact message format which can reduce the storage space and communication bandwidth. However, both the sender and the receiver have to master the message syntax and the semantics of the entries. All contents must be defined and arranged according to a fixed order.

An alternative message format is the so-called *Attribute-Value-Pair*, in which every content consists of an attribute (e.g. $setpoint$) and a value (e.g. 5.3). All values in an Attribute-Value-Pair are self-identified. They can be flexibly grouped, hierarchical structured, and flexibly arranged.

A further development of the Attribute-Value Pair is the Extensible Markup Language (XML) [32]. A typical XML-message contains contents and additional information about the message type, coding, time stamp etc. It can be accurately interpreted both by machines and human users. Additionally, an XML message can contain a reference to a *schema* which specifies the structure, grammar and content of the message. According to this schema, incoming messages of a receiver can be formally evaluated and correctly interpreted. XML is widely used to realize a platform- and implementation-neutral communication across networks. XML is established for instance in the realization of web services. Many software agents (cf. Section 2.5) also apply XML-based message formats, e.g. KQML [33]. An example XML-message will be introduced in Section 3.3.

2.5 Agent Orientation

This section introduces the design paradigm of agent orientation which provides a new approach for software engineering in process automation. After an introduction of the theoretical basis in Section 2.5.1 and Secion 2.5.2, a conceptual reference model for automation agents will be introduced in Section 2.5.3.

2.5.1 Introduction

The word agent is derived from the Latin “agree” which means “to do”, “drive” or “act”[2]. In general, agent describes an entity which can be a human being, a trained animal, a machine or a computer program. An agent serves as an intelligent delegate of the client and is able to accomplish certain tasks autonomously. According to the VDI/VDE 2653 recommendation [34], a technical agent is defined as:

An encapsulated (hardware/software) entity with specified objectives. An agent endeavors to reach these objectives through its autonomous behavior, in interacting with its environment and with other agents.

According to [34–36], a technical agent should possess a subset of the following primary characteristics:

- *Autonomy* allows an agent to control its internal state, objective-oriented behaviors and decision-making.

[2] In German: “agieren”, “treiben” or “handeln”.

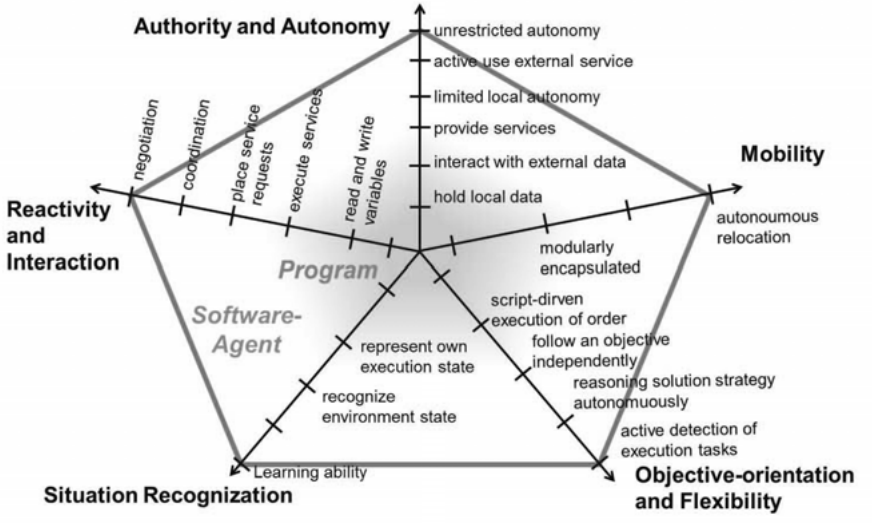


Figure 2.12: Differences between programs and automation agents (Figure based on [35, 38])

- *Encapsulation* requires an agent to encapsulate states, behaviors, strategies and objectives which are not visible externally. An agent must be an enclosed functional unit with completely specified interfaces for the information exchange with the outside [36, 37].
- *Persistence* describes the capability of keeping internal states during the life cycle of an agent.
- *Reactivity and Interaction* enables an agent to sense the environment, to generate reaction, and to interact with other communication partners.
- *Scope of action* limits the autonomy of an agent.
- *Mobility* allow an agent to move from one location (e.g. a PC on which the agent is executed) to another.

Further characteristics such as proxy, rationality, and veracity are also intensively discussed in different research works. An overview of related opinions and theoretical analysis can be found in [38–40]. Since these characteristics are application-specific and have less general meaning, they are regarded as secondary characteristics in the present work and will not be discussed in depth.

The primary characteristics of agents are not quantitatively defined. The abstract definitions have not clearly defined the difference between agents and non-agent entities. In software engineering contexts, it is often confused whether a program can be regarded

as an agent. The scales in Figure 2.12 are specified on the basis of related discussion in [35, 38]. They can be applied to differentiate automation agents from classic programs.

In general, all software agents are programs, but not all programs are agents. There exists also no sharp distinction between agents and traditional programs. Classic programs have a low degree of properties (middle part of the diagram) as depicted in Figure 2.12. Programs with a high degree of properties from the middle of the axes can be regarded as agents. The outermost degrees of the diagram can be seen as the ultimate goals of artificial intelligence.

The ability of one single agent is limited. Multiple agents often work together and compose a so-called *Multi-agent-System* (MAS). An MAS can be regarded as a society of autonomous agents. Single agents play specific roles (comparable with soccer players assigned to different positions on the field). Complex missions (comparable with soccer matches) can be divided into elementary tasks which are solved by individual agents (comparable with defender, midfielder and forward). Agents accomplish missions through different combinations and strategies (comparable with soccer formations). In addition, special agents can be employed which play the role of facilitator and are responsible for coordination works, such as broking, matchmaking, recruiting of service provider, broadcasting of information etc. (comparable with soccer coaches and referees).

In technical fields, agent-orientation was originally a branch of the research area of Artificial Intelligence (AI). It has been developed nowadays into an important modeling paradigm which is utilized in different research and application areas. For example, [41] has presented an approach, in which consumer's buying behaviors (e.g. need identification, product brokering, negotiation etc.) in e-commerce are automated by different agents. In urban traffic networks, agents can be applied to control traffic lights, sensors and further facilities, so that travel time and congestion can be reduced [42–45]. In areas of power supply, agents are suited to optimize the supply and consumption among distributed power resources (e.g. wind park, solar farm, fuel cells) and consumers [46–48]. In the area of software engineering, many researchers (cf. [49]) refer to agent-orientation as the next dominating modeling principle after object-orientation. By means of object orientation, entities are modelled as objects, whose capacity is represented as type-specific attributes and methods. As an extension of this modeling principle, agent orientation defines advanced compatibilities that an intelligent entity or object should possess.

The research on agent orientation involves two main aspects: knowledge and communication technology. A knowledge base allows an agent to sense its environment, recognize situations, deduce solutions, and reach execution objectives autonomously. Communication technology allows agents to interpret messages and to coordinate among each other or with human-users. One central direction in this research area is the development of a machine language in imitation of human languages. Research on the agent language involves speech-act theory, ontology, role play, communication language and

protocol. Various approaches like KQML [33], FIPA-ACL [50] and JADE [51, 52] have been developed in recent years. These studies allow agents to deal with different situational dialogs such as delegation, rejection, negotiation etc.

2.5.2 Usability in Industrial Automation

Many agent-oriented software solutions have been developed for industrial automation in recent years. Some of them have already been tested in practical use. [53] presented an approach for data collection and interpretation in Decentralized Control Systems (DCS) and Computer-Aided Engineering (CAE) systems. [54, 55] have proposed different self-management functionalities (healing, optimization, configuration, protection etc.) that can be autonomously managed by agents. [56, 57] have introduced an approach in which sensor measurement can be substituted through correlated sensors in the case of sensor failures.

Aside from the applications in specific areas, [35, 58, 59] have discussed the usability of agent-orientation in the modelling of automation functionalities in general. By means of agent orientation, the process control system can be constructed as a Multi Agent System (MAS) where autonomous agents are in charge of different tasks. Different automation objectives can be achieved through agent-agent and agent-human collaboration.

As shown in Figure 2.13, agents can be applied to achieve various operation objectives in automation systems. Entities (e.g. product, equipment) or functionalities can be represented as process automation agents. Process control agents control and observe field devices (e.g. pump) or device groups. Measure agents are in charge of operational measures for production, product transport or diagnostic tasks etc. System agents manage distributed computing resources for warning, protocolling or archiving and provide them as common systems services. Agents can also provide services for engineering, asset management and model management.

The proposed use cases focus mainly on the control level and parts of its two adjacent levels (cf. the area with shadowed background in the automation pyramid in Figure 2.13). However, the discussion on agent-oriented engineering is generic and level-neutral. Many use cases (e.g. archiving) are also suitable for further automation levels.

All agents are modularly encapsulated. They possess a knowledge base, behave autonomously, provide services to the outside and possess a unified service interface. Services can be explored and invoked dynamically. Execution objectives can be defined in form of service requests (e.g. “archive a value”). Agents can interpret the requests and achieve the objectives on behalf of users. Implementation details (e.g. detection of archive location, configuration for the data transmission) do not need to be known by users.

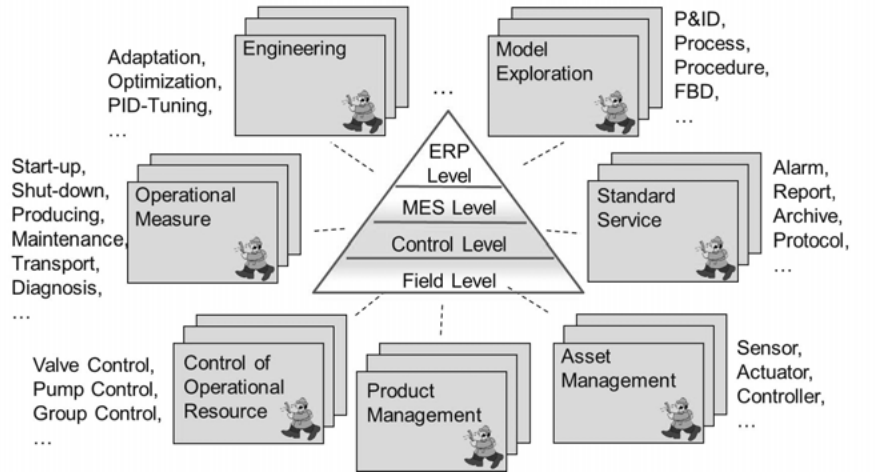


Figure 2.13: Process automation agents (Figure on base of discussions in [58, 59])

Agent orientation can improve the flexibility of process control systems. In classic automation systems, single automation functionalities normally have heterogonous communication interfaces, and are usually rigidly coupled. All possible combinations of functionalities have to be fixedly projected in advance. An extension involves often high engineering cost. However, agents can cooperate to each other and solve complex tasks together. An agent provides services to the outside and can request services from further agents. The cooperation relationship between agents can be dynamically built.

Additionally, the usability of automation functionalities can be improved by agents. In traditional process control systems, operational resources for realizing system functionalities are signals and communication interfaces. In order to control a valve for example, users need to acknowledge which software module is to be applied, which variable(s) should be set so that the valve can be opened and which feedback signal(s) should be checked. For the automation of complex plants, the engineering workload of users is high. By means of agent orientation, however, agents offer the achievement of objectives (e.g. services) as operational resources for users. Agents serve as intelligent delegates of users and can achieve operational objectives autonomously. Users invoke services provided by agents and do not need to master implementation details and heterogeneous communication interfaces of heterogeneous systems.

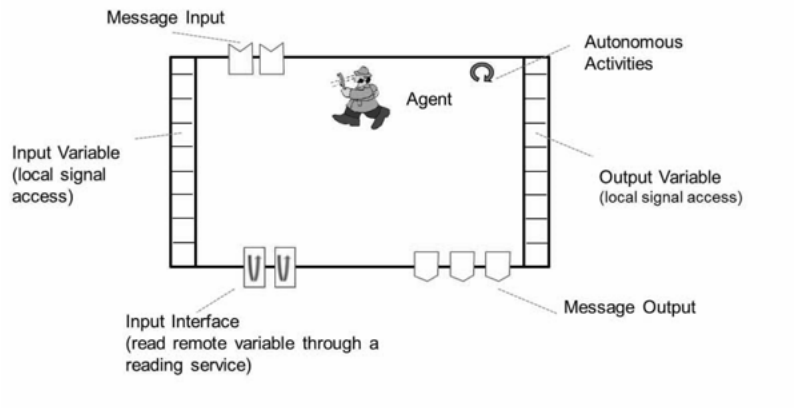


Figure 2.14: Interfaces of an automation agent

2.5.3 Concept of a Reference Model

Process automation agents can be applied in different application areas. Many common design aspects (e.g. structure, behavior, communication interface etc.) can be generally specified in a reference model. [58] has proposed an agent reference model which combines advantages of agent orientation, service orientation and function block technology. This model builds a development basis for the discussions in Chapter 3 and will be introduced in the following section.

As shown in Figure 2.14, an agent encapsulates certain autonomous activities and is present in form of a function block. In similarity to a classic function block, an agent can be activated or deactivated, and can communicate with its environment via signal inputs and outputs.

Function block technology fulfills the main requirement on agents' characteristics discussed in Section 2.5.2: Firstly, function blocks are modular and *encapsulate* algorithms for achieving operation objectives. Secondly, function blocks are *persistent*. This means that a function block (e.g. for pump control) can accompany the life cycle of an entity (e.g. a centrifugal pump in field), and keep its internal algorithms and data (e.g. set point for the power control, error state and software version). Furthermore, function blocks are *active*, since they can be executed cyclically and almost continuously. They are *reactive*, since they can sense their environment through input variables and exert influence via output variables.

One central characteristic of the reference model is that agents can provide services. In process automation contexts, services can be operative activities (e.g. process control) or organizational activities (e.g. exploration of meta-information, error diagnostic). Further discussions and examples can be found in Section 3.2.

Within an agent, components should be implemented to realize different functionalities. Typical functionalities of an agent are situation recognition, service management, status observation inference based on local knowledge bases etc. In order to realize a flexible service-oriented communication, the execution frame of classic function blocks is extended with the following three special types of interfaces (cf. Figure 2.14):

Message Inputs are responsible for receiving service requests. Every agent has the same number of message inputs as services that the agent can provide. Depending on different services, incoming requests (or commands) can be buffered or not. Service requests are realized in form of messages. A message contains more enriched information than a signal does. Every message is composed at least of the sender address, receiver address and the service identity. Further service-specific parameters and management parameters (e.g. Quality of Service) can be defined optionally.

Message Outputs are in charge of sending messages to communication partners (e.g. another agent). A message output is in charge of sending messages to a communication partner. Every agent may have more than one message output. Every command output is realized as a function block. All necessary entries for generating an outgoing request are defined as input variables of the output function block.

Input Interfaces can read variables on the local server or remote servers. In contrast to the signal input of a classic function block, the variable is read not by a signal connection, but a “Get Variable” service which can access the target. In case the variable is on a remote server, it will last for some time until the communication system delivers the variable back. During this time, the execution of the agent should not be blocked. The sampling interface should be implemented in such a way that it activates the “Get Variable” service, proceeds with its execution, and checks after a period of time whether the return value is arrived.

Similar to the communication between process control units introduced in Section 2.2.6, the message-oriented communication between agents is unidirectional. A message will be sent to the message input of the receiver without feedback by default. Every agent has a standard output variable which lists the last incoming service requests and their status. The sender can explore this variable and sense whether this request is confirmed or rejected by the receiver. In special cases, the sender can ask for feedback messages (e.g. confirmation or rejection or data message). The messages should be sent via a message output of the receiver, but not via its message input.

The elimination of feedback messages allows a simple and easy management of the communication within automation systems. An agent will not be blocked by waiting for a feedback. Complex algorithms for consistent backup and synchronization of messages can be avoided. Detailed discussion and practical examples can be found in [20, 36, 58].

The service oriented communication realizes a separation of the message exchange between the agents and the message processing within an agent. Services of an agent can be explored and requested form the outside by sending and receiving service mes-

sages. The message-oriented communication between agents can take place asynchronously. On the other hand, the execution of the services and the determination of detailed execution behavior are controlled by the service-providing agent. This design forms an important base for the development of autonomous behaviors of intelligent agents [36].

3 Specification of a Reference Model for Automation Agents

The reference model introduced in Section 2.5.3 is abstract and conceptual. Many engineering aspects are not specified in detail. The scope of complementation and improvement can be summarized as follows.

- It is an assumption that services are provided and should be processed. It is to be defined how automation functionalities can be abstracted as services and how services can be encapsulated in agents. Additionally, mechanisms for registration, exploration, description, invocation, and processing of services should also be specified in more detail.
- The service-oriented communication is not compatible with the signal-oriented communication in traditional automation systems. The former one is discrete, whereas the latter one is continuous. Additionally, service messages and classic signals have different structure and should be processed differently. It is to be defined how these two communication principles can be combined in the execution frame of automation agents. Mechanisms for message structuring, delivery and processing should also be discussed
- The previous work focused mainly on the construction of multi-agent automation system and the external design of agents. The internal construction and the execution control of internal components within an agent should be specified in more detail. In order to fulfill specific requirements on engineering in process control, description methods for continuous and procedural functions should be defined.

In order to gain a complete model for the implementation of automation agents, complementation and extension seem necessary and will be discussed in the following sections. Firstly, the functional and non-functional requirements on the engineering of automation agents will be analyzed in Section 3.1. Based on this discussion, design decisions on the agent reference model will be reviewed, revised and complemented. Starting from Section 3.2, a further development of the conceptual model will be proposed. The following engineering aspects of the model will be discussed in detail: Service model for the abstraction of internal logics (Section 3.2); Message-oriented communication (Section 3.3 and Section 3.4); internal structuring and description methods (Section 3.5); service interfaces (Section 3.6); knowledge base (Section 3.7); reference

models for the execution in runtime systems (Section 3.8); and finally, relationships and differences to related automation technologies will be summarized in Section 3.9.2.

The objective of the analysis of engineering requirements and the further development of the reference model is to establish a guideline for the implementation of various agents in process automation systems (cf. Figure 2.13).

The reference model is based on function block technology. For convenience, it will be named *FB-agent* which is derived from "Function-Block-based automation agent".

3.1 Engineering Requirements

Due to high requirements on safety and technical sustainability, process automation is conservative. In order to apply agent orientation in this field, specific engineering aspects and restrictions should be carefully regarded. In this section, general engineering requirements on automation agents will be summarized. Some requirements are borrowed from discussions in related research works [34, 35]. Some aspects were discussed in a previous work of the author [59] or are newly specified in the present work.

3.1.1 Functional Requirements

Functional requirements on engineering specify structures, functions and behaviors that are necessary for the realization of the agents' characteristics and abilities introduced in Section 2.5.

Requirement 1: Modularity and Classification

As introduced in Section 2.5 and [34, 35], every agent is an intelligent individual and should be encapsulated and persistent. It is worthwhile to encapsulate agents as classified modules in automation systems. Classification and modularity can ensure the usability and reusability of agents. Modular agents can much easier be defined as classes and multiply instantiated. Additionally, modularity is an important premise for mobile agents. In case an agent needs to change its location, modular encapsulation allows the agent to hold its internal data (e.g. state and intermediate result) and proceed with its execution at the new location.

Requirement 2: Remote and local Communication

Agents are distributed in the automation network. They should be able to communicate with local partners or remote partners across networks. Appropriate communication technology and mechanisms for data exchange should be designed.

Requirement 3: Interoperability

To realize a flexible cooperation, agents distributed in the automation network should be interoperable. Functions provided by agents should be easily accessible. Unified data exchange formats and communication protocols should be defined.

Requirement 4: Rigid and loose coupling

The cooperation partnership between agents can be dynamically built and changed. Agents in a flexible Multi-Agent System (MAS) should often be dynamically created, deleted, manipulated or relocated. As a result, automation agents should be coupled to the environment as loosely as possible. Their dependency on other system components should be kept low, so that the error risk during a reconfiguration can be reduced. However, data to be exchanged between loosely coupled communication partners should contain comprehensive information, among others, the sender identity and the receiver address. To realize a secure communication, authorization information of the sender is often necessary. In comparison to a rigid coupling (e.g. the classic signal connection), the loose coupling may lead to a significant increase in network load. Additionally, the rigid coupling can better ensure the timeliness, continuity and causality of the communication.

Automation agents should support a mixed form of ridged coupling and loose coupling with the environment. The rigidly-coupled communication can be realized by classic signal connections. The loosely-coupled communication allows a flexible communication partnership and can be suitably supported by the service-oriented communication introduced in Section 2.3.

3.1.2 Non-functional Requirements

Non-functional requirements are criteria for the quality evaluation of the agent model. According to the ISO/IEC 9126-1 [60], typical non-functional criteria for software engineering are usability, maintainability, portability etc. In contrast to functional requirements, non-functional criteria are not mandatory. They serve as a design guideline for agent engineering and can be tailored for individual application areas.

Requirement 5: Compatibility with Existing Automation Systems

New technologies can be better accepted by industrial users, in case they can be harmonically integrated into the existing automation systems. During the development of automation agents, it is worthwhile to reference and utilize existing system infrastructures, communication technologies, and description methods. Automation agents should be designed in such a way, that they can be easily plugged into the existing runtime systems.

Existing agent approaches are mostly executed in PC-based operating systems (e.g. Windows) which do not process strictly deterministically and cyclically. As listed in [61], many agent approaches are implemented in high-level languages, typically in JAVA. For instance, one main development framework JADE [52] employs a type of agent engineering that is completely based on JAVA. However, JAVA is not compatible with most existing runtime systems in process automation (e.g. PLC). Although there are also JAVA-based real-time systems (e.g. [62]), it still takes long time, until these approaches are stable and mature enough to replace the current runtime system in process automation. One central aspect to be considered by the agents' developer is the compatibility with the real-time context and programming languages according to the well-established standard IEC61131-3 (cf. Sections 2.1.2).

Agents that are incompatible with automation systems can be implemented on separate hardware and linked via special communication interfaces to the automation systems. This design limits the usability of automation agents. A seamless integration is more important. A new concept for the design and implementation of automation agents is to be developed.

Requirement 6: White-box Engineering

Classic Multi Agent Systems (MAS) and agents are enclosed and can be regarded as black-boxes. Users declare their needs via graphical interfaces (e.g. websites of e-business corporations like eBay or amazon). The operation objectives are achieved through interactions among various agents (e.g. for need identification, product brokering etc.). Users can acknowledge the execution results on the graphical interface, whereas they do not need to know how the system works and how the agents interact among each other. Users even do not need to sense the existence of the agent system. Furthermore, the engineering and maintenance of the system are not carried out by users but by professional employees.

However, users of process automation systems need be aware of the implementation in great detail. In most cases, users are also involved in the development of system functions. For instance, control engineers often need to know which algorithm is used during the automatic tuning of a PID controller; production procedures will be drafted by chemical engineers, implemented by automation engineers, and monitored by oper-

ators. All these solutions should be developed on a user-driven basis without involving the developer of the automation system.

To support the user-centralized work, automation agents should possess a *white box* structure. This means that the internal logic of agents should be transparent and visible. A user-driven engineering should be allowed. To describe different functions within agents, description methods which can be easily handled by users with no in-depth knowledge of programming or software engineering should be chosen.

Requirement 7: Platform- and Vendor- Independence

The usability of agents will be strongly limited, in case different agents can only be implemented in special runtime systems and require special communication interfaces form certain vendors. Additionally, mobile agents can only change their location flexibly, when they are compatible with different systems. It is therefore desirable to design agents independently from the underlying execution platform and the designated realization technology from a certain vendor.

3.2 Service Model

The conceptual model introduced in 2.5.3 can partially fulfill the engineering requirements summarized in Section 3.1. Among others, the non-functional requirements should still receive more considertation. Starting from this section, extensions and improvements on the conceptual agent design will be discussed according to the summarized engineering requirements.

As indicated at the beginning of this chapter, one main engineering aspect of FB-agents is the modelling of services. The service orientation introduced in Section 2.3 allows a loose coupling and a flexible cooperation of FB-agents. Additionally, it allows an abstraction of implementation details within FB-agents.

To abstract automation functions into services, the layered model in Figure 3.1 is defined for FB-agents. The model elements will be introduced in the following paragraphs.

Internal logic is the concrete implementation code for realizing a specific automation function. Internal logics should be implemented in programming languages and contain all implementation details.

An *operation* represents an elementary function or processing activity that can be performed by an agent. Operations can be “open the valve”, “explore meta information”, or “shut the plant down according to a control procedure”.

Internal logics within FB-agents can be abstracted into operations. For instance, the logic “OUT:=TRUE; SET_POINT=30” within a “valve control agent” can be abstracted as the operation “OPEN”. The aforementioned “shutdown procedure” can be abstracted

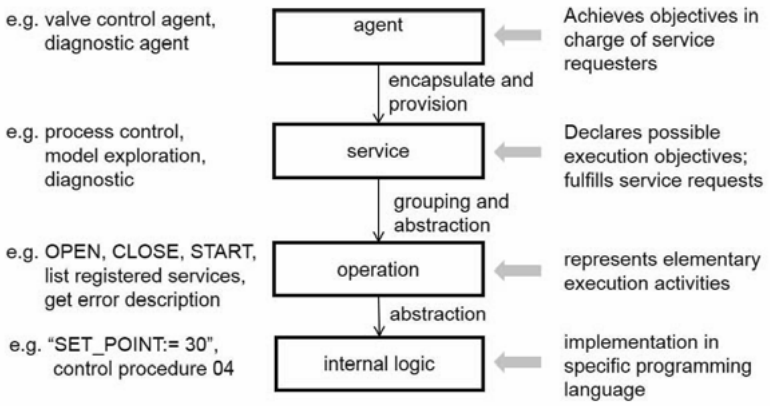


Figure 3.1: Abstraction hierarchy of service orientation

as a "shutdown" operation. Operations should be registered within the FB-agent and can be explored from the outside. It should be clearly defined which input information is required for the operation invocation.

Services can be regarded as high-level abstractions of operations. Services group operations involve less implementation details. Possible services of automation agents are process control, diagnostic, exploration etc. Services represent functional abilities that can be provided to the outside. Services should be registered and should be explorable form the outside. Every service should possess a service description which states the objective and clarifies execution criteria.

A service request specifies a concrete execution objective. The achievement of the objective is to be realized by invoking an appropriate operation or operations. For instance, an agent provides the service "product transport" which can be obtained through different operations like " transport with a certain flow rate", "transport via the shortest path" etc. The suitable operation can be explicitly defined by service requester or can also be chosen by the agent autonomously.

A service can have multiple underlying operations. Every operation can have only one covering service. Operations under the same service are normally mutually exclusive. For instance, the "open" and "close" operations of the aforementioned valve control agent represent alterative operating modes of the valve and of the covering service "process control". Only one of the conflicting operations can be activated at the same time.

Operations of special services can also be activate at the same time. For instance, an "exploration" service can group operations such as "get KPI", "get meta infromation" etc. The operations are not strictly alternative and can be processed at the same time.

It should be defined for single services whether their underlying operations are mutually exclusive. An example of controlling alterative process control operations will be introduced in Section 7.3.2.

The top layer *Agent* in the model is the carrier of internal logics and the provider of services. Agents are encapsulated and possess interfaces for communicating with the environment. Every agent may contain more than one service and operation. Every service can have many underlying operations. An operation can only be assigned to one service.

The layered model in Figure 3.1 is defined on the basis of the service model shown in Figure 2.11. FB-agents are service providers and providing entities at the same time.

3.3 Message Format

Service oriented interaction between FB-agents builds upon the exchange of messages. Based on a theoretical discussion [26] of the Chair of Process Control Engineering, a XML-based message format is specified in a student work for FB-agents. A standard message can be generated as follows:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <SvcMsg xmlns:ns="MsgSchema">
3      <Header locAdr="server1/processdata" sysAdr="10.42.12.54
           :7509" MsgID="13223" RefMsgID="12432">
4              <AuthData>Authentication-Data goes here.</AuthData>
5      </Header>
6      <Body Service="diagnostic"
7        Operation="get key performance indicators" Ticket="userXY"
             >
8            <StructData ID="ServiceRequest">
9                  <Object ID="Pump14">
10                   <KVP Key="pressure">1.0bar</KVP>
11                   <KVP Key="flow">30m3/h</KVP>
12                 </Object>
13           </StructData>
14     </Body>
15 </SvcMsg>
```

The start and the end of the contents are explicitly marked. A standard message has two parts: *header* and *body*. The former part contains information about sender, receiver, coding, sender authorization, time stamp, message identity etc. The latter part consists of a set of Attribute-Value pairs (cf. Section 2.4). The given example message requests the operation “get key performance indicators (KPI)” of the service “diagnostic”.

$ServiceRequest$ indicates the message type. $Object\ ID$ specifies the service requester. The expected KPIs are defined as $KVP\ Keys$. An example value is given for every desired KPI in the message and indicates data format, number of decimal places and physical unit.

In order to simplify the message parsing, different optimization approaches can be applied in practical use. The proposed message format can be applied as standard message pattern. As soon as two communication parties build up a fixed and confident partnership, they can exchange simplified messages. For example, in case a control agent for “valve014” receive the pre-agreed code “cmd40” from a supervisory agent “group control 010”, it will start the service “process control” and the operation “open” with the parameter “opening=40%”.

The FB-agent model presented in this work supports two general message types: *service request* and *data message*. As the name implies, the former one describes a request with regard to a service (e.g. “diagnostic”). The latter one contains data such as KPI, current execution state, request confirmation etc. In case the message is a service request, $service$ must be defined in the message body. Depending on different services, $operation$ in a service request can be either required or optional. In the second case, the agent should identify a suitable operation autonomously.

As introduced in Secion 2.5, many agent approaches support complex conversations (e.g. confirm, propose, rejection, negotiation) in imitation of human languages. Syntax, semantics and rules of a certain conversation type can be modeled in a so-called *ontology* which should be loaded on the sender and the receiver. Every message header should indicate which ontology should by loaded by the communication parties, so that the sender can recognize the conversation type. For instance, The variety of conversations between communication parties in process automation systems is limited. For instance, it does not seem worthwhile to allow automation agents to negotiate with the user autonomously, whether a production task is to be performed. In the most cases, the two general message types are sufficient enough for the communication between FB-agents.

The message format for automation applications and the classification of messages is part of a larger research work conducted by the Chair of Process Control Engineering in Aachen. Ongoing works and publications can be found on the homepage of the Chair. As a general reference model, FB-agent supports only the two simple message types. Extensions according to existing agent languages can be researched in future work. For convenience, messages in the following discussions will be shown in simplified from:

“ sender= ⋯; receiver= ⋯; service=⋯ ; operation=⋯ ; parameter= ⋯”

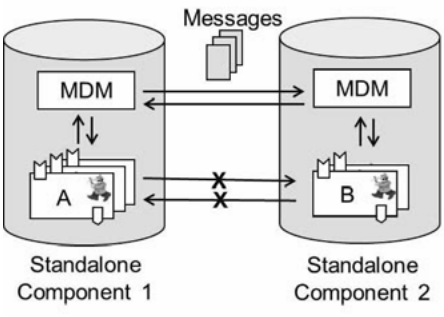


Figure 3.2: Message exchange via Message Delivery Modules (MDM)

3.4 Message Delivery Model

The message exchange between automation agents can be realized in different ways. A reference model for the implementation in automation runtime systems (cf. Section 2.2.3) is defined in Figure 3.2.

One possible design is to allow a direct agent-to-agent communication. This design has two disadvantages. Firstly, every single agent should be able to allow and manage communication with remote partners. Special interfaces for sending and receiving messages over the network should be defined. Agents should be able to deal with abnormal situations during the remote communication. For instance, in case the network is temporary overloaded, all outgoing message should be buffered locally and sent again when the network is free again. The agent model and its service interfaces will be unnecessarily complex and bloated. Secondly, it is difficult to monitor and observe the network load. Agents are decentrally distributed in the automation network. The message-oriented communication between agents can be dynamically build at runtime. Messages will be sent discretely and irregularly. In case the agents are allowed to send messages directly, it is not easy to determine how many messages will be sent within the whole network during a certain time period.

The second possible design is to manage the message-oriented communication in a centralized manner. As shown in Figure 3.2, agents are installed on Standalone Components (SC) which are the elementary components for organizing programs (cf. 2.2.3). Every SC possesses a Message Delivery Module (MDM) which manages the message transmission between local and remote agents. Agents are not allowed to exchange messages directly, but only through the local and remote MDMs which act as intermediaries.

In case a message is to be sent from agent A to agent B as indicated in Figure 3.2, A sends the message to the MDM on $SC1$. The MDM forwards the message to the MDM

on $SC2$. The second MDM sends the message to agent B. In case the sender and the receiver are located on the same Standalone Component, the local MDM delivers the message to the receiver directly.

Message Delivery Modules (MDM) only read the receiver address of messages that are to be transmitted. As an option, they can check the authority of the sender. Further entries in a message are to be evaluated and interpreted by the receiver not by the MDM.

The agent-MDM-agent communication has the advantage that the discrete and message-oriented communication on single Standalone Components (i.e. servers) and in the whole network can be better observed and supervised. The Message Delivery Module on every standalone component acts as the central communication module which manages the message exchange and deals with abnormal situations during the communication. For instance, in case the network is overloaded, all messages will be buffered centrally in the MDM and sent until the network is free.

The idea of MDM is borrowed from network communication technology. Many existing solutions for this area can be directly applied to solve communication problems, such as the discovery (or addressing) of the receiver, the measures for abnormal situations etc.

The FB-agent model does not restrict mechanisms for the discovery and the addressing of communication participants. These two functions are to be realized as two basic system services. In further discussions of the dissertation, the location of message senders and receivers will be defined for convenience by their absolute path which is composed of device name, server name and local path on the server.

The message delivery among all communication participants is unidirectional. As discussed in Section 2.5.3, feedbacks are not supported and are to be sent separately as data messages.

3.5 Internal Structure

As the name implies, FB-agents are to be encapsulated as function blocks. To realize the white-box engineering required in Section 3.1.2, function block technology is applied to construct the internal logics of FB-agents. The external design and the internal construction form an invisible whole and will be termed *Execution Frame* of the FB-agent model. The Execution Frame is encapsulated and possesses a local namespace for variables and underlying function blocks of an agent. Internal function blocks within the frame cannot be accessed directly from the outside.

As shown in Figure 3.3, internal logics of an agent are to be encapsulated as modular function blocks which can be classified into two types:

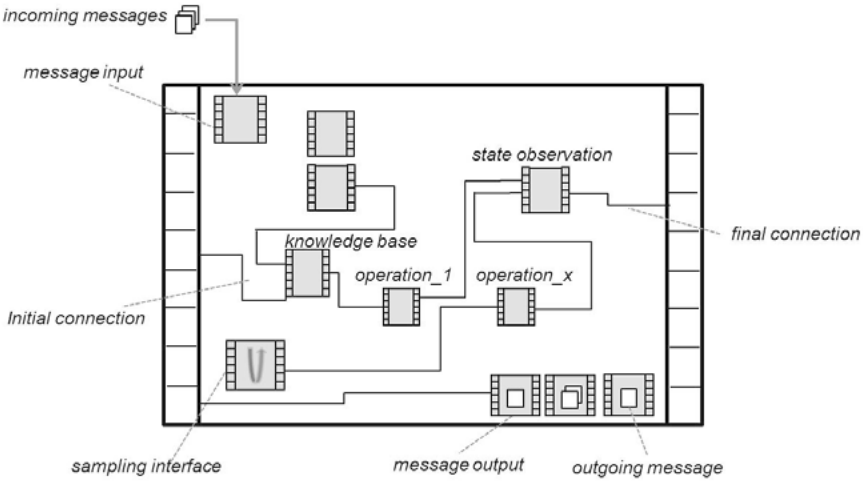


Figure 3.3: Execution Frame of an FB-agent: Modular composition of internal logics

1. *Atomic Function Blocks (AFBs) or complied function blocks* are “black-boxes”. Their internal logic is fixedly defined, invisible from the outside, and can normally not be edited by users. AFBs can be applied to realize functions of different complexity, from simple logical operations (e.g. AND and OR), over complex algorithms (e.g. model predictive control), to knowledge base (e.g. in neuronal net). For the implementation in automation systems, an AFB can be defined as a class in a library, and be instantiated multiple times. In case the logic is changed by the user, the AFB class is to be recompiled and reloaded into the system.
2. *Composed Function Blocks (CFBs)* are “white-boxes”. Viewed from outside, CFBs can be handled equally to AFBs. They are encapsulated and have input variables and output variables. However, their internal logic is a composition of modular components. The logic can be edited by users without recompilation. The modular composition allows an intuitive graphical visualization on the Human Machine Interface (HMI). CFBs can be further divided into two subtypes: *continuous CFB* and *procedural CFB*. The former one describes continuous functions. The latter one is for sequential functions (e.g. control sequence) and state-based functions (e.g. state machines). The modular components within a continuous CFB are interconnected function blocks which can be atomic or composed. The nesting of CFBs in CFBs allows a hierarchical structuring of complex logics. Internal components of a procedural CFB are states and transitions. Their data exchange and execution control will be discussed in Chapter 4.

The communication between AFBs and CFBs is realized through the exchange of signals. The FB-agent model does not restrict the programming language for the im-

plementation in runtime systems. Atomic function blocks (AFBs) can be implemented in IEC61131-3 [3] languages and further non-standard languages such as ANSI C or JAVA, as long as they are supported by the underlying runtime system. Continuous CFBs can for example be implemented as Function Block Diagrams (FBD) according to the IEC 61131-3 [3]. Procedural CFBs can be designed and implemented in different description methods (e.g. Sequential Function Chart). Appropriate approaches will be analyzed in Chapter 4 and Chapter 5.

Function blocks realize a modular encapsulation of the internal logics of automation agents. This modular design is native in the area of process automation and makes a compatible integration of FB-agents in existing automation systems possible. Additionally, an acceptance of users can be expected, since the users are accustomed to use function blocks. Additionally, the requirements on modularity and white-box engineering outlined in Section 3.1.2 can also be fulfilled.

Atomic and composed function blocks are not new in classic automation systems. Users apply atomic function blocks to perform mathematic calculations such as addition and multiplication. Function Block Diagrams are applied to implement interlock logics e.g. for a pump. However, the function block technology has gradually been reduced to a programming language with regard to its status in industrial use. The concepts of modular encapsulation and hierarchical nesting are often neglected. For instance, procedures in many automation systems are not modularly encapsulated. Hierarchical nesting is often also not supported (cf. Chapter 5).

To improve the usability of automation systems, it is worthwhile to highlight the guidance of function block technology during the engineering of automation functions [35, 63]. The function block technology is therefore applied as the essential modeling principle in the FB-agent model. In contrast to classic function blocks, the programming language for the implementation in runtime systems is not restricted. Additionally, all continuous and procedural functions should be strictly encapsulated as modular function blocks.

Section 3.6 will introduce the realization of service interfaces. The execution control of internal components within the Execution Frame will be modelled in Section 3.6. The differences between traditional function blocks and FB-agents will be summarized in Section 3.9.1.

3.6 Service Interfaces

The service-oriented communication between agents is based on the exchange of messages. To merge the message-oriented communication with the signal-oriented context within FB-agents, services interfaces are realized as special function blocks which convert messages to signals or vice versa.

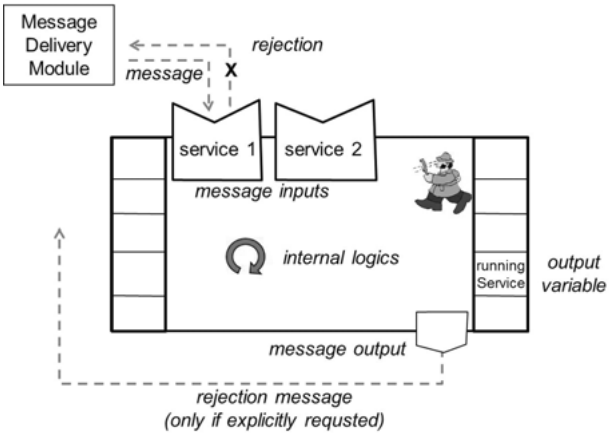


Figure 3.4: Agent with multiple message inputs for registered services. In case the desired service does not exist, the service requester cannot be informed about this error.

Viewed from the outside, an agent possesses special interfaces for receiving and sending messages. Within the agent, the service interfaces behave like normal function blocks. An incoming message will be parsed. Its contents will be saved as output variables of the function blocks and can be read and processed by further function blocks, e.g. via signal connections. To generate an outgoing message, the message contents can be saved as input variables of a message output which composes a message from the entries. Service interfaces of the FB-agent model will be introduced in detail during the following sections.

3.6.1 Message Input and Message Inbox

Message Input

According to the conceptual model introduced in Section 2.5.3, an agent should setup a message input for every single service (cf. Figure 3.4). To deliver a message, the Message Deliver Module (MDM) introduced in Section 3.4 finds the receiving agent and sends the message to the message input of the desired service. The message will be buffered in the message input and be processed by the receiving agent. The message-oriented communication is unidirectional. Conformations or rejections are to be sent as separate messages.

This design has the disadvantage that the service requester cannot be easily informed, in case its service request is invalid or the desired service is not registered

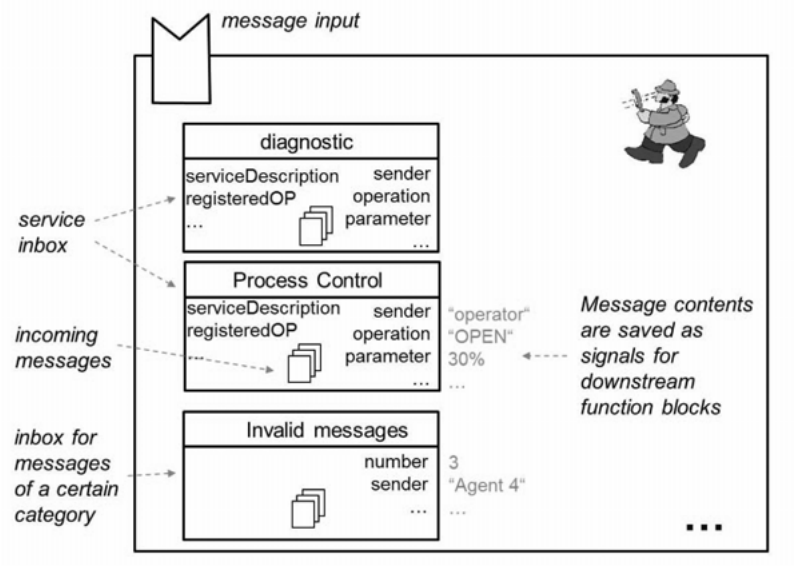


Figure 3.5: Service interfaces for the processing of incoming messages

in the receiver. In case the Message Delivery Module (MDM) in Figure 3.4 sends a service request on an unregistered “service 3” to the agent, this request cannot be buffered and processed by the receiver, since no message input named “service 3” exists. As the receiving agent can only process messages in message inputs, it cannot generate a rejection message and send it back to the sender.

The receiving agent can have a signal output (cf. $runningService$ in Figure 3.4 which highlights the services that are being processed or rejected. The sender can read this output and sense that its service request is not accepted. However, a service request can be rejected for different reasons, for instance, the agent is busy, the message is invalid or the desired service does not exist. The sender cannot easily recognize the actual reason.

It is defined as a concept decision that every FB-agent should possess only one central interface for the message reception. Within the name space of an FB-agent frame, the message Input should have a unique identity, e.g. ‘$msgIN$. In case the Message Delivery Module receives a message for agent A, it will deliver the message to the target address $AgentA.msgIN$.

As shown in Figure 3.5, every agent should possess only one message input. In contrast to the conceptual model introduced in Section 2.5.3, the message input does not represent services and does not buffer any messages. Incoming messages are forwarded and buffered in *Inboxes*.

Message Inbox

A Message inbox is a container of incoming messages of a particular category. Every registered service within an agent should have an exclusive message inbox which will be termed *Service Inbox*. The message input forwards all message for a service to the inbox of the service. Service inboxes are special message inboxes but not vice versa. Further inboxes can be optionally defined for example for invalid messages and for service requests that are to be rejected.

The idea of message input and message inbox is similar to an e-mail account which can automatically sort incoming mails into subfolders for “work”, “bills”, “spam” and etc. The Message Delivery Module (comparable with web mail server) addresses the receiver via its unique message input $Agent.msgIN$ (comparable with liyong.yu@plt.rwth-aachen.de), irrespective of the existence of the service register (comparable with the subfolders). This design has the advantage that invalid messages can also be buffered and processed by the receiver.

Service Registration

As introduced in Section 3.2, services and operations should be registered within an agent. Services and operations should be able to be added or deleted by users.

Since every registered service has a service inbox, the list of all service inboxes can represent the service register within an FB-agent.

A function block or a group of function blocks within an agent can be registered as an operation (cf. Figure 3.1). Every operation should have a covering service. Ever service inbox holds a local register of underling operations.

As shown in Figure 3.5, the inbox of the “Process Control” receives a message which requests the operation “OPEN”. Information about sender, operation identity and parameters (e.g. $opening = 30\%$) will be parsed by the service inbox and saved as its output variables. These variables can be read by downstream function blocks for the operation realization and for status observation.

Services and their underlying operations should be self-descriptive, namely, they should possess a description which contains at least a textual introduction and a list of parameters. The service- and operation-description can be defined on the meta level of service inboxes. In order to allow a flexible extension and modification by users, it is suggested to define service description and operation description as variables of service inboxes.

As shown in Figure 3.5, every service inbox has an input variable $serviceDescription$ which holds the textual description of the service and can be explored by service requesters. The variable $registeredOP$ lists and describes all underlying operations of the service.

The function-block-oriented design of the service register allows a flexible engineering of services and operations by the user. In order to define and register a service, the user can instantiate a service inbox. The service description and the underling operations can be defined as parameters. In case a service inbox is modified or deleted, the service is redefined or de-registered.

Section 6.2 will introduce an approach for the registration of operations as representative objects. In this approach, every registered operation should have an exclusive representative object under its covering service inbox. The operation object holds the operation description and can be flexibly defined and modified by users.

Plausibility Checks

In every execution iteration, the message input checks whether new messages exist. The following general checks will be performed on every incoming message:

- The message format must be plausible.
- In case the message is a service request, the desired service must be registered. In other words, an inbox for the service is present in the agent.

The following checks can be performed optionally:

- an operation is defined in the message and is registered in the agent.
- the message sender must be explicitly given.
- the sender must be authorized for accessing the agent.
- the sender must be authorized for using the service.

Messages that fail the checks are regarded as invalid and will be deleted by default. Invalid messages will be forwarded to the message inbox “invalid” (cf. Figure 3.5), only if the sender requires a feedback (confirmation or rejection) explicitly. Valid message are forwarded immediately to service inboxes. At the end of every execution, the message input should be kept empty.

The service inboxes check according to the service description whether the format of incoming messages is plausible, and whether all necessary parameter are defined. Service inboxes parse messages. Key contents of a message are saved as signal outputs of the inboxes and can be read by further internal function blocks (e.g. via signal connections).

Message Processing Rule

Messages in an inbox can be processed in different ways. Every message inbox should have the following properties which can be realized as variables of the inbox instance or be fixedly defined in its meta model:

- *Prioritization Rule* defines the processing order of the messages. Messages can be prioritized for example according to their arrival time and be processed in the principle of “First-In-First-Out” (FIFO) or “Last-In-First-Out” (LIFO). Messages can also be prioritized according to a certain property, e.g. the sender’s authority or the importance. For instance, an emergency stop instruction is more important than a normal instruction and should receive higher priority.
- *Capacity* defines the number of messages that can be at most contained in an inbox. The inbox capacity is unlimited by default.
- *Measure for Overflow* defines the behavior of an inbox, in case its capacity is reached. The inbox can refuse to receive any further messages. Alternatively, it can delete the oldest message and can always receive new messages.

With these properties, different processing rules can be defined for individual message inboxes.

For instance, the following rules can be defined for the service inbox “Process control” of a valve control agent:

- Prioritization rule 1: Since the valve can only be allocated by one service requester at a time, only one service request message will be accepted. Further requests will be ignored or rejected.
- Prioritization rule 2: A manual operation should be allowed any time. Thus, the operator has higher priority than all software requesters.
- Prioritization rule 3: In case two senders have the same priority, the message that received earlier has higher priority (FIFO Principle).
- Capacity: 5 messages can be buffered in the inbox.
- Measure for overflow: In case more than 5 messages are received, the oldest message will be deleted. This rule ensures that service requests from operators will never be ignored. The service inbox can always be operated by the operator, even if its message capacity is reached.

The valve control agent may receive the following three requests on the service “process control” :

$$\begin{aligned}
&\text{Message 1}: \quad ... \ \text{Sender} = \text{Agent_A}; \ ... \ \text{Operation} = \text{OPEN}; \ ... \\
&\text{Message 2}: \quad ... \ \text{Sender} = \text{OPERATOR}; \ ... \ \text{Operation} = \text{ALLOCATE}; \ ... \\
&\text{Message 3}: \quad ... \ \text{Sender} = \text{Agent_B}; \ ... \ \text{Operation} = \text{CLOSE}; \ ...
\end{aligned}$$

The sender $Agent\_A$ of the first message wants to open the valve. The operator asks to allocate the valve and control it manually. $Agent\_B$ wants to close the valve. However, the valve cannot open and close at the same time. It can also not be operated manually and automatically at the same time. According to the processing rules, all three messages will be buffered and processed, since the capacity of 5 is not reached. $Agent\_A$

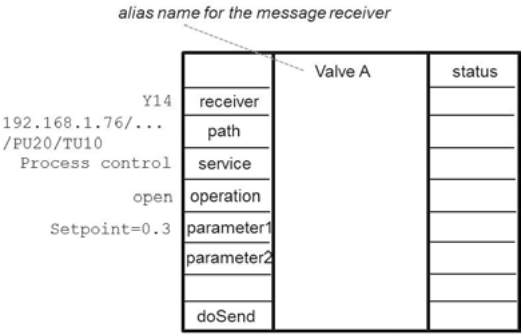


Figure 3.6: Message output implemented as function block

and $Agent\_B$ have the same priority. Message 1 arrived first and therefore has higher priority than message 3. Message 2 from the operator is to be accepted.

Processing rules for further services can be defined alternatively. For instance, requests on a "diagnostic" service are not mutually exclusive. All incoming requests must be buffered and processed by the receiver. The following processing rules can be defined for the service inbox:

- Capacity: unlimited.
- Priority rule: First-In-First-Out (FIFO)

3.6.2 Message Output

A message output generates messages and delivers them to the local Message Delivery Module. In contrast to the message input introduced in Secion 3.6.1, the message output follows the conceptual model introduced in Section 2.5.3 but is not completely redesigned.

As shown in Figure 3.6, all entries for a message are defined as input variables such as receiver address, name of the service to be requested, operation name and parameters. Every message output function block has a standard input variable "doSend". In case this Boolean variable is set from $FALSE$ to $TRUE$, the message output checks the plausibility of all entries, generates a message object and delivers it to the local Message Delivery Module introduced in Section 3.4.

Every FB-agent may have more than one message output. The identity (i.e. function block name) of a message output can be set as an alias name (e.g. "Pump0014", "Valve" or "Archive") which represents a certain message receiver. A message outbox can also be responsible for the generation of messages of a certain type, for instance

diagnostic data or rejection. In this case, its identity can be set to “diagnostic data” or “rejection”.

The location and the actual identity of the receiver are defined as two input variables of the message inbox: $receiver$ and $path$. They can be defined manually or be dynamically complemented. In the conceptual model introduced in Section 2.5.3, $path$ can be complemented by the runtime system. This design allows a direct value assignment from the outside. To ensure the encapsulation property of the FB-agent frame (cf. Execution Frame termed in Section 3.5), $path$ is not allowed to be set directly from the outside, but only via an input of the covering execution frame.

3.6.3 Input interface

An Input Interface can read the current value of a variable which can be present on the local or on a remote server. In comparison with the conceptual model introduced in Section 2.5.3, the input interface will be defined in more detail but not completely redesigned.

As shown in Figure 3.7, an input interface is to be encapsulated as a function block. The identity and the path of the desired variable are defined as input variables. Similar to the message output introduced in Secion 3.6.2, the identity and the path of an input interface can be statically defined or dynamically complemented at runtime.

To read a remote variable, the input interface invokes a system service “Get Variable” which is to be standardly provided by every Standalone Component. The service finds the variable, samples its current value and delivers it back to the input interface. In addition, a time stamp is and a status variable should be sent back to the input interface.

The time stamp indicates the sampling time of the variable. In case the target variable has no time stamp, the current system time will be taken.

The status variable indicates the quality of the value. A value can be good, bad or questionable etc. The reference model FB-agent does not restrict the type of variable status. A related research work on this topic can be found in [64].

Input interfaces are suitable for occasional or one-off accesses. They read the target variable only if necessary. Classic signal connections, however, read the target variable cyclically and load the system continuously. Additionally, the service oriented sampling of variable value can reduce the dependence of a mobile FB-agent on its environment. For instance, in case a mobile agent changes its location, its input interfaces do not need to be reconfigured. No rigidly-connected signal connection should be deleted at the old location and be rebuilt at the new location.

Input interfaces can also read variables periodically. The “Get Variable” service can be touched off again, when the last reading activity is finish. An input interface of a diagnostic agent can for example read a temperature value once an hour.

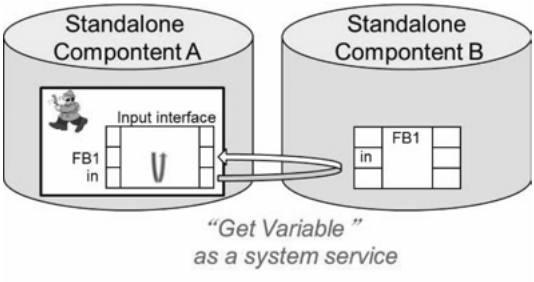


Figure 3.7: Signal communication across networks via active input interface

3.7 Knowledge Base

Agents should possess a local knowledge base to realize autonomous behaviors. The present work focuses on the construction of automation agents, not on the development of autonomous algorithms. To realize the knowledge base in FB-agents, both, theory and practice in area of process automation and computer science can be utilized.

Knowledge-based realization of automation tasks is not a novel research area in process automation. Many knowledge-based approaches have been developed in resent years. For instance, observation and diagnostic logics for field devices (e.g. pump) can be automatically generated [65]. Basic automation logics (e.g. single control units, HMI faceplates) can automatically initialized according to design data and knowledge bases [66, 67]. An overview of related works can be found in [67, 68]. These advanced automation algorithms can be applied to realize the knowledge base of FB-agents. Section 7.4 will show an example of engineering agents, in which knowledge-based engineering algorithms for the initialization of process control and plant simulation are encapsulated.

A common implementation issue that the knowledge-based automation applications face involves the coupling to existing automation systems. The most common solution is to develop advanced algorithms in a specific programming environment and couple the implementation to further automation systems via a special communication interface (e.g. OPC server). A development of application-specific solutions for visualization, data archiving, and communication interface is often necessary.

The execution frame provided by the FB-agent model allows a direct integration of advanced algorithms into existing automation systems. Knowledge-based algorithms can be encapsulated as function blocks (atomic or composed) and be plugged into FB-agents. They can be abstracted as operations and services that can be invoked via unified service interfaces of agents. Existing services in the system for diagnostic, model management, data archiving etc. can be utilized by the new agents.

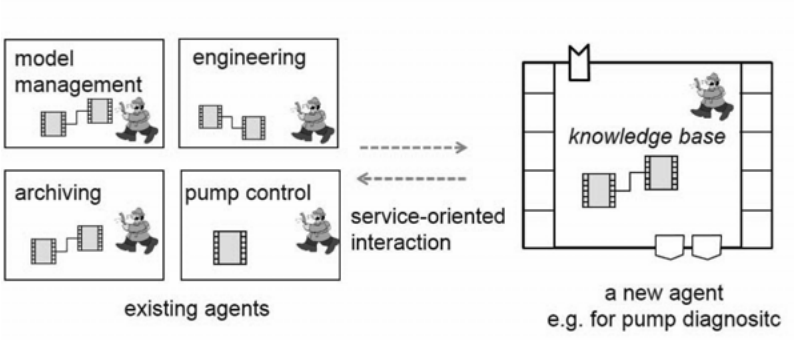


Figure 3.8: Integration of advanced automation algorithms as a knowledge base of an FB-agent: Service-oriented interaction with existing system components. Existing solutions can be utilized for model management, engineering, archiving of data and knowledge base etc.

For example, the aforementioned knowledge-based algorithms for pump diagnostic can be encapsulated within a “diagnostic agent”. As shown in Figure 3.8, the new agent can communicate with the target valve control agent via service-oriented interaction. It can also request an existing “data archiving agent” to read diagnostic data from the long-term archive and save diagnostic results in the archive. Further services for model management and engineering can be applied to maintain, analyze and backup the knowledge-base of the new agent.

3.8 Execution Model

Theoretically, FB-agents are standalone modules that can be executed simultaneously and independently from each other. However, in order to avoid unexpected execution result and ensure a safe operation in automation systems, a deterministic execution within the FB-agent frame is desirable. In other words, internal components of an FB-agent should be executed according to a clearly defined order.

In every iteration cycle, the following execution procedure will be performed once:

1. Switch incoming signal connections and update all inputs.
2. Execute the unique message input
3. Execute message inboxes
4. Switch the so-called *initial connections* which connect inputs and internal function blocks (cf. Figure 3.3).

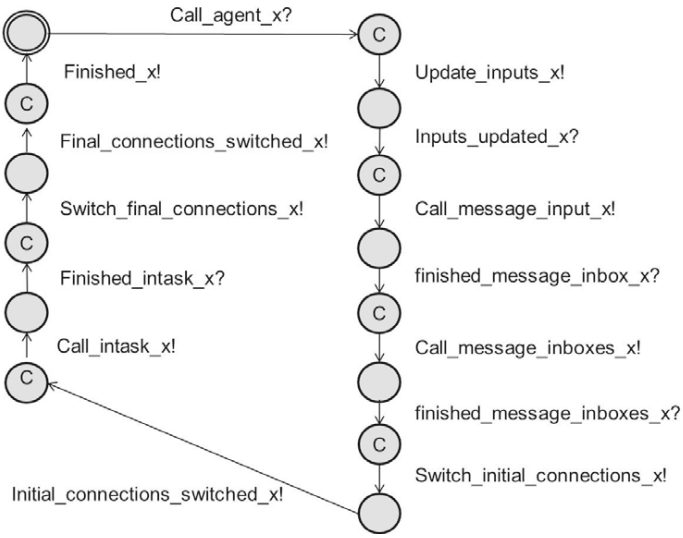


Figure 3.9: UPPAAL automaton for one iteration of FB-agent execution frame

5. Execute an internal task list named $intask$ which controls the execution of internal function blocks within an agent. Although the message input and the inboxes are also function blocks, they should be regarded as special inputs and should be executed at the beginning of the iteration cycle. Function blocks under the task list will be executed by default “from above to below, and from left to right” as they are positioned on the graphical visualization. The execution order can also be flexibly defined by users.
6. Switch *final connections* which are connections ending with a signal output (cf. Figure 3.3).

The execution procedure is represented in form of a UPPAAL automaton in Figure 3.9. Based on this procedure, FB-agents can support the following execution behaviors:

- *Cyclical Execution* approximates a time-continuous execution. The aforementioned execution procedure will be processed cyclically. A cyclic processing agent is always active.
- *Event-driven Execution:* An event-driven agent is inactive in normal state. It starts with its execution only if a certain event occurs. The FB-agent model does not have event inputs. An event can be for instance the value change of variables, or the receiving of new messages. An activated agent will be cyclically executed, until it reaches a certain stable state (e.g. all internal parameters do not change anymore,

or an underlying procedure is terminated). Then, the agent will be deactivated automatically.

- *Call-based Execution*. The agent is always inactive and will be executed, only if it is invoked, e.g. by a task. Call-based agents are prepared for runtime systems with no cyclic-processing context.

Due to the central role of the cyclic-processing context in industrial automation (cf. Section 2.1), further discussions in this dissertation focus on this mode unless otherwise noted.

3.9 Related Automation Technologies

The development of the FB-agent model has referenced many related works and existing solutions in process automation and computer science. The relationship and differences will be summarized in this section.

3.9.1 Relationship with Function Block Technology

The FB-agent model applies the function block technology as a basic modelling principle for the modular encapsulation of agents and their internal components. In order to be integrated in existing automation systems, the FB-agent has inherited many mechanisms (e.g. communication interface and execution scheduling) from these two function block standards IEC 61131 [3] and IEC 61499 [14]. However, the FB-agent has the following three main differences to the two standards.

Firstly, the FB-agent model extends the classic function block with service interfaces which realize a loosely coupling and dynamical cooperation of agents (cf. discussion in Section 3.6). In contrast to the event interfaces according to IEC 61499, service interfaces are not fixedly connected with communication partners. Service-oriented communication can be dynamically built, whereas event-oriented interactions among IEC 61499 FBs should be realized as fixed connections. Additionally, every FB-agent has only one standardized message input, whereas IEC 61499 function blocks may have multiple event interfaces. Furthermore, structure and key contents of FB-agents' service messages have a standardized definition, whereas the IEC 61419 event is abstractly defined without the use of a unified data model.

Secondly, function block is defined as a program organization unit and a programming language in the two standards, whereas FB-agent applies function block as a modelling principle. On the one hand, as introduced in Section 3.5, the description method for the implementation of FB-agents' internal blocks can be freely chosen. On the other hand, the FB-agent model is neutrally defined for different execution environments (call-

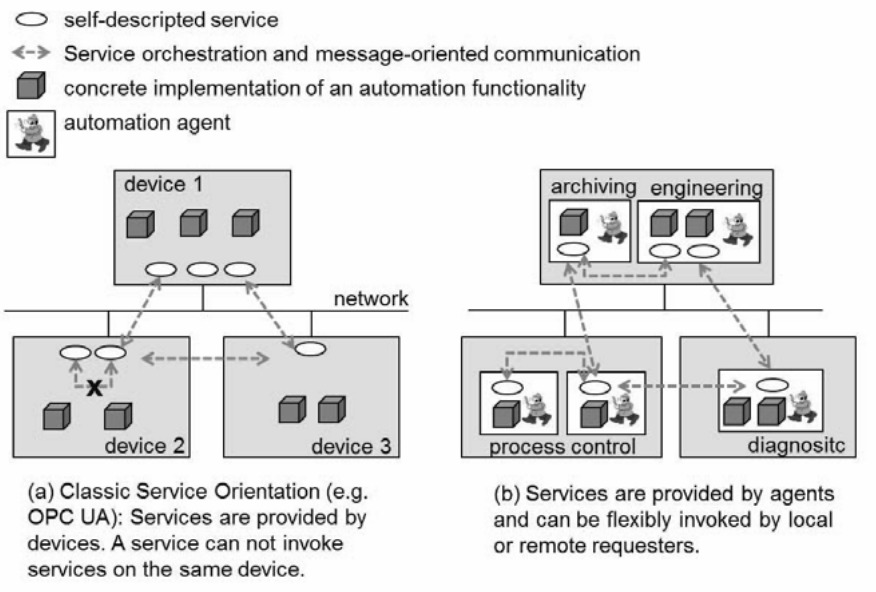


Figure 3.10: Two different concepts of Service orientation in process automation

based, cyclic or event-driven), whereas the function blocks according to the IEC 61131 and IEC61499 are closely coupled to a specific context.

3.9.2 Relationship with Service Orientation

The FB-agent model combines the advantages of agent-orientation and service orientation (cf. Section 2.3). Internal logics of FB-agents should be abstracted as services. The communication and interaction between FB-agents are service oriented.

Similar to the service-orientation, agent orientation also realizes an abstraction of implementation details. However, agents possess a knowledge base and behave autonomously, whereas classic service providers do not need to (cf. Section 2.5).

Theoretically, services are abstract and do not need to have representatives in the runtime system. However, FB-agents are concrete and encapsulated software modules in runtime systems. They possess individual name spaces and act as service providers.

As introduced in Section 2.3, service orientation is currently applied in process automation for realizing a neutral communication between different systems. According to the OPC UA standard [30], services are provided by systems or servers installed in the automation network and can normally only be invoked by remote requesters (cf. Fig-

ure 3.10 (a)). According to the FB-agent model, services are provided by agents. As shown in Figure 3.10 (b), services can be invoked by local and remote requesters. A flexible orchestration of local services on the same server is allowed.

FB-agents are regarded as the elementary service providers in the automation system. Services are provided by agents not by the servers on which the agents are installed. This design principle is similar to the soccer example introduced in Section 2.5. The “defense” and “attack” services are provided by the soccer team and the individual players, but not provided by the play field. The whole automation system can be regarded as a composition of agents which are in charge of services of “process control”, “diagnostic”, “engineering” etc.

Along with the rapid development of computer science, the boundary between devices and servers will get more and more blurred (compare discussions in Section 2.1.3). Modern computer technologies like virtualization and cloud computing allow users to design software irrespective of the underlying hardware infrastructure. The FB-agent model follows this trend and provides a new design concept for the construction of the future service-oriented automation systems.

3.9.3 Relationship with ACPLT/PF

Both the FB-agent model and the ACPLT/PF (cf. Section 2.2.6) follow the resource-measure-model introduced in Section 2.2.5. The FB-agent model borrows many design decisions from ACPLT/PF, among others, the function-block-oriented encapsulation and the unidirectional communication. However, FB-agents and PCUs have essential differences. These will be summarized as follows.

Firstly, ACPLT/PF is designed for process control, whereas FB-agent is a general design pattern for a broader range of application areas such as archiving and engineering (cf. Figure 2.13). Additionally, FB-agents have advanced requirements on their functional abilities. FB-agents should possess a knowledge base and autonomous behaviors.

Secondly, communication interfaces are differently defined in ACPLT/PF and FB-agent. The order interface in ACPLT/PF is realized as a special signal input of function block. In case it receives many messages from different senders, only the last received order (or command) will be taken. Further orders will be overwritten, even if they have higher priority and emergency. For instance, the request on a manual operation in the example in Section 3.6.2 will be ignored. However, the message input and service inboxes of FB-agents can process messages in different ways. Different prioritization rules and processing modes are defined for individual services. As discussed in Section 3.6.2, no important messages (e.g. a manual shutdown) will be overwritten or overlooked.

Thirdly, ACPLT/PF applies a simple message format of Value-List. FB-agents apply a XML-based format which allows a flexible construction of content-rich messages and a formal evaluation according to message schemas (cf. Section 3.3).

Furthermore, the FB-agent model realizes a white-box engineering of internal logics. In the ACPLT/PF prototype presented in [20], every Process Control Unit (PCU) should be implemented as an Atomic Function Block (AFB). The main limitation of this black-box design is that every variant of POU should be defined as a library class. Considering varieties of group control units and operation measures in process automation, it is difficult to avoid a class explosion. A workaround of ACPLT/PF presented in a student work allows the user to edit part of the internal procedures. This design only realizes a gray-box, which means that the internal logic can be partially acknowledged and manipulated by users. The white-box engineering outlined in Section 3.1 is still not supported. In contrast to ACPLT/PF, the FB-agent model allows a modular encapsulation of all its internal logics, a flexible combination of black-boxes and white-boxes and a service-oriented abstraction model (cf. Figure 3.1). Service interfaces of FB-agents are realized as internal function blocks of the agent frame. The transparent design of FB-agents ensures a user-friendly engineering. Internal logics of agents can be easily mastered and flexibly manipulated by users.

4 Usability Analysis of Existing Procedure Description Methods

Automation agents introduced in Section 2.5 and Chapter 3 need to observe internal execution status, recognize environment situation and formulate sequential operations. All these state-based functions or sequential functions can be called *procedures*, which define the strategy for carrying out process control processes in runtime systems. In the following sections, existing Procedure Description Methods (PDM) in industrial automation will be compared. Their usability for the FB-agent model presented in Chapter 3 will be evaluated.

Typical PDMs in industrial automation are Finite State Automaton, Petri Nets, Sequential Function Charts and Procedural Function Charts. An introduction and a brief comparison of the PDMs is given in the recommendation VDI/VDE 3681 [7]. In the following sections, the PDMs will be evaluated in detail. Specific requirements on the agent engineering outlined in Section 3.1 will be taken into consideration. Additionally, further PDMs (e.g UML Statechart and Grafcet) which were not introduced in the VDI/VDE 3681 but are important in process automation are also considered. All the regarded PDMs are evaluated according to the following criteria:

- **Completeness of syntax.** With the increasing complexity of automation functionalities, the requirement on the syntactical completeness of description methods has increased. A PDM is not suitable enough for comprehensive automation applications, if it is only composed of a set of simple steps and transitions. An overall solution is needed which can deal with actions, communication with the environment, and high level structures (e.g. alternative branching, hierarchy and concurrency).
- **Unambiguity of semantics.** Ambiguous semantics can lead to inconsistent results between the specifications of users and the implementation in runtime systems. As the user-centralized engineering is the central aspect of the FB-agent model, it is especially important to ensure that the semantics are clearly defined. Additionally, unambiguous semantics serve as a good base for the formal validation and verification.
- **Compatibility with existing automation systems.** As discussed in Section 3.1 and Chapter 3, FB-agents should be integrated into the existing automation systems. Thus, PDMs for agents should also be compatible with the control flow and

data flow in the automation systems. It will be evaluated, whether the PDMs can be implemented in the automation runtime environment. Additionally, to support the procedure description in Composed Function Blocks (cf. Section 3.5), it will be checked, whether the PDM can be composed of modular elements in runtime systems.

4.1 Finite State Automaton

Finite-State Automaton (FSA) (or finite state machine, state machine, sequential machine) is an abstract mathematical method for modelling the behavior of a system which is composed of a finite number of states. The word “automaton” can trace back to a Greek word which means “self-acting”. The automaton theory plays an important role in theoretical computer science. It is widely used in various application areas such as computer programming and sequential logic circuits.

In 1943, McCulloch and Pitts have presented an FSA approach for the description of neural nets [69] consisting of finite states. Based on this work, the final definition of FSA was established by Moore and Mealy [70, 71] in 1950s. Named after the authors, two kinds of FSA were defined: the Moore machine that generates outputs in states and the Mealy machine that generates outputs in transitions. A Moore machine can be converted into an equivalent Mealy machine and vice versa.

Syntax and Semantics

Finite State Automaton (FSA) consists of an input alphabet, an output alphabet, a set of states and a set of transitions. Inputs can be regarded as events. Outputs can be seen as actions, which are the consequences of events and state changes. States and transitions should be interconnected alternately, meaning that steps are followed by transitions and vice versa. Every FSA has an initial state. A state can be active or inactive. Concurrency is not supported. Only one state within a PDM can be active at a time.

Input \State	closed	opening	opened	closing
cmd_open	state opening			
fully_opened		state opened		
cmd_close			state closing	
fully_closed				state closed

Table 4.1: Equivalent State Table of the Finite State Automat in Figure 4.1

Figure 4.1 and Table 4.1 shows three equivalent description variants of the control logic of a switch valve: Moore machine, Mealy machine, and state table. The former two

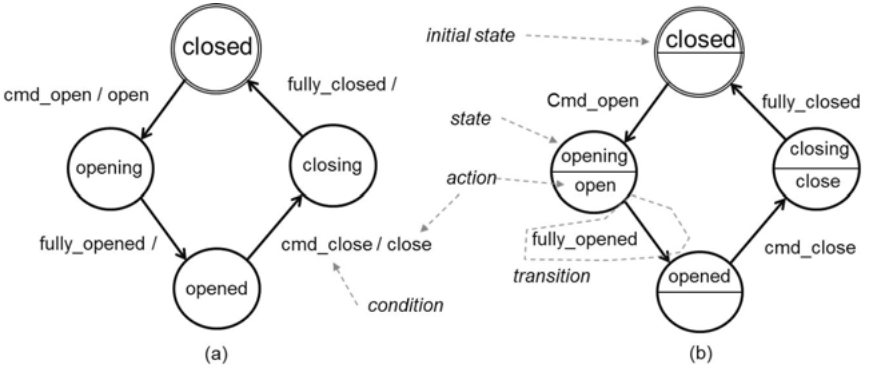


Figure 4.1: Equivalent Mealy machine (a) and Moore machine (b)

are graphical, whereas the last one is textual. The valve has two stationary states: fully open or fully closed. Their transitional phases are represented as two further states. The input alphabet of the FSAs is a vector of the conditions, whereas the output alphabet is a vector of the actions. In case an input alphabet is received and a condition is fulfilled (e.g. the command “cmd_open” asks to open the valve), the related action (e.g. “open”) will be emitted.

FSA is an early approach for the graphical description of procedures. Since FSA use simple elements and simple structures, their semantics can be easily formally defined and mathematically analyzed. In computer science, there exist many algorithms and tools for formal validation and verification of FSA.

However, the graphical notation of FSA can quickly become complicated and confusing, in case the described procedure has dozens of states. Due to the risk of state explosion, many commercial tools (e.g. S7-HiGraph for modelling control logics) are not well-established in the practical use.

FSA’s elements are abstract graphical notations. The inputs and outputs are clear text. It is not defined clearly in which format they can be implemented in the runtime system. Additionally, many semantics are not clearly specified. For instance, in case two outgoing transitions of a state are both fulfilled, it is not clearly defined which one of the conflicting transitions can fire. A prioritization rule is missing.

Application in Automation

The modelling method FSA is widespread in the practical use. In industrial automation, FSA is typically applied to specify PLC logic or evaluate an implementation [7]. FSA cannot be directly integrated into automation systems. For the implementation, FSA should normally be transformed into a textual programming language.

Usability Analysis

FSA is a simple but rigorous description method. It is suitable for describing simple procedures in FB-agents. Due to the risk of state explosion, it is also not suited to describe complex procedures. Due to the missing of implementation model for the runtime system, it is not suited to be applied to realized white-boxes, i.e. Composed Function Blocks (CFB, see Section 3.5).

4.2 Statechart

Statechart constitutes a further development of the Finate State Automaton (FSA). Among others, concurrency and hierarchy are supported.

Statechart was firstly presented by Harel in his work [72] for modeling discrete state transitions in reactive systems. On the basis of this work, many Statechart variants have been developed in the last decades, e.g. STATEMATE [73], PLC-Statecharts [74]. The UML specification [11] has defined an object-oriented variant which is also known as UML Statechart (also UML State Chart Diagram, UML state machine). The specification is released as an international standard IEC 19501 [75]. It has standardized the graphical notation of chart elements, a meta-model and the information exchange with the environment. Due to the normative nature of the UML, the term Statechart will now always refer to the UML Statechart, unless otherwise noted.

Syntax

Core elements of Statecharts are: transitions, events, actions, simple states, composite states and pseudo states (e.g. initiation, transition fork, memory for composite state etc.). An exhaustive introduction of all these extensions exceeds the scope of this dissertation. Instead, key elements will be introduced with the example Statechart in Figure 4.2.

As shown in Figure 4.2, *states* are marked as rectangles with rounded corners. States can be *simple* or *composite*. Simple states (e.g. State 1 and State 21) are atomic, whereas composite states (e.g. State 2 and State 3) contain subordinate Statecharts.

Every Statechart has at most one initial state and one final state. An initial state is shown as a solid filled circle. A final state additionally has a surrounding circle. In contrast to simple and composite states, initial and final states neither have state names nor actions.

A composite state may nest one subchart and build a hierarchical structure (e.g. State 2 in Figure 4.2). It can also nest many subcharts, which are processed concurrently. Every concurrent subchart (see State 3) is contained in a *region*. Neighbored regions are

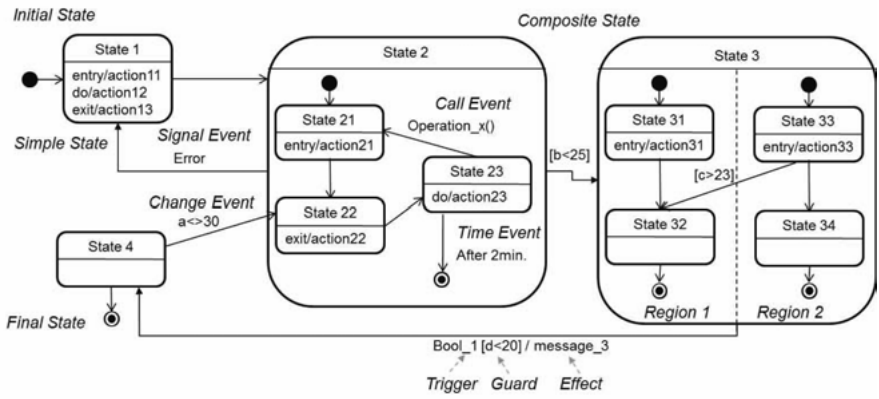


Figure 4.2: Elements of UML/Statecharts

separated from each other by dotted lines. Composite states realize a clear representation of complex structures and can significantly reduce the risk of state explosion. Composite states and concurrent regions are not encapsulated. A transition can connect two states in two regions directly (see the transition from State 32 and State 33 in Figure 4.2).

Every transition has one source state and one target state. Two states can be connected to each other via more than one transition. A complete transition has four parts:

- *arc* connects the source state and the target state,
- *trigger* is an incentive event for the transition. An event can be one of the following types:
  - A *Call Event*: a function is invoked,
  - A *Change Event*: the value of attribute(s) or association(s) is changed.
  - A *Signal Event*: a particular signal is received.
  - A *Time Event*: a deadline has expired. The deadline can be set relative to an explicit starting time, or to the time of entry into the source state.
- *guard* is a Boolean expression which allows ($TRUE$) or prevents ($FALSE$) the change of state,
- *action* will be performed, when the transition fires.

A transition can have more than one trigger and more than one guard.

Actions are the response of Statecharts on events. An action can be a variable assignment, function invocation or event generation. Statecharts merge the design of Moore machines and Mealy machines. It means that actions can be generated both

in transitions (e.g. the message generation in the transition from State 3 to State 4 in Figure 4.2) and in states (e.g. all simple states in Figure 4.2).

Every action should have a type which is represented as clear text before a slash and defines the execution behavior of the action. For instance, an *entry* action will be executed when the state is entered; a *do* action will be continuously executed and an *exit* action will be executed when the active state is left. A transition action is equivalent to an entry-action in the target state of the transition.

Semantics

A state can be active or inactivate. When a state is active, its outgoing transitions are also activated and will be evaluated continuously. When a transition fires, its source state will be deactivated, whereas its target state will be activated. In case a transition crosses the boundary of a composite state from the outside and fires (e.g. the transition from State 4 to State 22 in Figure 4.2), not only its target state (e.g. State 22) but also the covering composite state (e.g. State 2) is activated.

In case a state is left, the Statechart progresses until a new stable state is reached and no transition can fire. Theoretically, the progressing is seamless and timeless. It means, the new stable state is reached immediately, even if the old state and the new state are separated by a sequence of intermediate states.

In case a transition ends at the boundary of a composite state (e.g. State 2 in Figure 4.2), the subchart starts from its initial state by default. A final state defines the default ending of a subchart. In case the subchart of State 2 reaches is finals state, the chart is terminated. The State 2 keeps active until one of its outgoing transition fires. The subchart in a composite state can be aborted when an outgoing transition fires. In case the State 2 is active and the variable b is smaller than 25, the transition to state 3 will fire immediately, irrespective of the state of the subchart.

All composite states in the example have no *memory*. This means that, in case a composite state is deactivated, its sub-charts will be reset. In case the covering state is activated again, the sub-charts start form their initial states. A composite state may have also memory. In case the composite state is deactivated, it stays on its current state. The execution proceeds from the current state when the composite state is reactivated.

Transition triggers are to be checked continuously. A guard will be checked only if the triggers of the same transition are fulfilled. The transition can fire if its trigger and guard are both fulfilled. For instance, the semantics of the transition from State 3 to State 4 in Figure 4.2 can be described as follows: If the variable $Bool\_1$ is $TRUE$ and d is smaller than 20, the transition will fire. State 3 will be deactivated. State 4 will be activated. Message _3 will be emitted.

Application in automation systems

Statechart is a classic modelling language in computer science and is not widespread in industrial automation. One main reason for this is that many definitions may lead to an over-engineering of procedures. Similar to further modeling languages specified in UML, Statechart does not specify exact execution behaviors in the operative environment deliberately. These are left to the system developer. For instance, the syntax of guard, event and action is not clearly defined; semantics for the evaluation order of guards starting from the same state are also not given.

Classic Statecharts cannot be directly implemented in automation systems. [74] has introduced a variant named PLC-Statechart, which allows a graphical programming in PLC. The programming language PLC-Statechart simplifies the UML Statechart and borrows many design principles (i.e. action concept) from another programming language called Sequential Function Charts (SFC, see Section 4.4). PLC-Statecharts support only a specific part of the syntax of classic Statecharts. Among others, only signal events are allowed. Although PLC-Statecharts ultilize the action concept of SFC, it has not been defined explicitly how complex actions (e.g. mathematical calculation) are to be defined. All example actions in relevant publications are only signal assignments.

PLC-Statechart is formally defined. A mathematical model of PLC-Statecharts is presented in [18]. This model makes the formal validation and verification of PLC-Statecharts possible. However, semantics in this model are also not complete. For instance, semantics of composite states are not explicitly defined. Execution behaviors (e.g. prioritization rule) of complex actions are also not defined.

Usability Analysis

As a successor of classic Finite State Automata (FSA), the syntax of Statecharts has been extended. Among others, hierarchical or concurrent substructures can be realized as composite states. The complexity of a flat structure without hierarchy and the risk of a state explosion can be significantly reduced. Additionally, events which describe the interaction with the environment are defined.

Statecharts inherit the advantage of formal analyzability from FSA. Many analysis methods for Startcharts have been developed on the basis of the formal analysis of FSA. As a formal language, Statechart is very suitable for the specification of logics in FB-agents. However, similar to FSA, Statechart is also a visual description method and not suitable for the implementation in runtime systems. The main restriction is that the event-driven execution behavior is not compatible with the cyclic processing environment, which plays a central role in the process automation system.

The variant PLC-Statechart is compatible with the cyclic-processing context according to the IEC61131-3. In principle, this approach can be applied as one possible imple-

mentation approach for Composed Function Blocks (CFBs). However, semantics of PLC-Statechart are still to be complemented.

4.3 Petri Net

Petri nets (PN) were published by mathematician Petri in his PhD thesis [76] in the early 1960s. On the basis of this work, many types of PNs have been developed. PN is a very important description method in computer science. PNs are widely used in concurrent programming, workflow management, data analysis etc. Their main strength is that they are very suitable for the description of concurrency, especially in decentralized systems.

Syntax

As shown in Figure 4.3, Petri nets are composed of *places, transitions, arcs* and *token*. Places are drawn as circles, transitions as bars or rectangles, and tokens as solid circles. Arcs are directed and can connect either a place with a transition, or a transition with a place.

Tokens can move from one place to another. The maximal number of tokens that a place can store is called *capacity* of the place. The number of tokens that pass an arc every time can also be limited by the so-called *weight* of the arc. Unless explicitly defined, the capacity of a state is infinite, and the weight of an arc is one by default. The distribution of tokens at a certain time is called one *marking* of the net. The token distribution from which a Petri net starts to progress is called *initial marking*.

A Petri Net is *ordinary*, in case the weight of all arcs is one and all transitions are *trivial*, which means every transition has exactly one source place and one target place. A Petri Net is *general*, in case the arc weight can be greater than one. Many high-level PNs have been developed on the basis of general Petri Nets, for instance:

- Colored PN: Tokens are categorized. Tokens of different categories are marked with different color.
- Timed PN: Arcs of transitions can have a delay, so that the movements of tokens do not take place immediately. Every token has an own time stamp which indicates the creation time of the token.
- Hierarchical PN: Places can nest sub-nets.
- Signal Interpreted Petri Nets (SIPNs) advanced ordinary Petri Nets which support input and output signals, cyclical processing, global execution time and hierarchy. Every SIPN should have only one initial token. A multiplication of tokens in split (or simultaneous) sequences is allowed.

Detailed Introductions of the classification of Petri Nets can be found in [77, 78].

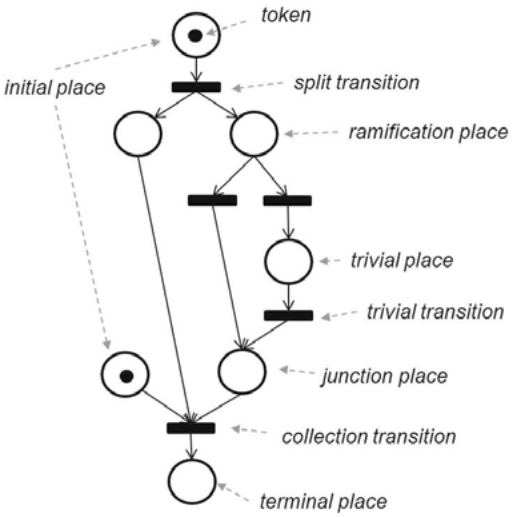


Figure 4.3: Example Petri Net

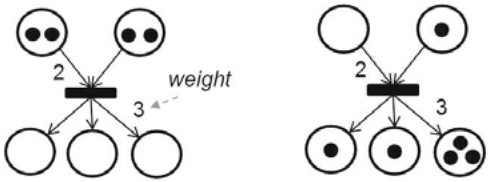
Semantics

Many theoretical work focus on the mathematical model, the network rules and the notation of Petri nets. In fact, the graphical visualization of networks is not the basic idea of Petri Nets. PN can be regarded as a physical theory proposed in the language of computer science. Many mechanisms are carefully designed according to the “natural laws”.

For instance, Petri-Nets do not follow any central iteration. There exists neither global time nor central schedule for the net elements. All transitions are standalone and are - the same as in the nature world - always ready to react to events immediately. Any fulfilled transition can fire at any moment without regarding the progress of the whole net. The global state of a PN is not represented by a single active state, but by the marking (i.e. token distribution) in the net.

In contrast to Finite State Automata and Statecharts, the state activity (or state mark) is not “transfered” from one state to another state in a Petri Net. As shown in Figure 4.4, Petri nets’ transition “consumes” as many tokens from its source places as the weight of its input arcs, and then “produces” as many tokens in its target places as the weight of its output arcs. During the consumption and the production of tokens, the total number of tokens in the PN is not kept constant.

Precise mathematical models and analysis methods have been introduced in different literature resources. For instance, a formal model is presented in [78]. These theoretical



(a) Beforehand **(b)** Afterwards

Figure 4.4: Consuption and Creation of Tokens in Petri Nets

studies can support a formal analysis of Petri Nets. Additionally, a Meta-model and a XML-based interchange format PNML for Petri net tools are specified in [79, 80].

On the basis of the mathematical models, the following properties of a PN can be formally analyzed:

- *Reachability* describes whether a marking can be reached from the initial marking.
- *Liveness* indicates the possibility of a deadlock during the progression of a Petri net. A net is *dead*, if no transition can fire. A net is *alive* when it cannot be dead for all possible markings.
- *Boundedness* gives the maximal number of tokens in places. A place is k*-bounded*, when it cannot contain more than k token for all markings. A Petri net is k*-bounded*, when all its places are k*-bounded*.
- *Reversibility* indicates, whether the initial marking of a net can be reached from any other markings.

These properties can be analyzed with various methods, such as coverability graph [81], reduction rules [82], finite net unfoldings [83, 84].

Application in Automation

Petri Net (PN) is very suitable for the specification, fine design, formal analysis and verification of automation functionalities, but less suited for implementation [7]. The event-driven progression without global scheduling of net elements is not compatible with the cyclic processing context in existing automation systems. In the practical use, a procedure modelled by a PN should normally be compiled into another language that can be executed in the target runtime system.

The variant SIPN can model timed behaviors in automation systems. As an early approach for describing automation logics, the syntax and semantics of SIPN are still simple and incomplete. Among others, only simple actions (e.g. variable assignement) are supported. Mathematical operations are normally not allowed; prioritization rules for conflict transitions (i.e. outgoing transitions from the same place) are not specified.

On the basis of Petri Nets, further approaches such as Grafcet (see Section 4.5) and SFC (see Section 4.4) are developed especially for the industrial automation. Many designs of Petri Nets (e.g. token mechanism) are inherited by the latter approaches.

Usability Analysis

The main strength of Petri Nets (PN) is the token-driven mechanism which allows a continuous execution and can appropriately describe concurrent behaviors. However, the execution principle is not compatible with the cyclic processing context, which is dominant in process automation. For the implementation of FB-agents, a strict deterministic execution of internal logics is required. All internal components should be sequentially executed according to a fixed order. In this context, the strength of PN becomes invalidated, whereas the shortcuts will limit their usability.

In principle, the variant SIPN can be applied to model internal logics of FB-agents. However, it can also not be directly implemented in automation systems without a special code generator. Thus, SIPN is not suitable for implementing Composed Function Blocks (CFB) in FB-agents.

4.4 Sequential Function Chart

Sequential Function Chart (SFC) is a procedure description method specified in the IEC 61131-3 [3] for the PLC programming (see also Section 2.1.2). SFC was developed on the basis of the work conducted by Levis [85] and standardized in the IEC 61131-3. SFC is usually regarded as a graphical language. In fact, a textual representation has also been given in the standard.

4.4.1 Syntax

PLCs were developed to model and replace analog circuit (hardware) in a software environment. For this historical reason, many PLC languages such as FBD and SFC, have adopted many working principles of electric circuits. Among others, the electrical wires are mapped as software signals and connections. SFCs communicate with the environment via signals. The information exchange between SFC elements (e.g. steps and actions) is also realized via signals. For instance, the name of an SFC action (e.g. a1 in Figure 4.5) is to be implemented as a Boolean variable (e.g. a1=FALSE) in the runtime system. This variable controls the activity of the action similar to an on-off switch in an electrical circuit. In case the action is to be executed, e.g. x:=5 in Figure 4.5, the Boolean variable a1 in the runtime is set to TRUE.

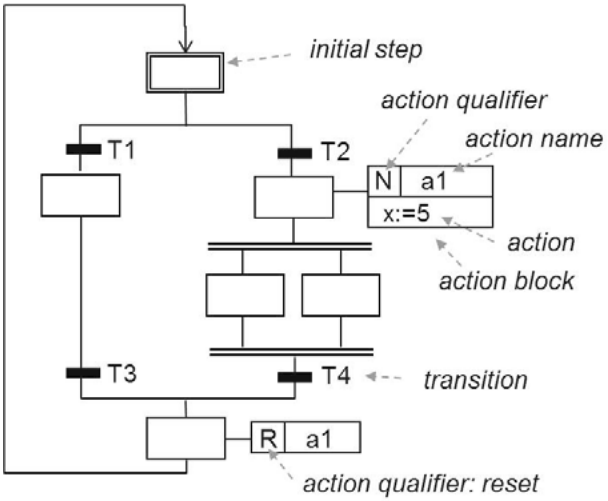


Figure 4.5: Sequential Function Chart (SFC)

As shown in Figure 4.5, SFCs are composed of steps and transitions. Every SFC has exactly one initial step. The ending of SFCs is not explicitly defined in the standard. Example SFCs in the standard and in literature have no final steps. At the end of the chart, the last step jumps back to a previous step. Final steps are allowed in many commercial tools. For example, SFCs in the DCS SIEMENS PCS7 should end with precisely one final step; PLCs of SIEMENS apply the syntax of final node which is borrowed from Statecharts.

Action and Action Block

Steps can perform two general kinds of actions: the *Boolean action*, which sets or resets a Boolean variable and the *non-Boolean action*, which can be implemented in any IEC 61131-3 language. Non-Boolean actions can be *local* or *global*. Local actions can only be invoked by their corresponding step, whereas global actions are shared and can be invoked by different steps. Global actions are a useful design for the practical use. In case the same action is used in different steps, it can be defined as a global action. In case its logic is changed, the logic of all related steps does not need to be adapted accordingly.

Every action should be associated to its corresponding step via an *action block* which defines the action name and the so-called *action qualifier*. The latter one defines the execution behavior of the action. According to the IEC 61131-3 standard, the following action qualifiers are available:

- N: non-stored. The action is active as long as the step is active
- R: overriding reset. The action is deactivated.
- S: stored (set). The action is active, even if the step is deactivated.
- L: time limited. The action is active for a certain time.
- D: time delayed. The action will become active at a specific time after the step has been activated.
- P: pulse. The action is executed once.
- SD: stored and time delayed
- DS delayed and stored
- SL: stored and time limited
- P1: pulse (rising edge). The action is executed once when the state is activated.
- P0: pulse (falling edge). The action is executed once when the state is deactivated.

Transition

All SFC transitions have a direction. The transition arc is normally hidden. It will be shown, only if the transition targets a previous step.

In contrast to Statecharts, SFC transitions are not allowed to invoke actions. Moreover, triggers and guards of a transition are consolidated into one *transition condition* which provides one Boolean evaluation result. When this result is evaluated as TRUE, the transition can be fired. A transition condition can be a Boolean variable. It can also be defined in Instruction List (IL), Structured Text (ST), Ladder Diagram (LD) or Function Block Diagram (FBD).

Complex Structures

SFCs allow complex structures such as alternative path, concurrency and hierarchy. Alternative path means that a step is followed by more than one successor transition. Only one of them can fire. Concurrency is realized by simultaneous sequences, which means a transition can also be followed by more than one simultaneous sequence starting from the same step. Hierarchy is not explicitly defined in the IEC 61131-3. In the practical use, the following two approaches are often regarded as the de-facto solution for realizing hierarchical SFCs:

- *SFC-action*: An action can be realized as an SFC and be invoked by a step of another SFC. This design is introduced in the IEC 61131-8, which specifies the implementation guideline for the IEC 61131-3 standard.

- *Macro step*: A macro step represents a particular part of an SFC. A macro step behaves similar to a normal step, but contains an underlying sequence. The contained sequence starts with a step and ends with a step. Macro steps can be seen as a simplified graphical representation for complex sequences. An SFC with macro steps can be expanded to an equivalent "flat" SFC, i.e. an SFC without hierarchy. This solution is actually not defined for SFC but for Grafcet (see Section 4.5). Since some widespread SFC-editors (e.g. CoDeSys) support this mechanism, it is often regarded as a standard structure of IEC61131/SFC.

.

4.4.2 Semantics

The SFC is informally specified, which means no standard mathematical model is defined in the IEC61131-3. Models in many literature sources (e.g. [86]) are normally vendor-specific or developer-specific.

The SFC is ambiguously defined and cannot be formally analyzed [87, 88]. The main reason is that the standard does not mandate a unified solution, but recommends many alternative and even conflicting solution approaches. After comparing various implementation of major PLC vendors, [89] has demonstrated that in case one solution approach is chosen for single semantic aspects (e.g. prioritization rule for transitions), the SFC implementation can be formally analyzed.

Step and Transition

In similarity to FSA and Statecharts, the source step of a fired SFC transition will be deactivated, and the target step of the transition will be activated. Every step must have two standard variables: activity $step\_name.X$ and time $step\_name.T$. The former one will be set or reset, in case the step is activated or deactivated. The latter one indicates the time since the step has been activated. It will hold its value when the step is deactivated, and be reset when the step is activated again.

In case a step is followed by more than one transition, only one of them can fire. The IEC61131-3 has defined three rules for the prioritization of conflicting transitions:

1. Priority from left to right,
2. Priority must be explicitly defined by user, or
3. User must ensure that only one transition can be satisfied.

Simultaneous Sequences

The semantics of simultaneous sequences are similar to the token multiplication of Petri Nets. In case a transition is fired, all its successor steps are marked and can be activated.

The multiplication of step marks may lead to unexpected execution results. The IEC61131-3 has defined two implausible structures: unsafe sequence and unreachable sequence. The former one causes uncontrolled mutilation of step marks in the whole SFC. The latter one can never be executed. Examples of the implausible structures can be found in the standard and relevant literature and will not be discussed here in details. The easiest and the most effective way to avoid implausible structure is to forbid jumps out of a simultaneous sequence or a sequence after an alternative branching. However, as indicated in [86], some structures cannot be realized, although they are valid. As an alternative solution, algorithms for the reachability analysis of Petri Nets can be borrowed to evaluate SFCs.

Scheduling of SFC Elements

There exists no standardized model for the scheduling of SFC elements during the execution. In general, SFC elements are executed according to the rules of “from top to bottom” and “from left to right”. Additionally, SFCs are executed according to the *Lock-Step* principle [88]. This means that in case a step is activated in a cycle, it cannot be left (i.e. deactivated) in the same cycle.

Execution models of SFCs have been discussed in different research works. The main differences of the models are the execution of the following three execution activities:

- Evaluation of transitions
- Update the activity of steps
- Sequential execution of actions

Different arrangement of the listed three execution activities may also lead to different results. An execution model named *deferred transition evaluation and deferred action (DT-DA model)* is introduced in [90]. According to this model, transitions should be evaluated prior to the action execution. Fulfilled transitions can only fire when all transitions are evaluated. Actions can be executed when the activity of all steps is updated. As introduced in [88], many commercial SFC implementations follow this model. The main disadvantage of this design is that actions of the initial step cannot be executed, when an outgoing transition is fulfilled at the beginning of the SFC execution. Discussions and analysis with more details can be found in [86, 88, 91].

An alternative prioritization rule is the *immediate action and immediate transit evaluation (IA-IT model)* [90]. According to this model, actions should be executed prior to transitions in every cycle. In case a transition is fulfilled, it must fire immediately.

The activity of its source step and target step should also be updated immediately. As analyzed in [90, 92, 93], the IA-IT model can react on inputs (e.g. value change) as quickly as possible and can avoid unexpected execution results effectively. It ensures, among others things that actions of the initial step can be executed correctly. The disadvantage of this model is that it is not easily implementable in runtime systems that are cyclic-processing [90]. An execution model for the implementation should be elaborately defined.

Additionally, the execution order within the three single activities is also differently realized in research works and commercial tools. For instance, the most commercial SFC tools execute actions from top to bottom. However, in the widespread PLC runtime system CoDeSys, actions are executed according to their alphabetical order. Additionally, every action will be executed twice. This design may lead to different execution results than the other SFC tools. Detailed analysis and execution examples can be found in in [88, 91].

4.4.3 Application in Process Automation

SFC is defined as a programming language for PLC and is well-established in the practical use. It is also suited for further design phases, like specification, rough design, and simulation [7]. In many DCS projects in the process industry, procedures are specified, implemented and documented in SFC without using further Procedure Description Methods (PDMs).

As a standardized PDM in automation, the development and the application of SFC are supported by various works of international committees and researchers. For example, the international organization PLCOpen is in charge of the compatibility certification of IEC 61131-3 for automation systems. PLCOpen has also specified a XML format for the information exchange between systems from different vendors.

Due to the native support of existing automation systems, a procedure specified in SFC can be directly implemented in the existing automation systems without a special code generator.

4.4.4 Usability Analysis

The SFC is designed according to the software and hardware characteristics of the existing automation systems. All syntax and semantics are compatible with the cyclic-processing and signal-oriented execution context.

Due to the native support of existing automation systems, SFC is suitable for designing and implementing various components within FB-agents. It can be applied as a textual programming language for the implementation of black-boxes (i.e. Atomic Function

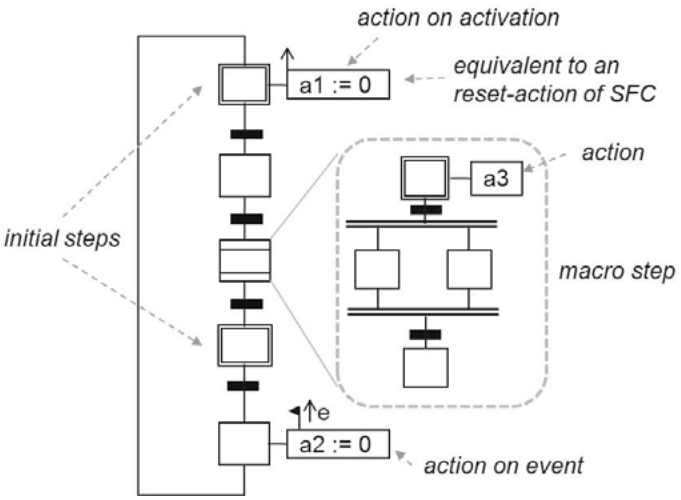


Figure 4.6: Grafcet

Blocks (AFBs). It is also suitable for application as a graphical language for Composed Function Blocks (CFBs).

4.5 Grafcet

Grafcet is a specification language developed in 1977 for the design of procedures in industrial controllers. Grafcets are the predecessor of SFC introduced in Secion 4.4 and can be regarded as high-level Petri Nets.

The name Grafcet was derived from “graph” and “AFCT”. The former part implies that this Procedure Description Method (PDM) is graphical. The latter part is the acronym of the Associtiaon Francaise pour la Cyberneticu Economique et Technique, which supported the development of Grafcet. The Grafcet was firstly published as a French national standard in 1982, and later as an international standard IEC 60848 [94] in 1988. In many publications, Grafcet is often translated as Sequential Function Chart in English. Hence, Grafcets and SFCs are are often regarded as synonymous in many research works.

An example of Grafcet is given in Figure 4.6. Since the syntax and semantics of Grafcet are very similar to SFCs, only their main differences will be introduced in the following lines:

- The Grafcet is a specification language, whereas the SFC is a programming language. A sequential procedure can be specified in Grafcet and can be imple-

mented in different ways. An SFC can implement a procedure that can be specified in Grafcet or other description methods.

- Grafcets are to be executed periodically, but not strictly cyclically. In case a Grafcet leaves its current step, it reaches a stable state immediately, even if the start step and the target step are separated through many further steps and transitions. This mechanism is termed *Maximum Progress* in [88]. However, SFCs should be cyclically executed and can only move for one step (see Lock-Step in Section 4.4.2).
- A Grafcet may have more than one initial step; an SFC should have only one initial step.
- Grafcet applies *Macro Steps* to realize hierarchical nesting. This design is also utilized to design hierarchical SFCs, although it is not explicitly defined in the IEC61131-3.
- Grafcet has 7 action control conditions, whereas SFC has 11 possible conditions (action qualifiers).
- Grafcet-actions are value assignments. The SFC supports more complex actions, e.g. Function Block Diagramms (FBDs) for mathematical calculations.
- In Grafcet, outgoing transitions of a step must be mutually exclusive, which means only one transition condition can be fulfilled. However, SFCs allows two further alternative approaches (cf. Section 4.4).

As an informally defined language for specification, Grafcet has no standardized execution model. The formal analysis of Grafcets is not intensively discussed in literature resources. [95] refers to a formal model of Grafcets. However, the origin and tool support of the model are not indicated.

Grafcet and SFC can be seen as two different expressions of the same logic in different life cycle stages: Grafcet for specification and SFC for programming. In the author's opinion, the difference between specification and programming in the context of process automation is not great enough to make two different description methods strictly necessary. As introduced in Secion 4.4, SFCs are applied to specify procedures in practical use. Grafcets are actually replaced by SFCs. This option can also be appropriately supported by the development trend in the technical community. Grafcets were intensively discussed in the last century, but are rarely seen in publications of the last decade since the IEC 61131/SFC was published.

Grafcet is only rarely applied in the industry, whereas its successor SFC is well-established. Hence, Grafcet is not suggested for the design of FB-agents.

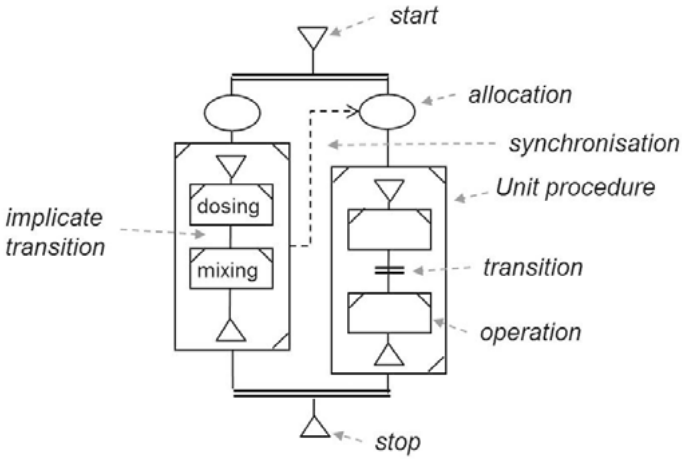


Figure 4.7: Procedural Function Chart(PFC)

4.6 Procedural Function Chart

Procedural Function Chart (PFC) is a specific description method for batch recipes in the chemical industry. PFC is derived from Grafcet and is standardized in the IEC 61512 [9]. PFC has inherited most of its syntax from Grafct (See Section 4.5). In the following section, only the main differences of the two description methods will be introduced.

Syntax

PFCs are composed of elements and transitions. To model different procedure elements, PFC elements are classified into four types: procedure, unit procedure, operation and phase. Additionally, another element for the allocation of production equipment is defined and can be applied in different procedure elements. Furthermore, the syntax for the synchronization between elements is defined (cf. Figure 4.7).

PFC transition can be implicit or explicit. An implicit transition is permanently TRUE. As shown in Figure 4.7, the successor step and proceeding step of an implicit transition are connected directly. An explicit transition is drawn as two short parallel bars and possesses a transition condition.

PFC has inherited the macro-steps from Grafcet. The single difference is, a PFC element may only contain lower-level elements. For instance, a unit procedure cannot contain procedures; an operation cannot contain unit procedures.

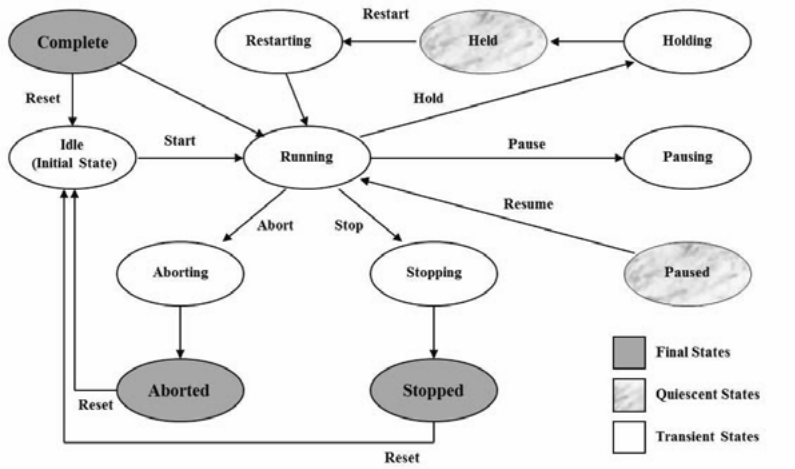


Figure 4.8: State transition diagram for PFC elements according to the IEC 61512 [9]

PFC actions are not explicitly defined in the IEC 61512 standard. In the most commercial tools (e.g. ABB 800xA Batch, Siemens Simatic Batch), PFC elements can assign variables (e.g. the activity variable of a dosing function in a PLC). Complex actions (e.g. mathematical calculations) are normally not supported. In case a calculation is required (e.g. calculate the current material consummation during a batch), it is to be implemented as a special element in the PFC or outside of the PFC (e.g. as an ANSI C program).

Semantics

PFC elements are self-ending and self-determinated. This means that an element can start only if its previous element is finished. Although the operations “heating” and “mixing” in Figure 4.7 are directly connected, the latter operation should wait until the first operation is finished.

In case the condition of an explicite transition is fulfilled, the first bar is marked as fireable. The successor step is asked to break down its execution. Only if this mission is finished, the second bar and the whole transition can fire.

To prioritize conflicting transitions in alternative sequences, PFC borrows the first prioritization rule from SFC (see Section 4.4). Alternative sequences should be evaluated from left to right. The firstly fulfilled transition can fire.

The execution of every PFC element is to be controlled by the standard *state transition diagram* shown in Figure 4.8. A PFC element can be running, held, aborted, stopped or complete etc.

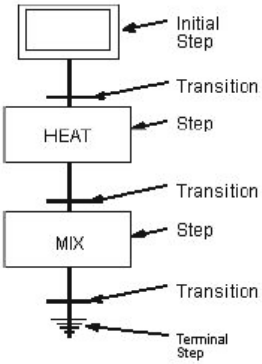


Figure 4.9: Example PFC in the documentation of the commercial tool Emerson/DeltaV Batch

Industrial Application

PFC is an exclusive description method for batch procedures. It simplifies the complex Grafcet and utilizes many features of SFC. Among others, the complicate and unintuitive action control of Grafcet is excluded. Many semantic details (e.g. prioritization of hierarchical PFCs) are clearly defined.

There is still no standard execution model which is accepted by the researchers and commercial vendors. The IEC61512/PFC is only partially implemented in commercial automation systems [7]. For instance, the double lined transition and the synchronization of materials is not supported by the most major vendors.

The implementation and graphical visualization of many PFCs in commercial tools are often more similar to SFC, but less similar to the IEC61512/PFC. Figure 4.9 shows one example PFC from the PFC document of a major DCS vendor. Due to the SFC-like syntax, users cannot easily recognize the differences between PFCs and SFCs in their system.

PFC is the only procedure description method which has explicitly defined the state machine for controlling the execution of procedure elements. The standard state machine in 4.8 is also applied in some SFC implementations (see for instance the Advanced Process Library (APL) of the DCS vendor SIEMENS).

Usability Analysis

Similar to Grafcet, PFC is also defined as a specification method. However, PFC has simplified and extended the over-complex Grafcet, and has specified many semantic details in accordance with the programming language SFC. Due to the consideration of

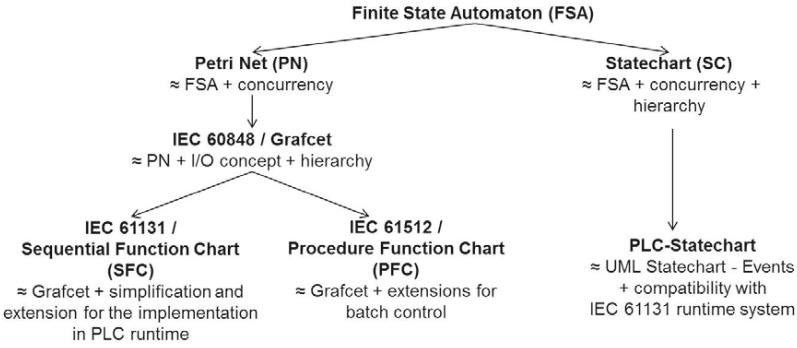


Figure 4.10: Development tree of Procedure Description Methods (PDMs) (revised illustration according to Figure 1 in [19])

runtime behaviors, PFC is also suited for the implementation. PFCs can be applied to specify and implement batch procedures in FB-agents.

However, syntax (e.g. element types) and semantics (e.g. hierarchical execution) are specifically defined for batch automation. The usability of PFCs will be significantly limited beyond the batch control area. Among others, a concept for actions should be defined.

4.7 Summary

Figure 4.10 summarizes the relationship and main differences between the analyzed Procedure Description Methods (PDMs).

Procedure description methods can be classified into two types: procedure specification (or modelling) method and procedure implementation method. The former one describes procedures generally and abstractly. The latter one is suited for the implementation of procedures in runtime systems.

Typical procedure specification methods are Finite State Automaton (SA), Statechart (SC), Statechart (SC), Petri Net (PN), Grafcet and Procedural Function Chart (PFC). They can be applied to draft and document procedures in FB-agents. For the implementation in runtime systems, alternative programming languages should be applied. Procedures in Atomic Function Blocks (AFB) can be implemented in a textual programming language such as Structured Text. Procedures in Composed Function Blocks (CFB) can be implemented with the two procedure implementation methods: Sequential Function Chart (SFC) and PLC-Statechart.

From the viewpoint of practical application, procedure specification and procedure implementation have no restrict dividing line between them. Firstly, execution behaviors in runtime systems should also rather often be defined in the procedure specification. For instance, the execution prioritization of hierarchical procedures is an implementation detail, but should also be clearly defined in the specification (cf. discussions on hierarchical Statecharts and PFCs). Secondly, implementation methods are often also suited to specify procedures in practical use. For instance, the programming language SFC is often applied to specify and document procedures.

There are also no convincing arguments for using different description methods for specification and implementation. As the compatibility with existing automation systems is one central requirement on the agent engineering, the two implementation-friendly approaches Sequential Function Chart and PLC-Statechart are chosen as suitable description methods for the FB-agent model. Sequential procedures (e.g. start-up procedure for a plant) can be described in SFC, whereas PLC-Statechart is suitable for the description of state machines.

Additionally, Procedural Function Chart (PFC) is also a suitable approach for FB-agents. Although PFC is a specification method, it has defined many semantic details that are important and helpful for the implementation in runtime systems.

Although SFC, PLC-Statechart and PFC have their own strengths and can appropriately describe procedures in different application areas, they can only partially fulfill the requirements defined at the beginning of this chapter. PFC is specially tailored for batch control and is only partially supported by commercial tools due to its high complexity. Its usability in other application areas is limited. PLC-Statechart lacks many design details, e.g. semantics for composed states. The present form of SFC defined in the IEC 61131-3 has ambiguous semantics. Some of its design (e.g. action control condition) is still over-complex. Thus, SFC is also not very suited to be applied as a general approach for the procedure description in different application areas and engineering phases.

The usability of the FB-agent model can be significantly improved, in case the user only needs to master one general approach for procedure description. This discussion will be continued in the next chapter.

5 Specification of a General Procedure Description Method

The introduction in Chapter 4 shows that the existing Procedure Description Methods (PDM) are only partially suitable as a general approach for describing various procedures in automation agents. In order to improve the intuition and usability of the FB-agent model presented in Chapter 3, a general procedure description method will be proposed in this Chapter.

Most automation procedures in process automation are not over-complex. For instance, a typical control procedure consists of 15 steps and 5 actions per step. These kinds of procedures with no complex structure and no exceptional functions will be termed *Ordinary Procedures* in the following sections. Most ordinary procedures in process automation can be appropriately described by a general approach without using special PDM. The engineering workload of users can be significantly reduced. Although the PDMs evaluated in Chapter 4 have different development backgrounds and different application areas, their basic structure and topology of PDMs are essentially the same. The following essential similarities of PDMs make the development of a general PDM feasible:

- A PDM is composed of states (or steps, elements, places) and transitions between them. States and transitions are interconnected via arcs. In case a transition fires, its source state will be deactivated; its target state will be activated.
- A procedure is a software module with individual name space for internal elements (e.g. state, variable). Procedures should exchange information with their environments. I/O concept (e.g. signal, event) should be explicitly defined.
- Procedures are reactive, meaning that they can perform actions which are the response on inputs.
- All description methods should provide solutions for dealing with alternative sequences, hierarchy and concurrency. Many design details are application-neutral and can be generically defined. For instance, Sequential Function Chart has summarized different rules for the prioritization of conflicting transitions. This collection forms a good guideline for the development of further description methods.

Every PDM has its own strength and can suitably describe procedures in specific application areas. It is neither meaningful nor feasible to develop a novel approach

which can replace all existing procedure description methods. However, a general PDM can serve as a reference model which identifies the most appropriate solution for single design details and combines advantages of existing PDMs. Additionally, a general approach can unify the engineering process, graphical visualization and the implementation of the ordinary procedures in automation systems.

To ensure the usability of the general procedure description method, the following key design principles will be regarded in the following sections:

- **Completeness**. The general PDM should provide complete solutions for dealing with actions, hierarchy, concurrency and etc.
- **Unambigurity**. Semantics of the general PDM should be clear and easily understandable. A unified solution is to be defined for every semantic issue (e.g. prioritization rule for conflicting transitions).
- **Compactness**. The general PDM should be designed as compact as possible. Many design features of the PDMs are grown historically or developed for the description of application-specific behaviors. Many of them are too complex and not well-established in the practical use (see for example the action control conditions of Grafcet (Section 4.5) and the data synchronization of Procedure Function Chart (PFC) (Section 4.6)). A compact procedure description method can appropriately support the work of users who have no in-depth knowledge in the areas of software engineering and computer science.

The need of a general procedure description method was also addressed in a previous publication [19] of the author. An approach named Sequential State Chart (SSC) was drafted. As the name implies, SSC is developed on the basis of the programming language Sequential Function Chart (SFC) and the specification language Statechart (SC). In the following sections, SSC will be revised and further developed. Many design decisions will be reasoned more precisely. The engineering requirements on PDM for FB-agents outlined in Chapter 4 will be of major concern. Additionally, the engineering requirements on FB-agents outlined in Section 3.1 will be taken into consideration. Among others, the requirement on white-box engineering will be carefully regarded during the SSC design.

It should be noted that the graphical notations of SSCs are only recommendations. They can be individually tailored for the implementation in different systems.

5.1 Execution Frame

Most PDMs have not explicitly defined the implementation form of procedures in the runtime system. It is usually assumed that a procedure is encapsulated in an *Execution Frame* which isolates procedures elements from the environment. To gain a complete

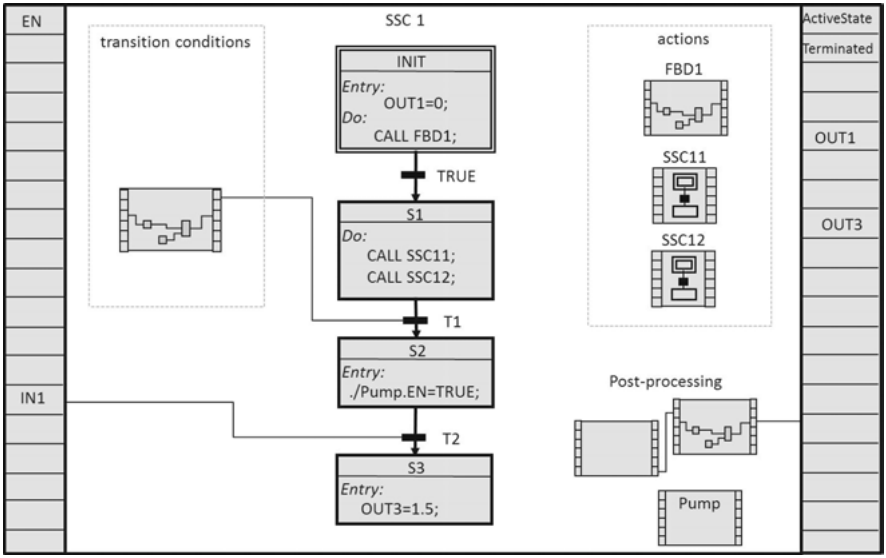


Figure 5.1: Execution frame and internal structure

execution model, an execution frame is necessary for the general description method SSC. The execution frame and the procedure elements build an inseparable module.

Additionally, the execution frame can adequately fulfill the requirement regarding modularity of internal components within FB-agents. Form the outside, an SSC can be handled as an encapsulated module with its own name space for internal components and variables. Within the execution frame, the logic can be composed of modular components which are linked via signal connections to each other.

As shown in Figure 5.1, every SSC is to be encapsulated as a Composed Function Block (see Section 3.5). SSCs consist of states and transitions. States can invoke actions. Transition conditions can be realized as variables (e.g. $IN1$ for $T2$) or function blocks (e.g. $T1$). All internal components within the execution frame are to be realized as function blocks.

Every execution frame possesses a standard input EN and two standard outputs $ActualState$ and $terminated$. EN controls the activity of the SSC. $ActualState$ shows the current active state. $terminated$ indicates whether the procedure is finished.

To control the execution of SSCs, the standard state machine shown in Figure 4.8. is applied. EN can be set to the commandos: $STOP$, $START$, $PAUSE$, $HOLD$, $ABORT$, $RESUME$ or $RESET$.

Every execution frame has a standard output $ActualState$ indicates the current state.

Aside from the procedure that is composted of states and transitions, continuous functions can also be defined in the execution frame. Continuous logics are termed *Post-processing* and will be cyclically executed at the end of every execution iteration without regarding the progress of the SSC procedure. Post-processing can be used to perform continuous calculations in parallel to the discrete procedure. For instance, the post-processing for a batch recipe procedure can be utilized to calculate the current material consummation.

Similar to the FB-agent frame introduced in Section 3.5, a deterministic execution is also required in the execution frame of SSC. The execution of states, transitions, transition conditions, actions and post-processing are to be controlled by an internal *Execution List*. Details of the execution list will be introduced in Section 5.8. In the following sections, single elements of the SSC model will be introduced.

5.2 State

The progress of a procedure can be represented in different ways. PDMs introduced in Chapter 4 can be classified into two groups:

- *State-based Procedure:* Every state represents one possible overall state of the whole procedure. Only one state can be active at a time. Finite State Automata and Statecharts are typical state-based PDMs.
- *Sequence-based Procedure:* The progress of these procedures is represented by a *marking* (see Section 4.3), i.e. the distribution of *State Marks* (or tokens, activity marks) in the procedure. States (or steps, places) with a mark are active. A procedure may have more than one mark. The distribution of marks can suitably describe the progression of concurrent sequences within a procedure. Petri nets, Grafcet, Sequential Function Chart (SFC) and Procedure Function Chart (PFC) are sequence-based.

Multiple state marks (or token, activity mark) may cause practical problems. Firstly, an uncontrolled distribution or multiplication of state marks may cause implausible structures (e.g. “unreachable sequence” and “unsafe sequence” introduced in Section 4.4). Complex checking algorithms should be carefully defined. Secondly, depending on different evaluation rules of individual implementations, different execution results for the same procedure could be a possible cause. Examples of unexpected execution results due to read/write conflicts can be found in [88, 89, 91]. The discussion on concurrency will be continued in Section 5.7.

SSC is designed as a state-based description method. In every SSC, exactly one state can be active. To describe sequential procedures or complex state machines, nesting structure can be applied (compare hierarchy in Section 5.4 and concurrency in Section 5.7). The progress of the whole procedure is represented through all active

states on every hierarchical level, whereas every single SSC should have only one *state mark* and one active state.

The state-based design ensures, that SSC can appropriately describe both, state-based procedures and sequence-based procedures. Execution risks during the multiplication of marks can be effectively avoided. Complex mechanisms for the control of mark multiplication are not needed, while the SSC model can be kept compact.

In similarity to SFCs and Statecharts, every SSC should have exactly one *initial state*. Statecharts can have any number of *final states*. Although the IEC 61131-3 standard has not explicitly defined the end of SFCs, most commercial SFC implementations allow multiple final steps. According to these two description methods, every SSC can have more than one *final state* which has no outgoing transitions. The example SSC in Figure 5.1 has exactly one final state and one initial state which is marked with a double-lined border.

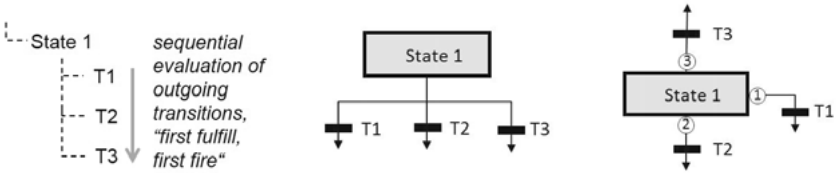
SSCs can be *terminable* or *non-terminable*. An SSC is terminable, when it has at least one final state. Terminable SSCs are typically applied to describe sequential procedures, e.g. a production procedure. A non-terminable SSC has no final states and can be used to describe state machines that always keep active. Every SSC frame has a standard output variable $terminated$ which shows the progress of the execution. $terminated$ of a non-terminable SSC will always be $FALSE$.

5.3 Transition

In general, a transition connects two states and possesses a transition condition. In the most PDMs regarded in Chapter 4, transitions are normally logical operations with no complex structure. The single exception is Statecharts, in which a transition is typically composed of guards, triggers and conditions. This design was adopted from computer science. Although more complex behaviors can be realized, it is less necessary in process automation.

The SSC utilizes the transition concept of the well-established SFC. All prerequisites for the state change (i.e. firing) are to be defined in a transition condition. Every transition has a Boolean variable $RESULT$. All prerequisites can be combined via logical operations such as OR and AND. A transition is *fireable*, if its $RESULT$ is set to $TRUE$.

Every transition should have a transition condition. The condition can be a constant (e.g. the $TRUE$ transition after the initial state in Figure 5.1) or a variable of the covering SSC frame (e.g. $IN1$ for $T2$ in Figure 5.1). Transition conditions can also be realized as function blocks (e.g. the composed function block connected with $T1$ in Figure 5.1). Simple conditions can be realized by atomic function blocks (e.g. "AND" or "$>$"). Complex condition can be defined as Composed Function Blocks. Function block conditions will be termed *FB-condition*. Every FB-condition should possess a Boolean



(a) Unified model for the arrangement and evaluation of transitions in runtime systems

(b) Possible representation for the rule: from left to right

(c) Possible Representation for explicit Priority

Figure 5.2: Prioritization of SSC Transitions (Priority in all there subfigures: $T1 > T2 > T3$

output, which is to be connected to the $RESULT$ of the corresponding transition. The same transition condition can be connected to different transitions.

Similar to the most PDMs, two SSC states can be connected to each other via an intermediate transition. Additionally, two SSC states can be connected via an *implicit transition* which is represented as a directed line. This design is borrowed from the Procedure Function Chart (PFC) and can be applied as a simplified graphical representation for a permanent $TRUE$ transition (e.g. the outgoing transition of the initial state in Figure 5.1).

5.4 Alternative Sequence

Alternative sequences are supported by all regarded Procedure Description Methods (PDM). The prioritization rule of alternative transitions is an important implementation detail, but disregarded in many PDMs (e.g. Finite State Automaton, Statecharts and Petri Nets).

In general, only one of the alternative outgoing transitions of the same state can fire. Different prioritization rules applied in existing PDMs and well-established automation tools are summarized as follows:

- Priority from left to right (as in SFC and PFC). The first fulfilled transition will be fired.
- Priority is to be explicitly defined by the user (as in SFC, MATLAB/Stateflow, PLC-Statecharts)
- It is a task of the user to ensure that all the transition conditions are mutually exclusive. (as in SFC and GRAFCET).

In the author's opinion, the PDM implementation should be designed in such a way that it can prevent unexpected execution results. In case the conditions of alternative transitions are not mutually exclusive although they should be, it is not meaningful to fire all fulfilled alternative transitions. Thus, the third rule is not supported in the SSC model.

The first prioritization rule is suitable for describing sequential procedures, which is typically processed from top to bottom. To ensure the intuition of the graphical visualization of procedures, the priority of conflicting transitions does not need to be explicitly shown. However, it is suited for state-based procedures (e.g. complex state machines), in which states are not arranged in a vertical column. For instance, to realize the state machine shown in Figure 4.2, transitions should be allowed to leave their source steps not only from the bottom but from all sides. In this case, the second rule is better suited.

The first and the second rule can be regarded as two engineering possibilities that are allowed in the graphical editor. A unified execution model for the implementation in runtime systems which can fulfill both rules can be defined. Figure 5.2a shows a reference model for the implementation of SSC in runtime systems. All outgoing transitions of a state are arranged under the step. In case a state is active, its outgoing transitions will be sequentially evaluated. The first fulfilled transition will be fired. Further transitions will not be evaluated.

This unified execution model realizes unambiguous semantics of the SSC model. Graphical visualization and engineering rules for graphical editors for SSCs are not stimulated. Figure 5.2b show a possible visualization for realizing the first prioritization rule. Figure 5.2c depicts a possible graphical visualization for the second prioritization rule. This design is borrowed from the widespread tool MATLAB/Stateflow and suitable for state-based procedures (e.g. a state machine).

The execution priority is not defined as an attribute of SSC transitions. Instead, it is represented by the arrangement of the transitions in the execution model. The question of which prioritization rule is to be applied will be defined in the graphical editor for SSC engineering. According to the chosen rule, transitions can be automatically arranged under their source steps.

5.5 Action

SSC states can be action-less and represent empty states. They should also be able to invoke actions, so that the SSC and the FB-agent can keep reactive to changes in the environment.

Action Types

Many Procedure Description Methods (PDM) only support value assignment or simple logical operations as actions. Some PDMs (e.g. SFC, Grafcet) support complex actions, for instance, for mathematical calculations and for the invocation of sub-sequences. As a general approach, SSC is designed to support different action types. An SSC action can be defined in one of the following forms:

- Value assignment to a variable of the execution frame (e.g. "OUT=0;" in Figure 5.1).
- Value assignment to one input of an function block within the execution frame, e.g. "./FB.IN1=30". The notation "./" indicates, that the target function block is directly encapsulated in the execution frame of the SSC. The function block name and the variable name are separated by a point.
- Call a local Function Block. For instance, the $CALL\ FBD1$ in Figure 5.1 invokes function block diagram $FBD1$ as an action.

The first form is supported by most PDMs analyzed in Chapter 4. The second one is an extension of the first form. The third form is borrowed from some commercial PLCs in which complex actions (e.g. SFC-action) are supported. All function blocks that are to be called as actions are termed *FB-actions*. This approach realizes a flexible definition of actions. FB-actions may be atomic (e.g. multiplication block) or composed (e.g. function block diagram).

Every state may possess any number of actions. FB-actions may be shared by different states in the same SSC. In other words, different SSC states can call the same FB-action. This design is inherited from SFC and can lower the engineering workload for reusable actions. To keep SSCs modular and encapsulated, FB-actions are not allowed to be shared by different SSCs. For instance, a sub-SSC cannot invoke FB-actions of the main SSC.

Action Control Condition

The PDMs compared in Chapter 4 have different solutions for action controls, ranging from simple and nondistinctive action control (as in FSAs and some variants of PNs) to complex 11 types for different execution behaviors (as in IEC61131/SFCs). Nondistinctive action control limits the functionality of the PDM, whereas too many variants of action control increase the complexity and reduce intuition and usability. The 11 different action control variants in SFC for example are helpful for programmers, but appear to be superfluous and not intuitive enough for users with no in-depth knowledge of programming or software engineering.

Similar to Statecharts, the execution of a SSC state is divided into three phases: *entry*, *do* and *exit*. As shown in Figure 5.1, every action should be assigned to an

execution phase. For the practical use, entry-actions are the most applied approach. Do-actions are typically applied to invoke FB-actions (e.g. subordinated SSC) cyclically. Exit-actions are typically used to reset variables and reset subordinated SSCs.

Further control conditions (e.g. saved and timed behaviors) will not be defined as an inherent part of the SSC model. The related time behaviors can be realized as function blocks. For instance, in case an action should set the variable $VAR1$ to $TRUE$ with a delay of 5 sec. after the activation of a state, a function block $DELAY$ can be defined as a do-action. Its Boolean output can be connected with $VAR1$. The function block will be cyclically executed. It checks the time duration in every iteration and sets $VAR1$, in case the delay time of 5 sec. is reached.

Action Execution

Actions of the same state will be sequentially executed. In case an FB-action is called, it will be executed for one iteration. This means that its inputs will be updated, the internal logic will be iterated once and the outputs will be updated at the end. The execution is theoretically timeless.

Actions with duration (e.g. “execution of a sub-procedure from beginning to end”) will be termed *activities* which are not directly supported by the SSC model. An activity can be defined outside of the execution frame of the SSC which should invoke the activity. The SSC can activate the activity (e.g. via signal assignment). During the execution of the activity, the SSC is not blocked. At the end of the activity, a confirmation signal can be sent back to the SSC. Section 5.6 will show an example of the execution of an external procedure which has unpredictable duration.

The second possibility of realizing activities is to approximate their continuous execution with a cyclical execution. An activity can be defined as a function block and invoked as a normal FB-action. For instance, a sub-SSC can be encapsulated as a function block (see also Section 5.6) which can be cyclically invoked as an FB-action. In case the sub-procedure reaches its final state, the activity is finished.

Action in Transition

In most PDMs, actions can only be invoked in states (or steps, places). Only Moore machines and Statecharts allow also actions in transitions.

To keep the model easy and simple, SSC transitions are not allowed to invoke actions. In case actions should be performed together with a transition, they are to be defined as exit-actions of the source state or entry-actions of the target state.

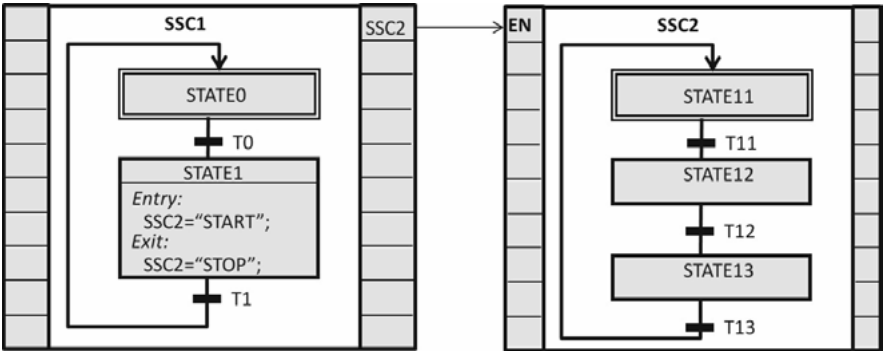


Figure 5.3: Signal-oriented control of subordinated SSCs

5.6 Hierarchy

Hierarchy is an important structure that is supported by most Procedure Description Methods (PDM) analyzed in Chapter 4. As introduced in Section 4.1 and Section 4.2, complex procedures without hierarchical abstraction quickly become unintuitive and confusing. A PDM without hierarchy (e.g. Finate State Charts) will be eliminated and replaced by other PDMs (e.g. Statecharts and Petri Nets). As a result, hierarchical structures are supported by Sequential State Charts (SSC).

Hierarchical procedures can be realized by simple procedures (i.e. procedures without hierarchy). As shown in Figure 5.3, the substructure of $STATE1$ in $SSC1$ can be realized as a standalone SSC with the identity $SSC2$. The activity of the sub-SSC can be controlled via signals. $SSC1$ possesses an output $SSC2$ which is connected with the activity control EN of $SSC2$. The assignment of $SSC2$ can be realized via "variable assignment" actions in $STATE1$.

This design is suited to realize simple hierarchical procedures. However, an SSC quickly becomes unintuitive and confusing, in case many states have sub-procedures. To close this gap, Sub-SSCs can be realized as FB-actions. As shown in Figure 5.1, SSCs on the lower hierarchy level can be defined as FB-actions of the main-SSC. Sub-SSCs can also encapsulate further sub-SSCs. This design is not novel but inherited from the *SFC-block* of SFCs and the *Macro Action* of Grafcets.

A prioritization rule for hierarchical SSCs should be defined. Figure 5.4 shows a general procedure with a nested sub-procedure. The markers show the active states in the both procedures. It is to be defined which transition can fire, when both $T1$ and $T11$ are fulfilled. In general, the following three general prioritization rules can be applied for the implementation in runtime systems:

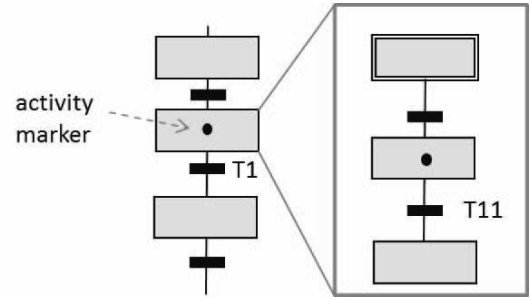
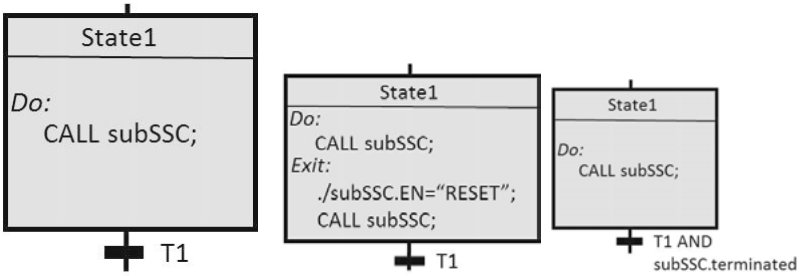


Figure 5.4: Prioritization issue of hierarchical procedures: which transition may fire, when $T1$ and $T11$ are both fulfilled?

- The main procedure has higher priority. This means that $T1$ may fire, $T11$ does not. The current state of the main-procedure will be left and deactivated. The sub-procedure will be interrupted and deactivated. It holds its current state and continues with the execution, in case it is activated again.
- The main procedure has higher priority. In contrast to the first prioritization rule, the sub-procedure should be reset after an interruption and start from its initial state when it is activated again.
- The sub-procedure has higher priority. Its execution cannot be interrupted. This means that $T1$ in Figure 5.3 may fire, only if the sub-procedure is terminated, which means that it has reached its final state.

This semantic issue is solved differently in the existing procedure description methods: Procedure Function Charts and Grafcets allow a unified rule; Statecharts allow various possible rules; other methods (e.g. Sequential Function Chart) have not defined prioritization rules explicitly. Similar to the prioritization of conflicting transitions discussed in Section 5.4, all these rules for hierarchical structure are important for the practical use. None of them can be abandoned.

To keep the semantics clear and easily understandable, SSC applies the first prioritization rule as the unified approach. Runtime behaviors of the other two rules can be realized via exit-actions. As shown in Figure 5.5a, a subordinate SSC is invoked as an action in $state1$. In case the transition $T1$ fires, the execution of $subSSC$ will be interrupted, but not be reset. In case the sub-SSC is to be reset (see Figure 5.5b), $state1$ generates a $RESET$ command and calls the $subSSC$ for the last time. To realize the third behavior, as shown in Figure 5.5c, the Boolean variable $terminated$ of the sub-SSC is to be checked in the outgoing transition. $State1$ can only be left, in case its subordinate SSC reaches its final step.



(a) Unified semantics: subSSC can be interrupted by T1.

(b) The subSSC can be interrupted by T1. subSSC must be reset and starts from its initial step when it is activated again.

(c) The main SSC must wait until subSSC is terminated.

Figure 5.5: Prioritization of SSCs on different hierarchy levels

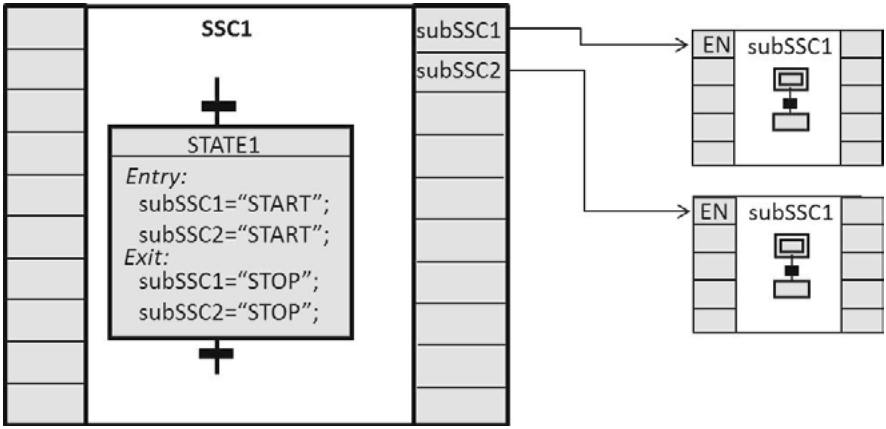
5.7 Concurrency

Concurrency is supported by nearly all existing Procedure Description Methods (PDM). The single exception is the early approach Finite State Automaton. As analyzed in Section 4.3, 4.4 and 5.2, the main execution risk of concurrency is the uncontrollable multiplication of tokens which control the activity of state. To ensure a safe and deterministic execution, concurrent behaviors of Sequential State Charts (SSC) should be meticulously designed.

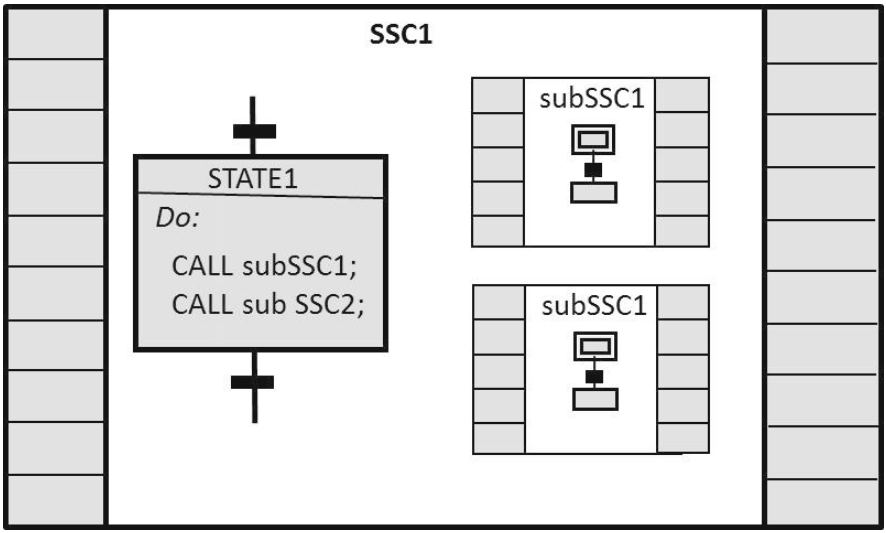
Theoretically, active states in concurrent or simultaneous sequences should be executed at the same time. However, a real concurrency can only be expected in a multi-core/multi-threading environment, which is not available in the most automation systems (e.g. PLC). Although concurrent sequences are allowed in the graphical editor and on the graphical visualization, concurrent states are executed sequentially, in fact, according to an implicit and system-specific order. As discussed in [88], different prioritization rules of concurrent sequences may lead to different execution results, and thus may complicate the formal analysis of procedures.

It is defined as a concept decision, that the SSC model does not support concurrent sequences directly. Every transition should have exactly one target state. Concurrent sequences should be encapsulated as separate SSC (cf. Figure 5.6).

Similar to hierarchical SSCs (cf. Figure 5.3), concurrent SSCs can be controlled by the main-SSC via signal connections. As shown in Figure 5.6a, the activity of the two sub-SSCs is set and reset in the main-SSC.



(a) Concurrent sub-SSCs can be controlled by the main-SSC via signal connections



(b) Concurrent sub-SSCs can be encapsulated in the main-SSC and invoked as actions in the same state

Figure 5.6: Concurrent sequences in SSC

Alternatively, sub-SSCs can also be encapsulated in the execution frame of their main-SSC and be invoked as actions. The execution priority of SSC-actions is represented by the position of CALL-actions in the state. As shown in Figure 5.6b, two concurrent SSCs are invoked in a state of the main-SSC.

In contrast to the existing procedure description methods, the prioritization of concurrent sequences is clearly and explicitly defined. In the signal-based solution shown in Figure 5.6a, the two sub-SSCs are encapsulated as function blocks. Similar to traditional function blocks, their execution order in the runtime system is explicitly defined. In the call-based solution shown in Figure 5.6b, actions of $STATE1$ are sequentially executed according to the rule “from top to bottom”. Thus, $subSSC1$ is executed prior to $subSSC2$ in every cycle.

5.8 Procedure Progress

The progress of procedures should be approximated by the cyclic execution in runtime systems. In an ideal case, a procedure progresses according to the *Maximum Progress* rule, which means that the procedure progresses as far as possible in a cycle until it reaches a stable state. However, as discussed in Section 4.4.2 and Section 4.5, Maximum Progress has the disadvantage that the system workload can vary greatly and is difficult to predict. Additionally, it is hard to avoid endless loops, if jumps are allowed in a procedure. As a result, SSC borrows the *Lock-Step* rule from Sequential Function Charts. Every SSC state lasts at least one cycle. In case a state is activated in one cycle, it cannot be deactivated in the same cycle, regardless of whether its outgoing transitions are fireable.

As introduced in Section 5.1, a deterministic execution within the execution frame of SSCs is required. The execution of the execution frame and its underlying components is to be controlled by an internal task list *intask*, which performs the following measures that are sequentially executed in every iteration:

1. Switch incoming signal connections and update signal inputs.
2. Execute internal logics:
   a) Switch *initial connections*, which are connections starting from an signal input (compare Figure 3.3).
   b) *Evaluate the SSC*: As only one state can be active and transitions are arranged under their source state, only the current state will be executed in this stage.
      i. In case the current state is executed for the first time (e.g. initial state), its entry-actions will be sequentially executed.
      ii. Determine outgoing transitions of the active state and evaluate their transition conditions according to a predefined prioritization rule (compare Section 5.3)

iii. In case no transition fires, do-actions of the current state will be sequentially executed. Otherwise, the current state will be deactivated; its Exit-actions will be executed.

iv. In case a transition fires, the succeeding state of the fired transition will be activated. Its Entry-action will be executed. Its Do-action will be executed once. All actions will be sequentially executed.

c) *Post-processing*: Execute continuous functions (compare, Section 5.1)

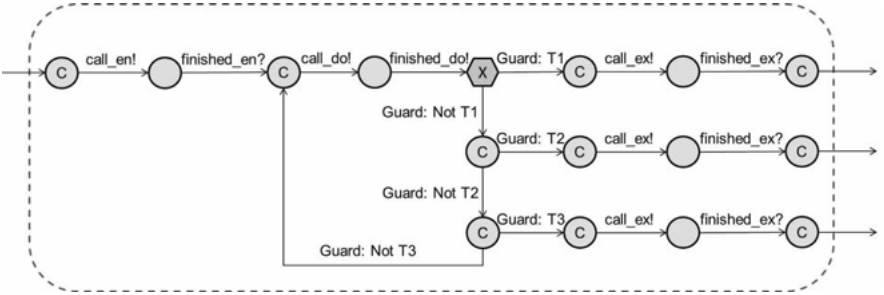
3. Switch *final connections*, which are connections ending with a signal output (compare Figure 3.3).

The model for progressing SSCs is shown in the form of UPPAAL automaton in Figure 5.7b. $SSC\ x$ will be activated, in case it receives a $Call$ signal. All states possess the same UPPAAL model in Figure 5.7a. Arcs on the left and right side of the model represent the connections between different states.

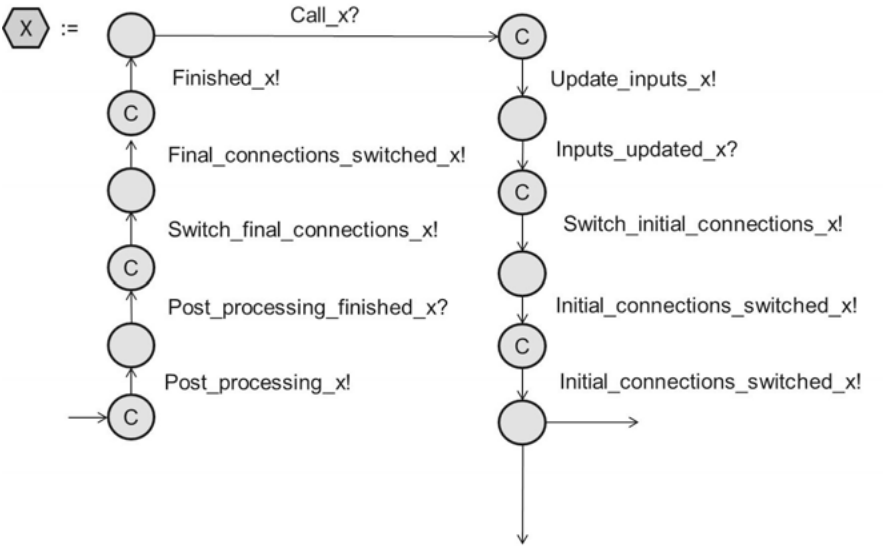
The execution model for progressing SSCs combines advantages of the *immediate action and immediate transit evaluation (IA-IT model)* and *deferred transition evaluation and deferred action (DT-DA model)* introduced in Section 4.4.2. All SSC states follow the execution order: entry-actions, do-actions, transitions and exit-actions. The end of an execution cycle takes place between the execution of do-actions and the execution of transitions. With this design, actions of the initial state can be correctly executed and will not be skipped in the first cycle. Transitions are evaluated prior to the actions in further cycles. “Transitions prior to actions” allows SSCs to respond to inputs as quickly as possible and can avoid unexpected results which can be caused for instance by read-write-conflict in a “actions prior to transitions” model.

Additionally, the execution model allows a fast progress of SSCs. Every state lasts at least one execution cycle and last exactly one cycle when its outgoing transitions are fulfilled. No execution cycle will be lost during a state change. In case a state is deactivated in one cycle, its exit-actions can be immediately executed. The entry- and do-actions can also be executed in the same cycle.

Furthermore, the representation of the procedure progress at the end of every cycle is correct and unambiguous. In the idle time between two cycles (compare Figure 2.5), an SSC is always in the progressing state “do-actions of the active state are executed”. This clear and defined execution state is not ensured in many existing procedure description methods. For example, PLC-statecharts [74] and Sequential Function Charts in the commercial tool CoDeSys end the cycle after exit-actions. In case a state (or SFC step) is left, it is still marked as the “current state” in the idle time, although it should actually be inactive. Its next state, which is the real current state, is still not started. Due to this confusing state representation in idle time, some literature sources differentiate the two terms “current state” and “ready state”. However, the state representation of SSCs is always clearly and correctly defined. There are no SSC states, which are “inactive but still running” or “active but still not started”. The correct state representation is



(a) Runtime Model of an SSC state (Hexagon represents the macro step in Figure 5.7b)



(b) Macro step: runtime Model of the Execution Frame for SSC x

Figure 5.7: UPPAAL Automation for SSC

especially meaningful for SSCs, since SSCs and FB-agents could be executed in a non-cyclic execution context (compare Section 3.8).

5.9 Summary

Sequential State Charts (SSC) are mainly based on Seqential State Charts (SFC) and Statecharts (SC) and clarify many design details which are ambiguous in the both pre-

decessors. The main differences and the relationships among SSC, SFC and Statechart can be summarized as follows:

> SSC $\approx$ SFC $-$ simultaneous branching $+$ simplified action control $+$ encapsulated execution frame $+$ unambiguous semantics

and

> SSC $\approx$ Statechart $-$ event $-$ action in transition $-$ complex hierarchy $+$ implementation mechanism for actions $+$ encapsulated execution frame $+$ unambiguous semantics

Sequential State Charts are suited to describe sequence-based procedures and state-based procedures in the FB-agent model introduced in Chapter 3. SSCs provide complete and unified solutions for different design details (e.g. hierarchy, prioritization rule). The SSC semantics are clearly defined. The main features and design decisions for SSC can be summarized as follows:

- All procedure elements are encapsulated in an SSC execution frame which is characterized by its signal interfaces, enclosed name space, deterministic execution and white-box construction.
- Only one state can be active within an SSC. Multiple tokens (or activity marks) are not allowed. Hierarchical procedures and concurrent sequences are to be defined as individual SSCs.
- Every SSC has one initial state and any number of final states.
- Unified semantics are defined for action execution, transition prioritization, hierarchy and concurrency. Mathematical models according to UPAAL have been defined which can support a formal analysis of SSC implementations.
- Actions are timeless. Activities (i.e. action with duration) should be approximated by cyclic execution of actions or external function blocks.
- To ensure a quick response, transitions are executed prior to states and actions in every cycle.
- In case a state is left, its succeeding state will be activated in the same cycle. The activity of all states is correctly set or reset. The state representation of the whole SSC in the idle time between two cycles is clearly defined.

The general procedure description method SSC combines advantages of existing PDMs. All concept decisions on syntax and semantics are made on the basis of synthesis of existing PDMs. No novel procedure element has been developed. To gain a complete and meaningful execution model, many design details (e.g. prioritization rules) have been clearly specified. This model can also be regarded as a reference model which serves as a guideline for the implementation of different procedures in existing runtime systems.

The concept of SSC was published in a previous publication [19] of the author. Details about the execution model and a prototype were published as an internal Technological Paper of the Chair of Process Control Engineering in Aachen in 2013. On the basis of these works, a reference model for procedure description in the context of Industry 4.0 was published in a further publication of the Chair [96]. In contrast to the SSC introduced in the present work, this reference model is tailored for the Industry 4.0 context and involves less design details for the implementation in runtime systems.

6 Prototypical Implementation

This chapter presents a prototypical implementation of the FB-agent model (Chapter 3) and the Sequential State Chart (SSC, Chapter 5). In order to keep the implementation vendor- and platform-neutral, the development environment provided by ACPLT-technologies is applied during the modelling and implementation phases. After an introduction of ACPLT technologies in Section 6.1, a detailed design of the FB-agent and SSC will be introduced in Section 6.2 and Section 6.3 respectively.

6.1 ACPLT Technologies

ACPLT[1] Technology is the umbrella term for reference models and software implementations developed at the Chair of Process Control Engineering at the RWTH Aachen University in Germany. ACPLT technology targets application areas within the field of process control engineering and is designed and implemented in a vendor- and platform-independent manner. Models, concepts and implementations of ACPLT technologies have already been applied in different industrial projects and standardizations.

6.1.1 Object Management System: ACPLT/OV

ACPLT/OV[1][97] is an object-management system. As shown in Figure 6.1, OV includes the following components: a language for defining object-oriented class models (OVM), an object-oriented application programming interface (API) for ANSI C (libov), and a platform-independent runtime system (OV server). An OV server can be regarded as a standalone component (cf. Section 2.2.3) which is responsible for memory management, task control, object management etc. The server supports dynamic loadable class libraries, meta-objects and meta-classes, as well as persistent storage.

In comparison with classic automation runtime systems (e.g. the runtime according to the IEC61131-3), ACPLT/OV provides several novel and distinctive features:

Complete meta-model available on the runtime system: In traditional automation runtime systems according to the IEC 61131-3, the machine code running on the target

[1] AaChener ProcessLeitTechnik: The German expression of Aachen Process Control Engineering
[1] Objekt Verwaltung: German expression for object management.

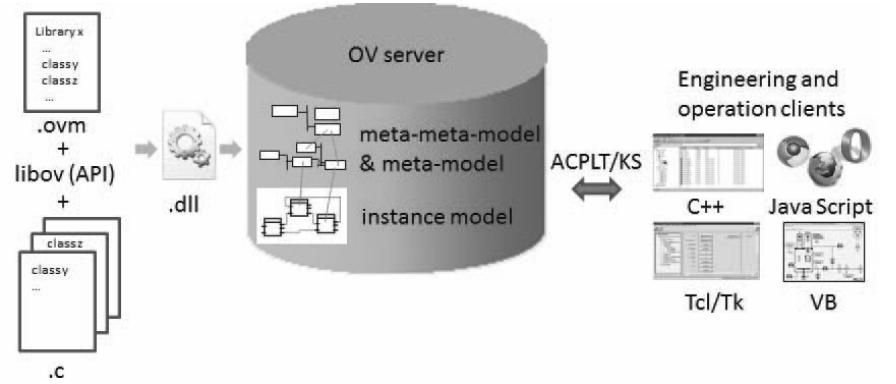


Figure 6.1: ACPLT/OV

hardware (e.g. a PLC) is not *self-descriptive*. This means that meta-information about the loaded instances is not available on the runtime system. Information about instance type, behavior and connections among instances cannot be explored from the outside. In ACPLT/OV, instances, instance models and their meta-models are all present in the form of *objects* on the runtime system. The dissection of objects into the three types is a semantic grouping. However, viewed from the OV system, it makes no difference with regard to the management of those objects. Instance models, meta-models and meta-meta models are treated equally in OV. This design simplifies the maintenance tasks and engineering tasks. The development of adaptive systems and self-X technologies can also be well supported.

Integrated Engineering and Execution: Classical automation runtime systems follow the engineering cycle: edit-compile-load-execute. Control algorithms are engineered in graphical editors of an engineering system (typically a standalone PC-station) and can be complied into machine codes and loaded onto the execution hardware (e.g. the automation controller in Figure 2.1). In case the code is to be modified, the execution of the runtime system should be interrupted. In OV servers, control algorithms are represented as objects. Instead of compiling and loading machine codes into the runtime system, users implement the control algorithms by instantiating and parameterizing objects. Function blocks, signal connections, states, transitions and actions can be directly defined in the form of objects in the runtime system. One effect of using ACPLT/OV is that there is no distinction between the engineering system and the execution system of the control algorithms. Objects in the runtime system can be flexibly modified, even during the running operation. The access to OV servers can be controlled. Only authorized users or systems can explore and manipulate objects on an OV server.

Online model exploration and manipulation via ACPLT/KS ACPLT/OV supports the communication protocol ACPLT/KS [98] which allows - similar to OPC technol-

ogy [30] - a platform-neutral communication among different systems. Data and meta-information of objects in an OV server can be explored and edited through local or remote KS clients. This design ensures great flexibility in performing engineering, operation and monitoring tasks.

Online reconfiguration via model transformation: One central advantage of having instances and their meta-models on the runtime system is the possibility of performing automatic reconfiguration. In order to adapt an existing control algorithm to a new situation (e.g. a new version, or a certain execution situation), its initial situation and the target situation can be clearly described via a model of objects. Model transformation rules can be formally defined. and performed by active objects (e.g. engineering agent). The transformation can be carried out offline or online.

The development of ACPLT/OV and ACPLT/KS is supported and accompanied by different research works and publications at the Chair of Process Control Engineering. So far, ACPLT/KS clients have been developed in C++, Tcl, JavaScript and VB. The usability and stability of OV and KS have been proved in different academical and industrial projects.

6.1.2 Basic Libraries

Related libraries for FB-agent and Sequential State Chart (SSC) are shown in Figure 6.2.

The fundament of the OV environment is the OV library which defines basic classes (e.g. object, domain and variable) and basic associations (e.g. inheritance, containment). OV doesnt support multiple inheritances. This is indirectly realized through the association $embedment$. To combine attributes and behaviors of different classes, one $object$ may embed objects of different classes. One example is depicted in Figure 6.4.

The FB library specifies meta-models for function blocks, execution tasks and signal connections. According to the IEC 61131-3 [3], a function block can only be executed when they are assigned to a task. In the FB library, $functionBlock$ is directly derived from $task$. Every function block can then be executed as a task. $functionBlock$ is abstract, which means that it is only a derived class (e.g. ADD, AND, $valveControl$ and etc.)

$Task$ can be applied to realize the cyclic execution context introduced in Section 2.2.4. Tasks can be arranged under a main task (e.g. the root task in Figure 2.6) via the association $taskList$ and build a *task tree*. The association parent of $tasklist$ is termed *task parent*, whereas the association child is called *task child* or *subtask*.

Signal connections are realized as instantiable objects. Every connection instance is linked with the source function block and the target function block via two separate associations.

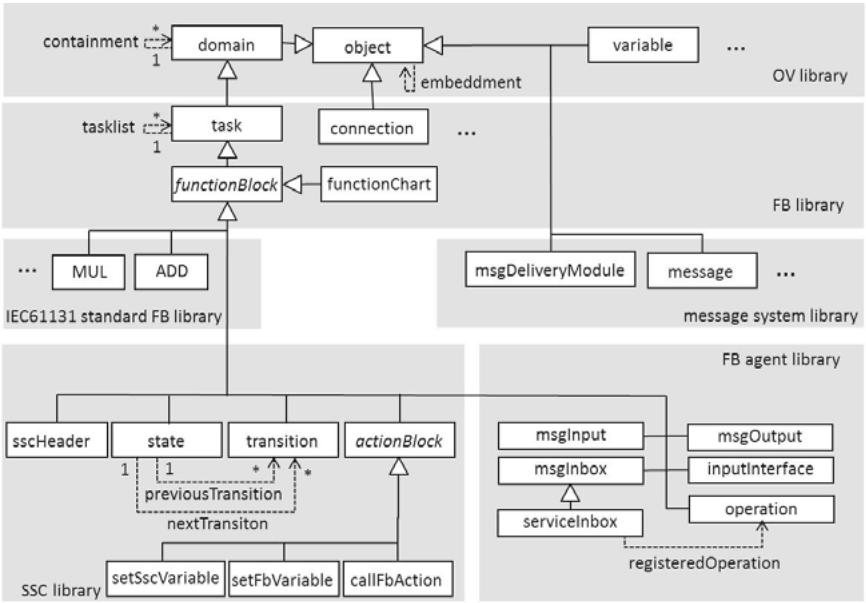


Figure 6.2: Class libraries in the ACPLT/OV environment (simplified)

$functionChart$ realizes the Composed Function Blocks (CFB) presented in Section 3.8. $functionChart$ is derived from $functionBlock$, and allows an aggregation of the internal algorithms with function blocks. Every $functionChart$ is encapsulated and possesses a local name space for underlying variables and function blocks. The execution of a $functionChart$ is dissected into three phases:

- *pre-processing* will be executed at the beginning of the execution cycle.
- *intask*
- *post-processing* will be executed at the end of the execution cycle.

Every function block within a $functionChart$ should be assigned to an execution phase. Most function blocks are to be sequentially executed in the *instask* phase.

Another fundamental library in this context is the KS library [98] which realizes data exchange between OV objects. Elementary communication functions are encapsulated in an Application Programming Interface (API) which can be utilized in further libraries. KS exceeds the scope of this thesis and is not shown in Figure 6.2.

The KS API is, for instance, applied in the Message System Library which realizes a message oriented communication. This library is generically defined. Message sender and receivers can be represented as any OV objects. This library is applied to realize

the agent-to-agent communication specified in Section 3.4. Messages are realized as instances of the class $message$. Every Standalone Component (cf. Figure 2.4) possesses a Messages Delivery Module (MDM). It delivers messages from the sender to the local or remote receivers. In case an agent is the receiver, the message object is linked via the association $containment$ under its message input (i.e. an instance of the class $msgInput$ in Figure 6.2).

The introduced libraries form a development basis. On this base, further libraries have been developed in different works for different application areas such as diagnostics [99], product transport [100], Human Machine Interface (HMI) [101] and others.

6.2 FB-agent Library

The FB-agent library implements the FB-agent model (cf. Chapter 3) in the ACPLT/OV environment. The class diagram of the FB-agent library is shown in Figure 6.2. Model elements and the construction of an FB-agent will be introduced in the following paragraphs.

Model Elements

As introduced in Section 3.5, an FB-agent is encapsulated in an Execution Frame which controls the execution of all internal components according to the model specified in Figure 3.9. In the FB-agent library, no specific class is defined for the execution frame. This can be realized as a standard Composed Function Block of the class $functionChart$.

All internal logics of agents should be function blocks. Service interfaces specified in Section 3.6 are defined as instantiable sub-classes derived from $functionBlock$: $msgInput$ for message inputs; $msgInbox$ for message inboxes; $msgOutput$ for message outputs; and $samplingInterface$ for input interfaces.

In contrast to a normal $functionChart$, the unique service interface and the service inboxes of an FB-agent must be executed at the beginning of every iteration, i.e. in the *post-processing* phase introduced in Section 6.1.2.

Every input interface $samplingInterface$ reads the current value of a variable in the network. It has an output $VALUE$ of the OV data type ANY which consists of a time stamp, the status of the type unsigned integer and a value of any data type. The data format of the last variable is void by default and can be dynamically determined. The state shows the quality of the value. In general, a value may have the quality: $NOT\_SUPPORTED$, $UNKNOWN$, BAD, $QUESTIONABLE$ or $GOOD$.

A message inbox $msgInbox$ processes messages of a certain category (e.g. invalid), though it may also process messages for a specific service.. The prioritization rule,

capacity and overflow measure (cf. Section 3.6.1) are defined as three parameters of the $msgInbox$ class. Different processing rules of incoming messages can be defined according to different parameterizations, e.g. "First-In-First-Out".

As introduced in Section 3.6, a service inbox acts as the representative object of a service. The identity of a $serviceInbox$ instance should be set to the service name. Every $serviceInbox$ has a STRING variable which holds the textual description of the service. The class $serviceInbox$ is derived from the $msgInbox$. In contrast to a common message inbox, a service inbox holds the description the service and manages all operations arranged under this service.

As introduced in Section 3.6, a function block or a set of function blocks within an FB-agent can be registered as an operation. Every registered operation has an exclusive representative instance of the $operation$. Every $operation$ should be linked via the association $registeredOperation$ to a covering service (i.e. a service inbox). The identity of an $operation$ should be set to the name of the operation it represents.

To read ad and write the XML-based messages (compare Section 3.3), an XML parser named *MiniXML* [102] was chosen in a student work. MiniXML is a small and free library developed based on ANSI C language. It allows a simple and quick processing of XML format without large non-standard libraries. In another Bachlor thesis, the message format introduced in Section 3.3 was specified for FB-agents and implemented in the OV environment with help of a MiniXML parser.

Instance Model

The class diagram in Figure 6.2 cannot suitably describe engineering rules and constrains of the FB-agent model. In order to provide a guideline for the construction of a concrete FB-agent model in runtime systems, a so-called *Instance Model* is defined in Figure 6.3.

The instance model is not described in a formal language. It applies UML-like notations which were developed in an internal research work at the Chair of Process Control Engineering. This approach was first applied for the description of a visualization model [103]. A theoretical research on a formal description method will be followed by an ongoing dissertation at the Chair of Process Control Engineering. Key elements of the instance model will be introduced in the following paragraph.

Every rectangle in the model represents an object instance. The instance identity and the class should be underlined and separated by a colon. In case the instance identity is kept blank, it can be freely defined. Otherwise, it is fixedly defined and should not be changed. Abstract classes (with italic text) can also be applied as placeholders of their sub-classes. For instance, the abstract class $functionBlock$ in Figure 6.3 indicates that an instance of any instantiable sub-class of $functionBlock$ (e.g. ADD) can be put on this place.

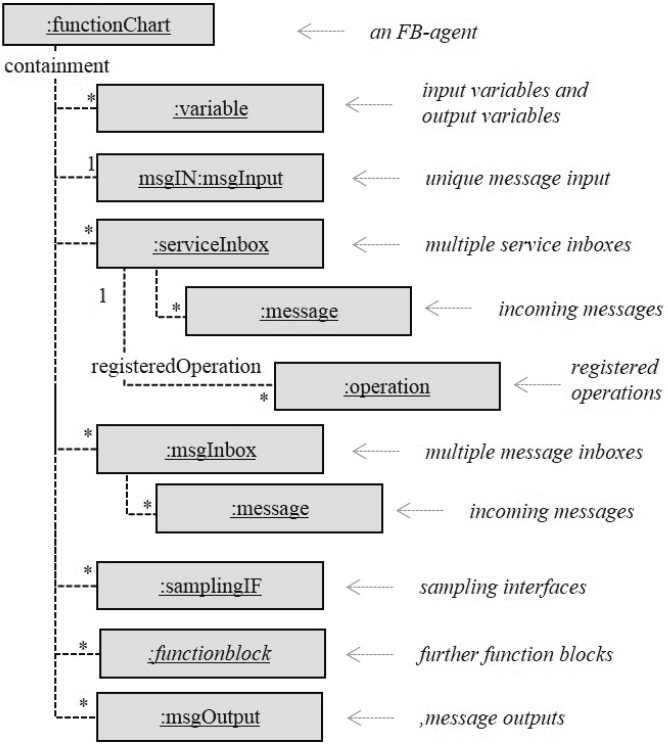


Figure 6.3: Standard Instance Model of an FB-agent

The arrangement of instances is represented as a tree structure. Objects are connected via associations (e.g. $containment$) to each other. The hierarchical level of instances is indicated by their indent. The number next to the association shows the allowed number of instances. A star $*$ indicates that any number of instances is allowed.

Figure 6.3 shows a typical FB-agent which is realized as an instance of $functionChart$. It can encapsulate multiple variables, function blocks and service interfaces. However, every agent can possess only one message input. The identity of the message input is standardly defined as $msgIN$. The identity of further objects can be flexibly defined.

The instance list of $serviceInbox$ represents the list of registered services within an agent. Every $serviceInbox$ can possess more than one registered operation.

In case a message is sent to the agent, a message instance will be created and arranged under the unique message input $msgIN$. After a formal checking (cf. Section 3.6), it will be forwarded immediately to specific message inboxes or service in-

boxes. During the forwarding, the message instance will not be deleted and recreated. Instead, it will only be “relinked” which means that its parent of the $containment$ association is redefined.

The elements in Figure 6.3 should be deterministically executed according to the order specified in Section 3.8.

6.3 SSC Library

The SSC library implements the Sequential State Chart (SSC) presented in Chapter 5.

6.3.1 Class Diagram

As shown in Figure 6.2, all SSC elements are directly or indirectly derived from the class $functionBlock$ of the FB library. This means that all SSC elements are encapsulated as function blocks. Their attributes can be declared as inputs and outputs that can be read or written via signal connections. Since $functionBlock$ is derived from $task$, every SSC element can be regarded as a sub-task and can be directly linked to a task tree. This design simplifies the development of the execution model of SSC.

Every SSC represents a procedural Composed Function Block (CFB) and is to be encapsulated as a $functionChart$. Every SSC contains an $sscHeader$ which is the central unit for the overall control of the procedure that is to be described. All steps and transitions of an SSC are to be managed by the $sscHeader$ of the SSC (cf. Section 6.3.2 for more detail).

$State$ and $transitions$ are defined as two classes. The initial state and final states are not defined as special classes. Every state has two standard Boolean variables $isInit$ and $isFinal$ which are set to $FALSE$ by default. $isInit$ of the initial state will be set to $TRUE$, whereas $isFinal$ of final states (i.e. states with no outgoing transition) should be $FALSE$. Every step is linked to its incoming transitions via the association $previousTransition$, and linked to its outgoing transitions via the association $succeedingTransition$.

Actions and transition conditions have no exclusive class. They are to be realized as ordinary function bocks (e.g. AND, OR, ADD, Function Block Diagrams etc.)

Action block is defined as an abstract class $actionBlock$ which has three instantiable subclasses: $setSscVariable$ for the assignment of a variable to the covering SSC; $setFbVariable$ assigns a variable to a function block within the SSC; $callFbAction$ invokes an FB-action (compare Section 5.5). An FB-action (i.e. function block action) can be invoked by different $callFbAction$ blocks.

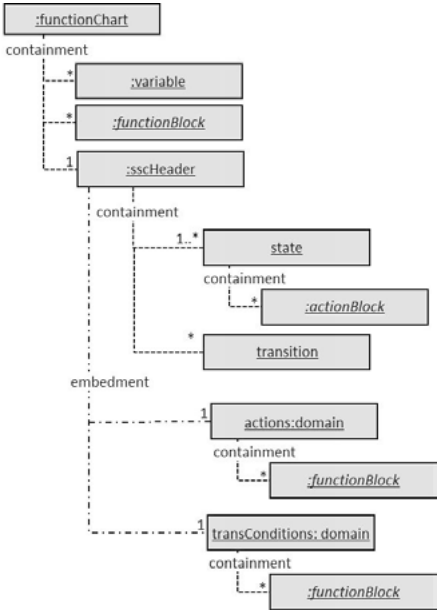


Figure 6.4: Instance Model of a Sequential State Chart (SSC)

6.3.2 Instance Model

Figure 6.4 shows the general instance model of SSC. Every SSC execution frame (i.e. $functionChart$) contains any number of variables and function blocks and precisely one $sscHeader$. In other words, every $functionchart$ can only represent one SSC. However, a function chart is allowed to contain further function charts whose internal logic is also an SSC. For instance, an SSC which is encapsulated as a function chart can be arranged under the doman “action” (cf. Figure 6.6) under an $sscHeader$.

States and transitions are arranged under the $sscHeader$. Every transition must have exactly one previous state and exactly one succeeding state. Every $sscHeader$ should contain at least one $state$ which represents the initial state. Every SSC state may contain any number of action blocks.

$functionBlock$ and $actionBlock$ are abstract and cannot be instantiated. Their place (object with italic text in Figure 6.4) is to be taken by instances of their subclasses. For instance, $actionBlock$ in the figure indicates that $setSscVariable$, $setFbVariable$ or $callFbAction$ can be instantiated here.

Every $sscHeader$ contains two standard domains (i.e. instance container), namely $actions$ and $transConditions$. These two domains realize the two special regions for FB-actions and transition conditions in Figure 5.1. Function blocks in $transConditions$

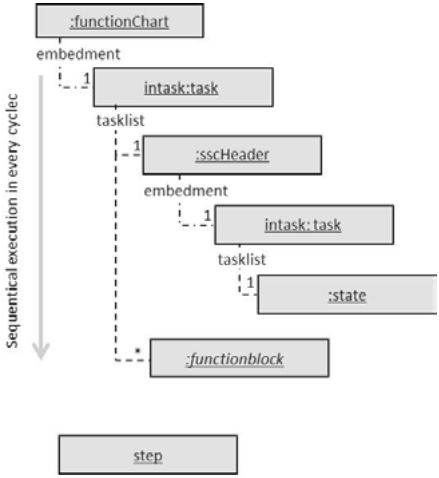


Figure 6.5: General Task Tree for an Sequential State Charts (SSC)

are to be regarded as transition conditions. Function blocks in the $actions$ region are FB-actions and should have no parent task. These two domains are linked via the association $embedment$ to the $sscHeader$. In contrast to $containment$, embedded objects are fixed. They can neither be renamed nor deleted.

6.3.3 Task Model

The execution order of SSC elements is not identical to their arrangement order. The instances in Figure 6.4 should be arranged via the $tasklist$ association according to the so-called *SSC task model* depicted in Figure 6.6.

SSC Exectuion

$functionChart$ and $sscHeader$ possess an embedded task named $intask$ which schedules the execution of child tasks of a function chart or SSC. This task is explicitly defined, so that the outputs and $functionCharts$ and $sscHeaders$ can be updated after the sequential execution of all child tasks (compare the execution order described in Section 5.8).

As shown in Figure 5.2a, transitions are to be arranged as child tasks under their source states. Thus, only states are directly linked to the $intask$ of an SSC. As introduced in Section 5.8, the state change should be performed in the same execution cycle. In case a state is left, the next state should be activated in the same cycle. The simplest solution is to arrange all steps as child tasks under $sscHeader$ according to a

sequential and fixed order. The disadvantage of this design is that it can lead to loss of one execution cycle during the SSC progress in runtime systems. For instance, in case an SSC jump from state $S5$ back to a previous state $S2$ and $S5$ is arranged after $S2$ in the task tree, $S5$ can only be activated in the second cycle.

To overcome this potential gab, SSC-states are dynamically linked to $sscHeader$. As shown in Figure 6.5, an $sscHeader$ should only have one $state$ instance as child task. Before an SSC starts to progress, its initial state is linked to the $sscHeader$ by default. Further states within the same SSC have no parent task. During the progress of an SSC, only the current active state will be linked to the $sscHeader$. When a transition fires, it unlinks the active state from the $sscHeader$ and links the next state to $sscHeader$.

A further advantage of the dynamical task list is that the computing resource can be saved. In case all states are statically arranged under the $intask$, their activity will be evaluated in every execution cycle, although only one of them can be active. In a dynamical task list, the $intask$ of $sscHeader$ has always only one child task and does not need to ask other deactivated states whether they should be executed.

$sscHeader$ checks in every execution iteration, whether it has a child task. If not, it finds the intial state (i.e. the state with $isInit = TRUE$) and links it via the association $tasklist$ to the $intask$ of $sscHeader$ automatically. This function ensures that always one state is active within an SSC. In case an SSC is initialized or reset, this function will also be performed.

Task Tree of a State

Figure 6.6 shows the task model of an SSC state. This model realizes the *immediate action and immediate transit evaluation* semantics (i.e. IA-IT model) discussed in Section 5.8.

Every step has four fixed subtasks: entry, transitions, do, exit. These tasks are integral and linked via the $embedment$ to their covering step. According to the assigned value of the attribute $qualifier$, action blocks are to be arranged under the entry- do- or exit-task.

All outgoing transitions of the step should be arranged under the subtask $transition$.

The tree structure shown in Figure 6.6 can be automatically generated according to the instance model of Figure 6.4. In the current SSC prototype in the ACPLT/OV environment, a set assessor is defined for the $qualifier$ of the class $actionBlock$. In case the $qualifier$ is set to $TRUE$, the action block will be automatically linked via a $tasklist$ association to the entry-action of the covering state. No engineering rules are to be developed in the engineering clients (compare Section 6.1.1).

The task tree is sequentially executed from top to bottom in every iteration. The entry-task and the do-task are active initially. In case the step is activated, these two tasks and their underling action blocks will be executed once. At the end of the first execution, the entry-task will be deactivated, the do-task will be kept active, the transition-task

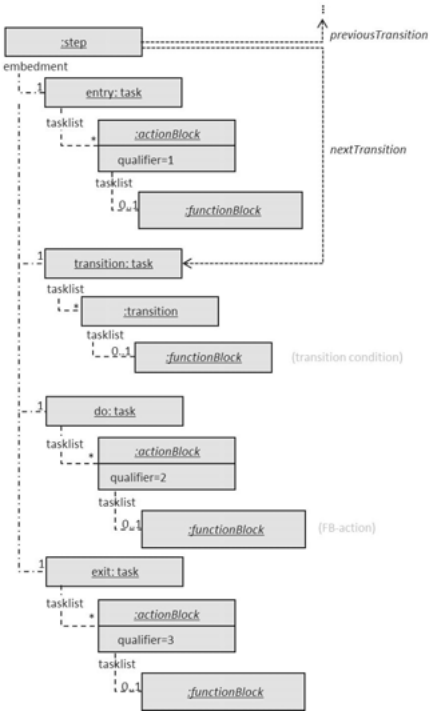


Figure 6.6: Standard Task Tree of an SSC State

will be activated. In the next iteration, transitions will be sequentially evaluated. In case no transition is firable, the do-task and its underlying actions will be executed. In case a transition fires, the do-task will not be executed and be deactivated immediately. Instead, the exit-task will be activated, executed once, and deactivated afterwards.

When a transition fires, it deactivates the do-task, activates the exit-task, deactivates its source state (i.e. the current state), activates its target state and links it to the $intask$ of the covering $sscHeader$ as the new child task (compare Figure 6.5).

The class $transition$ has no attribute specifying its priority (cf. Figure 5.2). All transitions are arranged under the transition-task and to be sequentially executed. The arrangement of transitions shall be defined according to the prioritization rule chosen in the engineering client for SSC (compare Secion 5.4).

7 Case Study

This chapter introduces the test results of FB-agents and Sequential State Charts (SSC) in a project where a multi-product and multi-function metallurgical furnace was automated. After an introduction of the automation concept in Section 7.1 and Section 7.2, solutions for process control and engineering will be introduced in Section 7.3 and Section 7.4.

7.1 Research Plant: Submerged Arc Furnace (SAF)

The FB-agent model (cf. Chapter 3) and the Sequential State Chart (SSC) (cf. Chapte 5) are tested during the automation of a new pilot furnace which was constructed at the Institute of Process Metallurgy and Metal Recycling (IME) of RWTH Aachen University. The new furnace with the patent name Modular Electric Reducing Furnace (SAF) is built and donated by the company SMS Group GmbH. The chair of process control engineering is currently in charge of the design and implementation of automation systems.

As shown in Figure 7.1, the SAF is equipped with one fixed mounted bottom electrode and three vertical movable electrodes that are positioned on top of the furnace vessel. Every of the top electrodes is carried by a hydraulic column. During a melting process, electric arcs are generated between the top electrodes and the melting material. The length of the electric arcs can be controlled through raising and lowering the top electrodes. The electrical energy and the melting temperature can be controlled correspondently. In contrast to a classic electric arc furnace, the SAF allows multiple melting modes: AC or DC power supply; with one, two or three top electrodes; with or without bottom electrode; with or without electric arcs.

The SAF in Aachen is applied for academic research, and is smaller than a industry-standard production furnace. It has a useful volume of $2m^3$, and can melt up to 10 tons of material in every charge. However, the complexity of functionality and the engineering cost (hardware and software) are even higher than they are for an industry-standard production furnace. As a research furnace, SAF applies more field devices as usual. For instance, in order to gain the complete temperature distribution in the vessel, more sensors are installed. In total, 50 actors, over 150 sensors, and about 2000 signals must be engineered. The SAF project places high demands on the automation engineering:

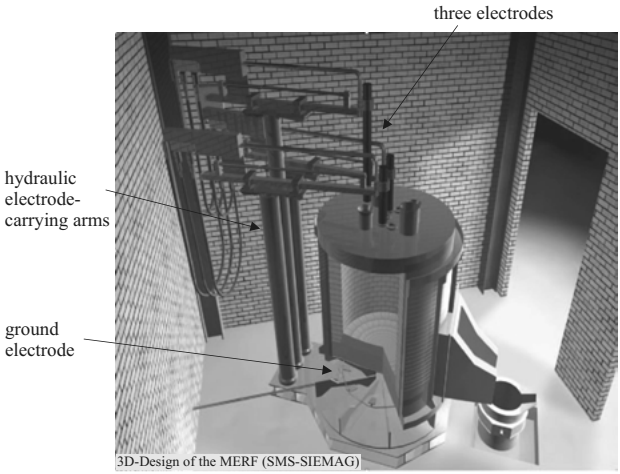


Figure 7.1: Research Plant: Submerged Arc Furnace (SAF)

Firstly, various (including some innovative) production processes will be tested or newly developed together with industrial partners. A flexible combination of operational resources (e.g. single devices, device groups) should be allowed.

Secondly, the process control strategy is also part of the research works on the SAF plant. To test a new method for measuring the vessel temperature, temperature sensors could be added or removed. To optimize the cooling system, the combination of pumps and control valves could be modified. The process control should be implemented in a manner that allows flexible manipulation and extension.

Furthermore, research works on the SAF are mostly driven by PhD students who typically work at the university over a time period of five years. Thus, the knowledge of the plant, processes and automation should be transferred from one generation to the next generation every five years. Due to frequent extensions of the plant and the associated processes, a smooth knowledge transfer should not only be supported by well-written documents, but also by intuitive implementations. Automation solutions should be easily understandable and manageable in order that new employees can easily master the implementations and engineer new functions.

7.2 Process Automation System

Figure 7.2 gives an overview of the applied automation system for the SAF. All field devices and hardware on the control level are catalog products from commercial vendors.

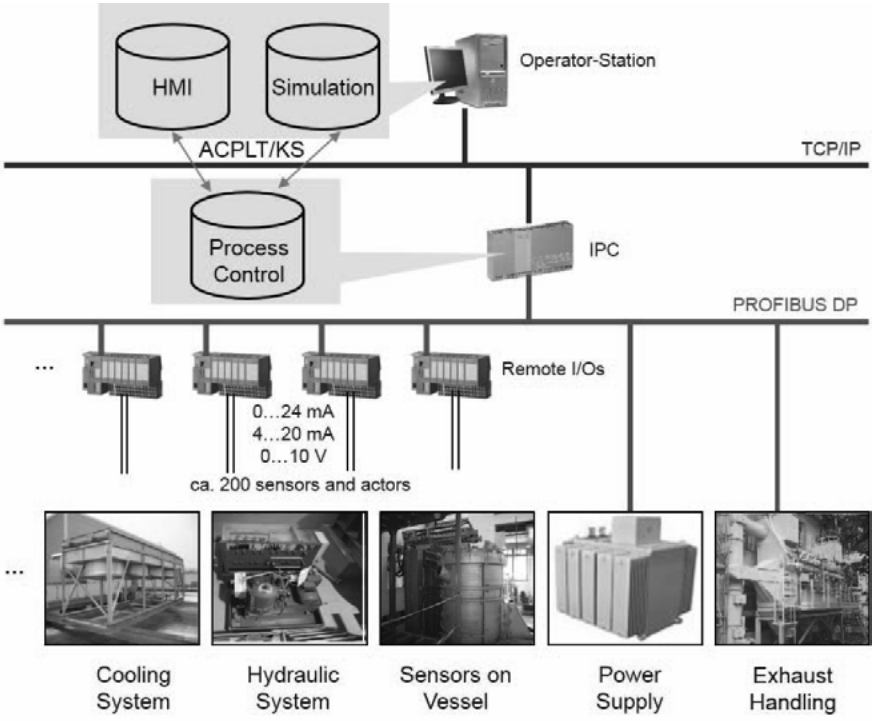


Figure 7.2: Automation System for the SAF at the IME Institute in Aachen

Automation software for the control level are implemented in the runtime environment provided by the ACPLT technology introduced in Secion 6.1.

The central device for process control is an Industrial PC (IPC) with a Windows-Embedded operating system. All peripheries like power supply for the electrodes, water cooling system, exhaust handling, chargers and manual control panels are connected with the IPC either directly via PROFIBUS or indirectly via remote I/Os. Furthermore, a desktop PC is connected with the IPC via Ethernet. It acts as the operation-station and is in charge of observation and manual control, archiving and protocoling. Functional Safety is not realized in the system shown in Figure 7.2. All safety relevant functions or Safety Integrity Levels (SILs) are realized by special hardware on the field level. A second IPC is in the planning stages to increase the availability of process control and to provide more computing capacity for future extensions of the process control strategy.

Process control logics are installed on an ACPLT/OV server on the IPC. A further OV server for Human Machine Interface is set up on the operation station. Additionally, a simulation server is set up on the IPC. On this server, simulation models for I/O commu-

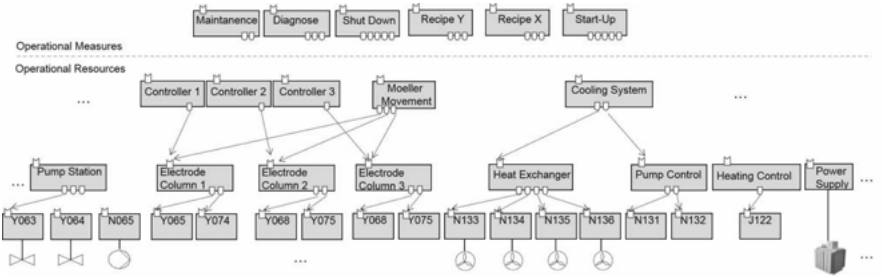


Figure 7.3: Process Control Agents for the SAF plant

nication, field devices and chosen melting processes are installed. They can be applied to test process control logics, before they are coupled to the physical plant.

Every OV server is a Standalone Component according to the unified runtime model introduced in Section 2.2.3. The communication between the OV servers is realized by ACPLT/KS (cf. Section 6.1).

The SAF project is a long-term project that is set to last for many years. Automation software such as process control and HMI have been implemented and virtually commissioned via the plant simulation. The recooling system and the control of the electrode columns have already been tested with the hardware. Further components of the furnace are being commissioned in progressive stages.

7.3 Process Control

Figure 7.3 shows the process control structure implemented for the SAF plant. This structure follows the model of operational resources and operational measures introduced in Section 2.2.5. Modular FB-agents compose a Multi Agent System: Operational Resource Agents control single field devices (e.g. valve, pumpe) or asset groups (e.g. pump station, heat exchanger). Operational Measure Agents are in charge of production procedures, engineering activities, diagnostic measures etc.

The process control of the SAF plant mainly uses two advantages of the FB-agents: flexible service-oriented cooperation and white-box engineering. Concrete use cases will be introduced in the following sections.

7.3.1 Service Oriented Interaction

In conventional process control systems, signals are applied as the operational resources for users. To perform operational measures, users should master all imple-

mentation details. In the agent-oriented system for the SAF plant, process control functionalities are encapsulated in single agents and provided as operations and services to the outside (cf. introductions in Section 3.2). Both the inter-agent interaction and the interaction between human users and agents are organized in a service-oriented manner.

Operational Measure Agents provide different services according to their individual responsibility such as “Production Recipe Execution” or “plant start-up”. Operational Resource Agents provide the service “Process Control”. All agents provide two services: “exploration” and “diagnostic”. The former one explores meta information, available services and operations of an agent. The latter one diagnoses the execution status and explores diagnostic information (e.g. the current setpoint) of an agent.

Operations are agent-specifically defined. For instance, the movement of the three top electrodes are controlled by three “electrode column control agents” which provide the operations of “RAISE”, “HOLD” and “LOWER”; an agent for a hydraulic pump with variable delivery capacity has the operations “QUICK”, “SLOW” and “STOP” which are arranged (or grouped) under the “Process Control” service. The “diagnostic” service of a pump control agent has two operations: “get Key Performance Integrators” and “get error status”. The “explore” service has two general underlying operations: “list registered services” and “list registered operations under a service”.

Operations abstract implementation details, whereas services group and abstract operations. In case the user wants to raise an electrode column, this execution objective can be described in form of a service request message. The electrode column agent can interpret the message and activate the operation “RAISE”. Implementation details, such as the steps that are to be followed, control signals that are to be set and check back signals that are to be observed are concealed behind the service-oriented interaction and are not to be mastered by the service requester (i.e. the user).

The “exploration” and “diagnostic” services can be provided to different requesters at the same time. A service provider is not allocated by its requesters. However, the “process control” service of an agent can only be provided to one requester. The arcs in Figure 7.3 show the cooperation relationship between agents in the context of process control. The target agent of an arc provides the “process control” service to the source agent which acts as the service requester. A service provider is dynamically allocated by its service provider and can only be released by it. Process control requests from further requesters will be rejected. Nevertheless, a special authority is reserved for the operator so that a manual access is always possible. Their cooperation relationship between agents can be fixedly defined or dynamically constructed. A lower level in the control hierarchy corresponds with a more rigid and fixed coupling between the agents.

Operational Resource Agents for single devices are rigidly connected with physical devices in the field via signal connections. Single control agents are normally controlled by the same group control agents. For instance, the allocation relationship between the

“heat exchanger control agent” and its four underlying agents for ventilator control is ridig and will not be changed.

The cooperation relationship between group control agents is fixed in most cases, but can also be dynamically changed in some cases. For instance, three closed-loop-control agents can allocate the three “electrode column control agents” and control the length of the electric arcs separately. The electrode column agents can also be allocated by a “moeller control agent” which control the raising and lowering of the three top electrodes in a synchronized manner.

All Operational Measure Agents are loosely coupled. They assign operational resources dynamically and release them when the operational measure is finished.

The modular encapsulation of agents and the dynamical cooperation between them allow a flexible modification and extension of the process control strategy. Agents can be modularly added or removed with a minimized influence on the existing implementation. For instance, the three electrode column agents were controlled manually by operators or automatically by close-loop control agents. The “moeller control” agent was newly added. It invokes the existing process control service provided by the three agents via dynamically generated messages. The service providers can recognize the sender of incoming messages and avoid simultaneous allocation automatically. Thus, the existing implementation and cooperation relationships do not need to be adapted. In traditional process control systems, complex allocation algorithms should be defined in this case.

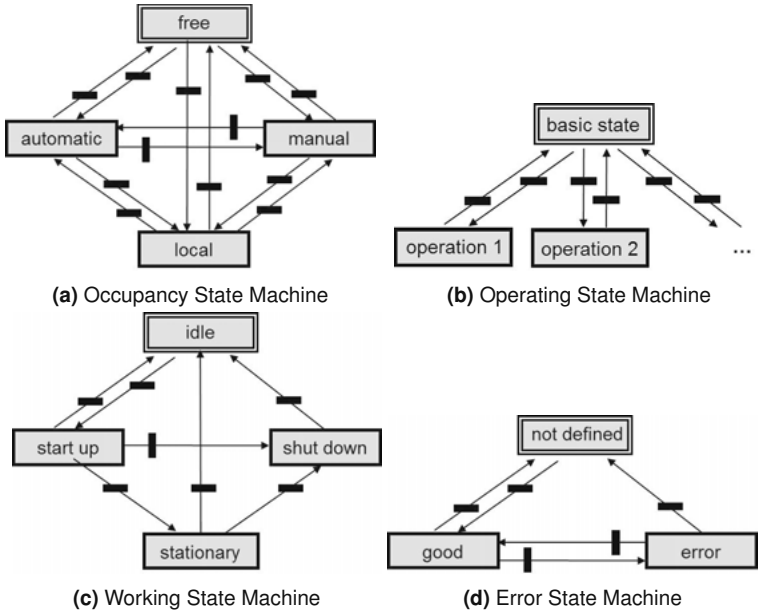
With the modular and flexible extendable structure realized by FB-agents, the process control engineering could be started, while the mechanical engineering of the SAF plant is still not finished. A service or operation can be registered and invoked, before the underlying implementation is completely defined. For instance, the “RAISE”, “HOLD” and “LOWER” operations of the electrode columns were firstly defined and used for the design and test of the superordinate control algorithms, before the number of underling hydraulic valves was set to be two. A further example is the recooling system. Its interaction with further plant segments is organized in a service-oriented manner. Although its structure was redesigned and new pumps and valves were added, its services and operations were not changed. The hydraulic system, electrode column control and interlock algorithms should not be tested again. Along with the progression of the SAF project, the process control strategy is continually extended and optimized. Among others, the number of field devices has been doubled untill 2013. Experience has shown that the implemented process control can be adapted with minimal reengineering work.

The present process control concept focuses mainly on service orientation and flexible engineering during the engineering phase of the SAF plant. The action scope of autonomy is deliberately kept small. On the basis of the implemented agent-oriented structure, advanced autonomous algorithms can be added. For instance, knowledge bases can be developed which recognize the situation of the environment in abnormal states and lead the plant to a safe state autonomously. Agents with self-x functions

can be developed that can, for instance, perform partial stroke tests for on/off valves autonomously.

7.3.2 White-Box Engineering

The white-box engineering realized by the FB-agent model allows users to register and deregister services and operations. Internal logics of FB-agents can also be defined, manipulated and extended by users. Continuous functions can be defined as Function Block Diagrams (FBD). Sequential procedures (e.g. control sequences) and state-based procedures (e.g. state machines) can be described in Sequential State Charts (SSC) (cf. Chapter 5).



(a) Occupancy State Machine **(b)** Operating State Machine

(c) Working State Machine **(d)** Error State Machine

Figure 7.4: Sequential State Charts (SSC) for the Status Representation of Single Control Agents.

As shown in Figure 7.4, every Operational resource Agent (ORA) has four standard SSCs for the representation of occupancy state, operating state, working state and error state. Operational measure Agents (OMA) also have three standard SSCs for representing the occupancy state, error state and the life-cycle phase (cf. Figure 7.5). All SSCs can be tailored for individual agents.

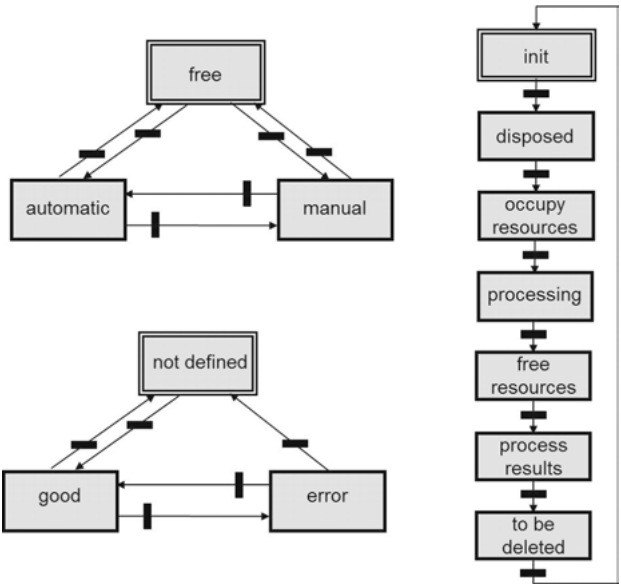


Figure 7.5: Standard state machines for operational measure agents

Figure 7.6 shows the control logic within an electrode column. Three SSCs are defined for raising, holding and lowering the electrode column. Each of the three SSCs has four standard states: idle, stationary, start-up and shutdown. Each of the last two states has a sub-procedure which is connected via signal connections to the main-SSC. Another SSC that is set up according to Figure 7.4b controls the transition between the three operations and a predefined basic state. In case an operation which is different to the current operation is requested, the current one will be shut down and lead to its idle state. The new operations will be started up afterwards.

The three main SSCs are registered as operations: “RAISE”, “HOLD” and “LOWER”. Operations are arranged under the service “Process Control”. Every operation has a representative object of the class $operation$ introduced in Section 6.2. Service requests on the process control service will be saved as signal variables (e.g. operation, parameter) of the service inbox “Process Control” (cf. Figure 3.5). These variables can be read via signal connections of the SSCs shown in Figure 7.6.

The introduction in this section has indicated that the general procedure description method Sequential State Chart (SSC) can appropriately describe various procedures for process control and state representation. No application-specific description methods are needed.

.

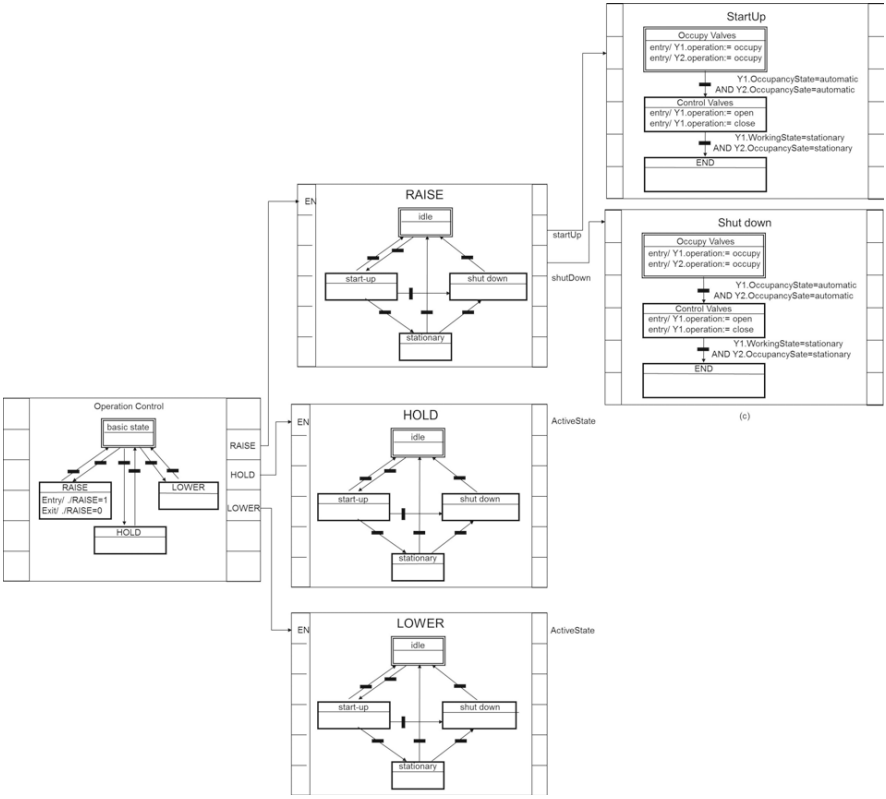


Figure 7.6: Control Logic of an Electrode Column

7.4 Knowledge-based Engineering

Many control logics (e.g. valve control) of the SAF plant are repetitive. A fully manual engineering is time-consuming and error-prone. To accumulate the engineering process, engineering agents are applied. They serve as autonomous assistants for the user and can initiate implementations or check the completeness of existing implementations. The engineering knowledge of the agents are formally described according to the theoretical work ACPLT/RE[1] [66], which allows a rule-based and knowledge-based engineering of automation functionalies.

Section 7.4.1 presents the concept of agent-oriented engineering. Specific use cases will be introduced in Section 7.4.2. A prototype was implemented in a diploma thesis

[1] RE: acronym of Regelbasiertes Engineering (German Expression for Rule-based Engineering.)

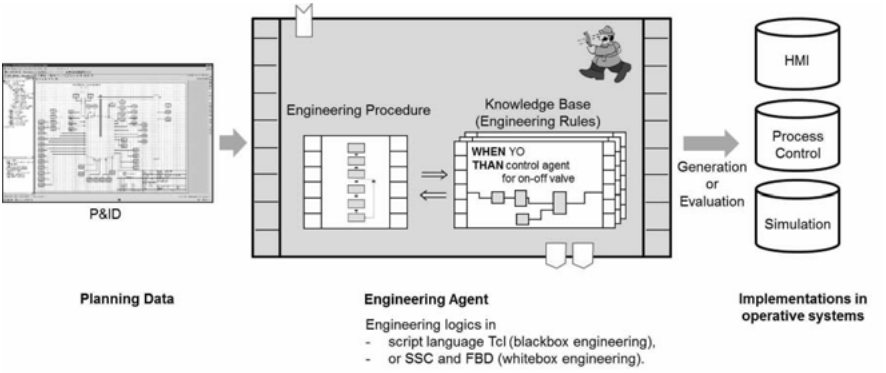


Figure 7.7: Eingeering Agent

and successfully applied during the SAF engineering. Some results have already been introduced in a previous publication of the author [104].

7.4.1 Concept

Figure 7.7 demonstrates the basic idea of the knowledge-based engineering. An engineering agent encapsulates procedures and fact knowledge for specific engineering objectives. It explores planning data of the plant and can provide the following two types of engineering services:

- Initialization of process control implementation in runtime systems,
- Evaluation of existing implementations

Planning data is defined in a Piping and Instrumentation Diagram (P&ID). It models the structure of the plant, connections between the instruments and signals that are to be exchanged between the process control level and field level. The planning data is present in electronic form and can be explored by engineering agents. The P&ID is defined in the CAE tool Comos and can be transformed into a CAEX model according to the standard IEC 62424 [105]. This model is present in the ACPLT/OV environment and can be explored by engineering agents.

The knowledge base of agents contains fact knowledge which is formulated as engineering rules in the standard form “IF premise, THAN conclusion”. A premise is a pattern for the exploration in the planning data. In case the pattern (e.g. a pump-valve module) is detected, its corresponding conclusion (e.g. “generate interlock logics”) will be performed. Engineering rules can be project-specific or project-neutral. Neutral rules can be applied in different projects.

Every agent encapsulates at least one *Engineering Procedure* which controls the progression of an engineering process. The procedure invokes engineering rules and performs actions such as “initialize an OV server for the simulation”, “load defined libraries onto the server” etc.

In the first prototype of the engineering agents, all engineering logics were programmed by a script language Tcl [106]. Script languages are not standard automation languages. Many solutions should be specially developed, among others, the communication interface to the runtime system, interfaces for operation and observation, solutions for error diagnosis etc. Additionally, users have to master an application-specific language with which they are not familiar. Experience in the SAF project shows that the automation engineers need at least two weeks of training before becoming familiar with the programming.

As an alternative to the textual script language Tcl, ACPLT/RE has been applied. In ACPLT/RE, engineering rules are formally described in the native automation language Function Block Diagram according to the IEC 61131-3 (cf. Section 2.2.2). ACPLT/RE provides a function block library for elementary engineering functions such as “create object”, “Link Object”, “Set Variable” and “Get Variable”. Engineering rules can be aggregated by using these function blocks. Most automation engineers of the SAF project are automation engineers, mechanical engineers or chemical engineers. The function block technology is an essential part of their academic study. During the SAF engineering, they only need to learn about the function blocks of the engineering library and can begin with the engineering work with no special training on the programming language. The white-box engineering realized by function blocks allows engineers to master and flexibly extend engineering rules.

7.4.2 Use Cases

Figure 7.8 depicts the engineering agents applied for the SAF project. Every single agent is responsible for a specific engineering work and provides engineering services such as “Initialization of Single Control Level” and “Analyze the plausibility of the existing I/O configuration”. The achievement of specific engineering objectives is ensured by the knowledge base encapsulated in individual agents. Specific use cases of the engineering agents will be introduced in the following paragraphs.

Engineering of I/O Configuration

Signals that are to be exchanged via field bus should be converted. Figure 7.9 shows the default Function Block Diagram (FBD) for processing a temperature signal. The data sent from the field bus is a real number between 0 and 100. It should be mapped to the range $[0°C, 1500°C]$, which is defined in the signal list. Additionally, a function block for signal monitoring is to be instantiated. It monitors up to four thresholds and generates

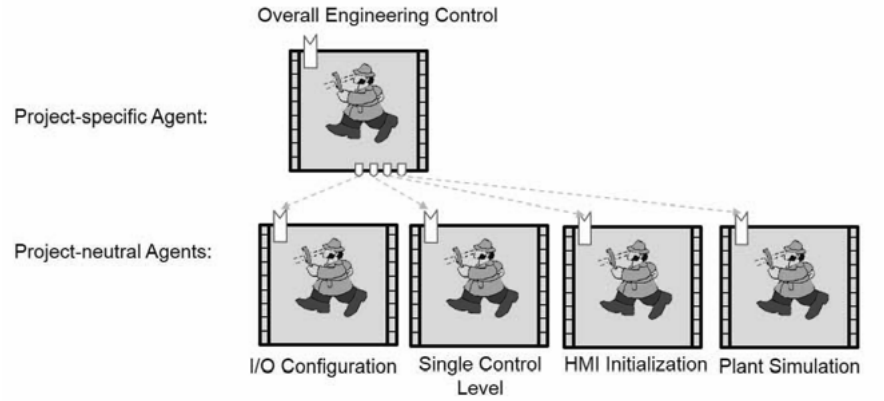


Figure 7.8: Agents for the initialization of automation applications of the SAF plant

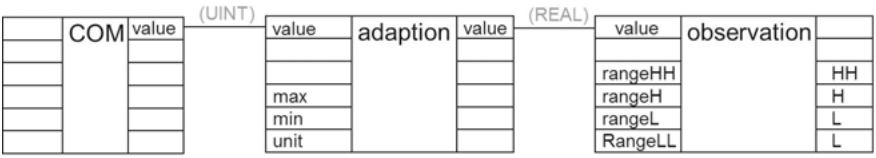


Figure 7.9: FBD to be generated for the conversion and the monitoring of a temperature signal (revised figure based on Figure 7 in [104]

four monitoring signals accordingly: very high (HH), high (H), low (L) and very low (LL). Similar FBDs are to be applied for every single analog input, analog output, digital input and digital output on the process control server.

An engineering agent for the service “Initialization of I/O configuration” is implemented. It instantiates the desired function blocks, creates signal connections and parameterizes the function blocks according to the entries (e.g. signal range, thresholds, hysteresis, warning text etc.) in the signal list.

All engineering activities are controlled by an engineering procedure shown in Figure 7.10. The procedure is described in Sequential State Chart (SSC). Every state realizes an engineering activity and calls an engineering rule. Every rule is implemented in form of a Function Block Diagram which consists of function blocks for the exploration of planning data and the configuration of function blocks etc.

The initial step of the Sequential State Chart (SSC) performs necessary configurations of the engineering rules through the assignment of parameters, such as the location of planning data. In a second step, the rule for generating the plant hierarchy in Figure 7.11 is invoked. Five hierarchical levels are automatically initialized: plant, instrument Com-

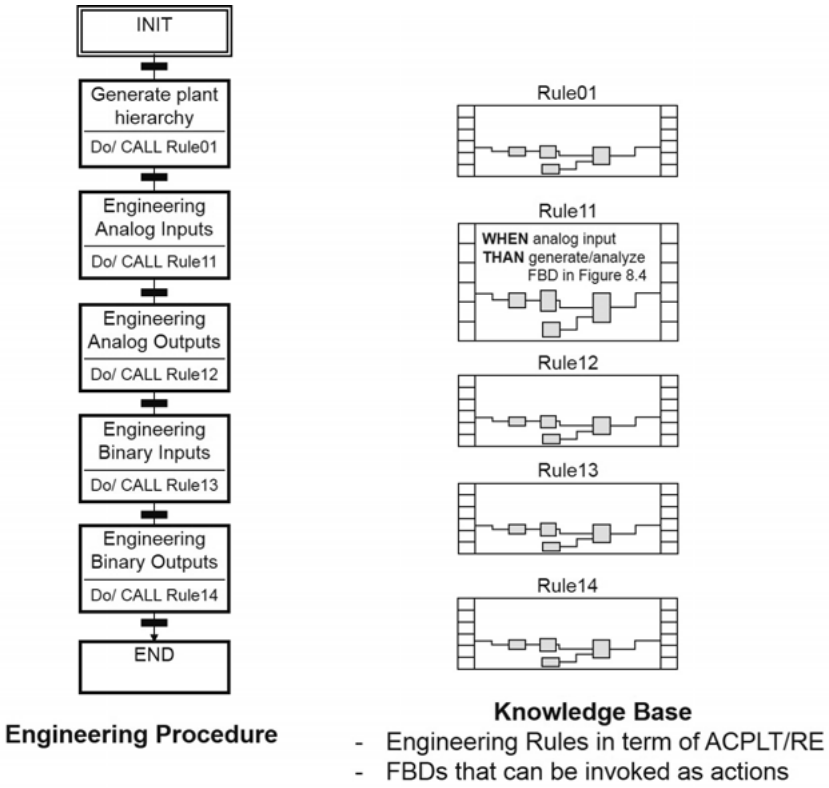


Figure 7.10: SSC and FB-actions for I/O configuration.

plex, plant unit, plant segement and technical unit. In the next four steps, engineering rules for different signal types are sequentially executed. For instance, Rule 11 creates and configures a copy of the FBD in Figure 7.9 for every analog input signal detected in the planning data.

The initialized logic can be manually changed in the engineering phase. For instance, parameters of the function blocks can be modified. A similar agent is implemented for the service “Check of I/O configuration”. In contrast to the initialization agent, the latter agent does not generate the function blocks but only checks their existence, necessary connections and the plausibility of their parameters. It is, for example, only plausible, in case all monitoring thresholds are within the given measurement range, and follow the rule of $HH > H > L > LL$.

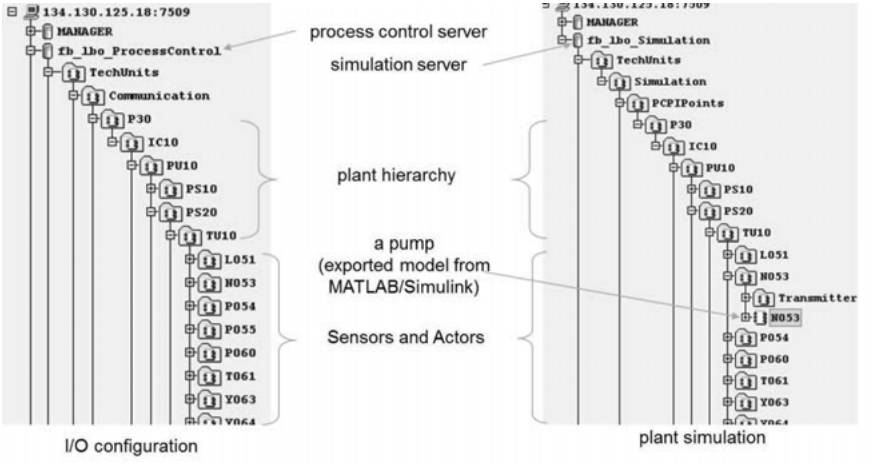


Figure 7.11: Automatically generated object structures (Screenshots from the engineering client iFBSpro @LTSoft

Engineering of Single Control Level

The SAF plant is equipped with many standard field devices such as pumps and on/off valves. Their corresponding single control agents (cf. Section 7.3) on the process control server are automatically generated by an engineering agent. Only the signal connections between the agents and the field devices, as well as specific interlock logics should be individually defined by the user.

The engineering agent for the process control initialization creates a single control agent for every actuator and configures its connections with I/O signals (e.g. set point, check back signals) which are generated by the aforementioned agent for I/O configuration.

A similar agent is defined which can check the plausibility of the existing single control level. For instance, it can check whether a manually engineered agent is correctly connected to the field device.

Engineering of Human Machine Interface (HMI):

The HMI uses many standard visualization elements, e.g. for monitoring sensor signals. An engineering agent for HMI is defined which can generate the standard elements and initiate their configuration.

As shown in Figure 7.12, every temperature signal has an exclusive visualization. The HMI engineering agent can generate the visualization element, set the signal name and

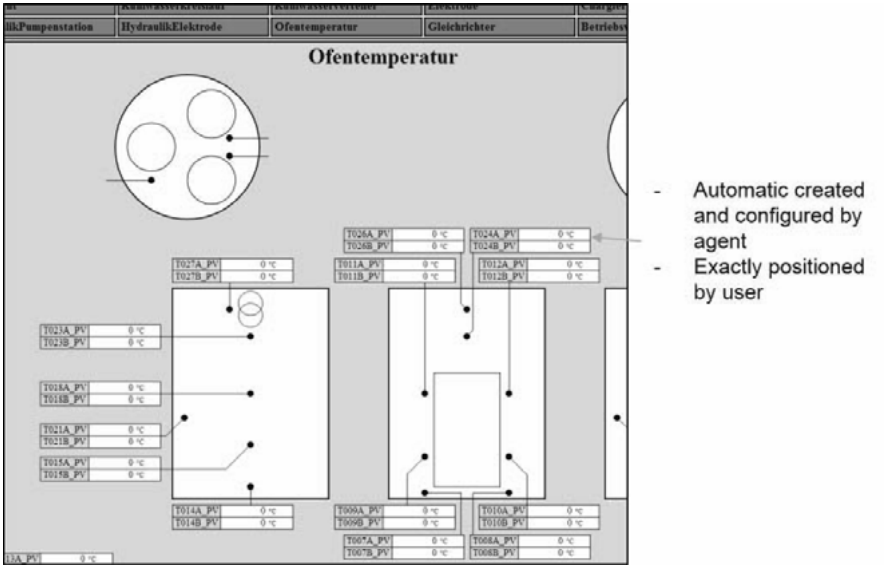


Figure 7.12: Visualisation of sensor value observation: automatically initialized and manually placed

the physical unit, link the element with the measuring signal form the I/O level, and initiate a solid line for indicating the mounting position of the sensor on the SAF vessel. Additionally, every signal has an exclusive face plate which is displayed by clicking on the signal. This face plate shows measurement range and monitoring thresholds of the signal and can also be automatically created by the HMI agent. All signal visualization, face plates and lines are initiated by the agent and then precisely positioned by automation engineers.

A similar checking agent is defined which can check whether all sensor signals are visualized and whether the HMI configuration is plausible.

Engineering of Plant Simulation:

In order to test the process control logic and to train operators, a simulation model for the SAF plant is applied. I/O signals, field devices and melting processes are simulated.

Every actor and sensor has an exclusive representative in the simulation. Simulation models for the field devices and processes were developed in MATLAB/Simulink in a diploma thesis. These models are compiled in ANSI C code and loaded to the simulation server (cf. Figure 7.2 and Figure 7.7). Figure 7.11 shows the object structure which is generated by the engineering agent. Initially, the agent generates the same plant

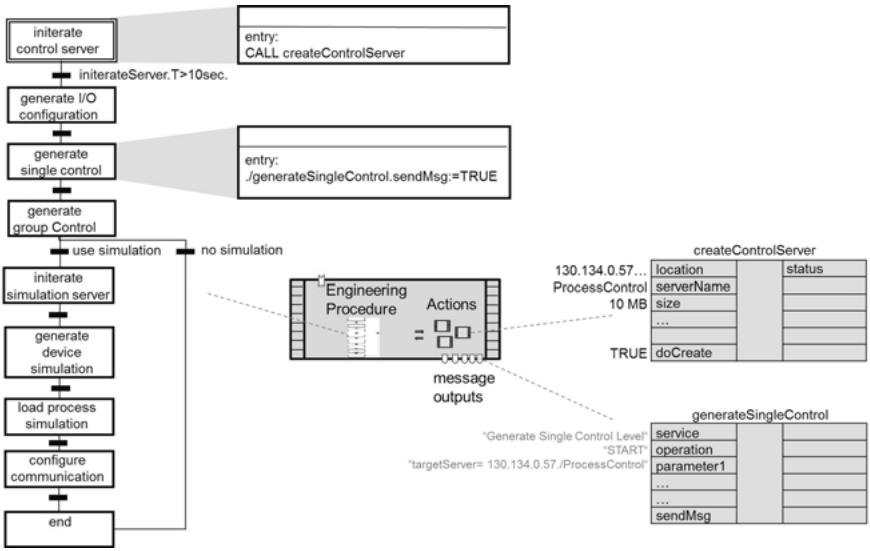


Figure 7.13: Procedure for Overall Engineering

hierarchy that takes effect in the process control hierarchy. It instantiates simulation models for field devices and then configures the communication between the simulation server and the process control server.

Aside from the agent for the simulation initialization, an agent for checking manually extended implementation is applied. The agent can check whether every single device has an exclusive simulation element and whether the communication between the simulation and the process control logic works properly.

Engineering of Automation System:

Agents for the previous four use cases are project-neutral and can be reused in further projects. A project-specific agent was developed which can initialize the whole automation system and control the aforementioned four agents.

Figure 7.13 shows the engineering procedure within this superordinate agent. This procedure is implemented as a Sequential Sate Chart (SSC). At the beginning of the procedure, a function block is invoked as an action which creates a server for process control (cf. Figure 7.2). It takes a while until the storage space is reserved and the server is started. After a waiting time of $10sec.$, the second state will be activated. In this state, a message output is activated which creates and delivers a service request to the aforementioned engineering agent for I/O configuration. Further services for the initialization of single control level, HMI and plant simulation will be requested sequentially in

the following states. Additionally, states can be defined for project-specific engineering activities. For instance, the group control level (cf. Figure 7.3) can be generated in an SSC state. Simulation models for special devices (e.g. power supply) and process models (e.g. thermodynamic model for a melting process) can be loaded at the end of the procedure.

7.4.3 Application Effects

The reference model FB-agent forms a design guideline for engineering agents. Engineering activities are modularly encapsulated and registered as services in individual agents. The engineering knowledge base is describted in Function Block Diagrams (FBDs) according to ACPLT/RE. Engineering procedures are described in Sequential State Charts (SSC). Both the FBD and the SSC realize a graphical representation and modular encapsulation of engineering logic. As the FBD and the SSC can be directly implemented in existing automation systems, special communication interfaces or APIs are not required.

In this context, the same execution frame, service communication and description methods that are applied for process control agents can be utilized. As the automation engineers of the SAF plant only need to master a unified design pattern for different application areas, the engineering workload can be reduced.

In traditional engineering processes, the process control strategy can only be implemented when the planning data will not be altered anymore. The engineering cost for a plant reconstruction is usually high. By using engineering agents, however, the process automation engineering of the SAF plant could be started in parallel with the mechanical engineering. The process control strategy can be quickly implemented in the runtime system and tested via a plant simulation. Implausibility and incompleteness in the planning data and the implementation in the runtime system can be automatically detected. The correctness and the completeness of a manually engineered implementation can be automatically tested by agents. All engineering rules and engineering procedures can be flexibly modified and extended by users.

8 Conclusions and Outlook

The present dissertation discusses engineering aspects of process automation agents. A reference model named FB-agent was defined which combines the advantages of function block technology, service orientation and agent orientation. The FB-agent model serves as a standard design pattern for the development of agents in different application areas (e.g. process control, engineering, model management, archiving and etc.)

In order to realize a flexible agent-agent or agent-human cooperation, internal logics of FB-agents are abstracted by means of service orientation. Elementary automation functions within an agent are to be registered as operations which represent the elementary functional ability of the agent. Operations are grouped and abstracted as services which can be explored and invoked from the outside. Services represent the execution objectives that can be achieved by agents. To perform an automation task, the user specifies the expected execution objective in form of a service request message. The service providing agent can interpret the service request and achieve the objective autonomously by using its local knowledge base and by cooperating with further agents. The cooperation relationship between agents can be fixedly defined or dynamically built at runtime.

The signal orientation in classic automation systems and the service-oriented (or message-oriented) communication from computer science are integrated in the FB-agent model. Viewed from the outside, an FB-agent is encapsulated as a function block which supports signal- and message-exchange with the environment. Data flow between internal components of agents is signal-oriented. Incoming messages from the outside are received by a message input of the agent and forwarded to internal message inboxes. The inboxes buffer and sort messages according to their objectives (e.g. service request) and types (e.g. invalid). Contents of the messages are converted to signals that can be processed by further components (e.g. function blocks) within the agent. Message outboxes collect internal signals, generate messages (e.g. service requests for other agents) and send them out.

Classic agent systems are normally implemented as black-boxes. Users do no need to acknowledge, how the agents work. However, users of automation systems should not be isolated from the design and implementation of automation agents. FB-agents applies a white-box structure and allows a user-centralized engineering. All components within an FB-agent shall be modularly encapsulated as function blocks. A flexible combination of atomic function blocks (black-box) and composed function blocks (white-

box) is allowed. Function block technology is highlighted as a basic modelling principle during the FB-agent engineering. Due to the native support of process control systems on the function blocks, FB-agents can be seamlessly integrated into the control flow and the existing systems. A user-friendly engineering is achieved, since users are already familiar with the function-block-driven engineering.

FB-agents can be integrated into runtime systems which can be call-based, event-driven or cyclic-processing. The execution of all function blocks within an agent is controlled by an internal task list which ensures a strictly deterministic execution. Runtime behaviors of the task list are clearly defined in a formal model according to UPPAAL's automata which can appropriately support the validation and verification of FB-agents in runtime systems.

The selection of description methods is the second main aspect that was addressed during the present work. Continuous functions within FB-agents can be described in Function Block Diagrams (FBD). For the description of state-based and sequence-based procedures, existing Procedure Description Methods (PDMs) in industrial automation and computer science are evaluated. The completeness of syntax, the unambiguity of semantics and the compatibility with existing automation systems were addressed as main criteria during the comparison. Sequential Function Chart (SFC), PLC-Statechart and Procedure Function Chart (PFC) were determined as possible approaches for the specification and implementation of FB-agents. However, none of the cited methods is suitable for being applied as a general approach for the description of diverse state-based and sequence-based procedures within FB-agents in their present form. In order to lower the engineering effort and raise the intuition, a general description approach called Sequential State Chart (SSC) has been developed.

Most existing procedure description methods are developed for specific application domains and can provide domain-specific syntax and semantics. The development of the general approach Sequential State Chart, however, focuses on the description of simple automation procedures which have no over-complex structure but are dominant in process automation. Based on the evaluation of existing description methods, ambiguously or neglected details have been identified. Unified solutions for the following design aspects are defined for the Sequential State Chart: the prioritization of alternative transitions, the execution order for actions, the execution of concurrent sequences, the progression of the entire procedure and the state representation in the idle time between two execution cycles. Additionally, every Sequential State Chart has an explicitly defined execution frame which encapsulates the chart, controls the deterministic execution of internal components and manages the data exchange with the environment. Semantics of SSCs are also described in the form of UPPAAL's automata which offer a formal analysis of SSCs in runtime systems.

Prototypes of FB-agent and Sequential State Chart have been tested in a practical project in which a metallurgical furnace SAF is automated. Experiences show that the modularity and the flexibility of process control are improved by using process control

agents. The engineering effort and the time consumption are also significantly reduced by using engineering agents. Additionally, various state machines and sequential procedures for the automation of the furnace can be appropriately specified and implemented in the form of Sequential State Charts. No application-specific procedure description methods were needed during the entire project.

The general approach Sequential State Chart (SSC) is not a novel description method. It is developed on the basis of a synthesis of the best practice of existing procedure description methods. The SSC is mainly characterized by its simple design, unambiguous semantics and the compatibility with existing automation runtimes. It can also be regarded as a reference model for the specification and implementation of automation procedures.

The reference model FB-agent and the general description method (or model) of Sequential State Chart establish a working platform for the description of process automation functions and for the construction of automation systems. Although the present dissertation focuses on agent engineering, the development of non-agent automation functions can also benefit from the discussions and design decisions that were made in the context of FB-agent and Sequential State Chart, e.g. modular encapsulation, integration of signal- and service-orientation, white-box engineering etc. These two models can be used as a guideline for the design of future automation systems.

The present work focusses on the construction of a general framework and on the selection of description methods for the development of automation agents in existing automation systems. The autonomous behaviors of agents were not intensively discussed. The scope of action of the example FB-agents is deliberately kept low. The development of advanced autonomous functions (e.g. self-x functions) for process automation is addressed as a central work for the future.

Bibliography

[1] M. Polke, *Process Control Engineering*. Weinheim, Germany: VCH, 1994.

[2] R. Lauber and P. Göhner, *Prozessautomatisierung II*. Germany: Springer-Verlag, 1999.

[3] *IEC 61131-3: Programmable controllers - Part 3: Programming languages*, 2013. 3rd Edition.

[4] *IEC 61131-5: Programmable controllers - Part 5: Communications*, 2000.

[5] *IEC TR 61131-8: Programmable controllers - Part 8: Guidelines for the application and implementation of programming languages, Technical Report*, 2003. 2nd Edition.

[6] *IEC 60050-351: International electrotechnical vocabulary - Part 351: Control technology*, 2013.

[7] *VDI/VDE 3681 Guideline: Classification and evaluation of description methods in automation and control technology*, 2005.

[8] *DIN EN ISO 10628-2: Diagrams for the chemical and petrochemical industry-Part 2: Graphical symbols (ISO 10628-2:2012); German version EN ISO 10628-2:2012*, 2013.

[9] *IEC 61512-2: Batch control - Part 1: Models and terminology*, 2002.

[10] *ISA-106 TR01: Procedure Automation for Continuous Process Operations - Models and Terminology, Technical Report*, 2013.

[11] *UML: Unified Modeling Language, V2.4.1*, 2012.

[12] "OMG's meta object facility." `http://www.omg.org/mof/`. Accessed: 2014-09-10.

[13] A. Münnemann, *Infrastrukturmodell zur Integration expliziter verhaltensbeschreibungen in die operative Prozessleittechnik*. PhD thesis, Chair of Process Control Engineering, RWTH Aachen University, Germany, 2005.

[14] *IEC 61499: Function blocks for industrial-process measurement and control systems*, 2000.

[15] K. Thramboulidis, “IEC 61499: Back to the well proven practice of IEC 61131?,” in *ETFA2012: 17th IEEE International Conference on Emerging Technologies and Factory Automation*, (Krakow, Poland), 2012.

[16] S. Grüner and U. Epple, “Paradigms for unified runtime systems in industrial automation,” in *ECC: Proceedings of the 12th European Control Conference*, (Zurich), pp. 3925–3930, IEEE, July 2013.

[17] W. Dai, V. Dubinin, and V. Vyatkin, “Migration from PLC to IEC 61499 using semantic web technologies,” *IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans*, vol. in print, 2013.

[18] D. Witsch and B. Vogel-Heuser, “PLC-statecharts: An approach to integrate uml-statecharts in open-loop control engineering - aspects on behavioral semantics and model-checking,” in *Preprints of the 18th IFAC World Congress Milano (Italy)*, 2011.

[19] L. Yu, S. Grüner, and U. Epple, “An engineerable procedure description method for industrial automation,” *ETFA2013: 18th Conference on Emerging Technologies and Factory Automation*, 2013.

[20] U. Enste, *Generische Entwurfsmuster in der Funktionsbausteintechnik und deren Anwendung in der operativen Prozessführung*. PhD thesis, Chair of Process Control Engineering, RWTH Aachen University,Germany, 2001.

[21] U. Enste and M. Fedai, “Flexible process control structures in multi-product and redundant-routing-plants,” in *MMM 98 – 9th IFAC Symposium on Automation in Mining, Mineral and Metal Processing*, Elsevier Science, 1998.

[22] S. Schmitz, A. Münnemann, and U. Epple, “Component modell for systematic design of process control functions,” in *GMA Congress 2005, VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik*, pp. 817–824, 2005.

[23] A. Münnemann, U. Enste, and U. Epple, “Hybrid modelling of complex process control function blocks,” in *S. Engel, G. Frehse, E. Schneider (Eds.): Modelling, Analysis, and Design of Hybrid Systems*, Springer-Verlag Berlin Heidelberg New York, 2002.

[24] L. Yu, G. Quirós, and U. Epple, “Service-oriented process control for complex multifunctional plants: Concept and case study,” in *ETFA 2010: 15th IEEE International Conference on Emerging Technologies and Factory Automation*, (Bilbao), IEEE, sep 2010.

[25] H. Mersch, M. Schlütter, and U. Epple, “Classifying services for the automation environment,” in *ETFA 2010: 15th IEEE International Conference on Emerging Technologies and Factory Automation*, 2010.

[26] L. Evertz and U. Epple, “Laying a basis for service systems in process control,” in *ETFA 2013: IEEE 18th Conference on Emerging Technologies and Factory Automation*, 2013.

[27] “WS-Trust 1.4. OASIS standard.” `http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.html`, 2009.

[28] M. Schlütter, U. Epple, and T. Edelmann, “On service-orientation as a new approach for automation environments,” in *Proceedings of MATHMOD 2009 – 6th Vienna International Conference on Mathematical Modelling*, vol. 2, pp. 2426–2431, 2009.

[29] Y. Natis, “Service-oriented architecture scenario,” *Gartner, ID Number: AV-19-6751*, 2003.

[30] *IEC 62541: OPC unified architecture - Part 1: Overview and concepts*, 2010.

[31] *NAMUR Recommendation NE141: Interface between Batch and MES Systems*, 2012.

[32] “W3c recommendation: Extensible markup language (xml) 1.0 (fifth edition).” `http://www.w3.org/TR/REC-xml/`, 2008.

[33] M. Gaspari, “Concurrency and knowledge-level communication in agent languages,” *Artificial Intelligence*, vol. 105, pp. 1–45, 1998.

[34] *VDI/VDE 2653 Guideline: Multi-agent systems in industrial automation - fundamentals*, June 2010.

[35] U. Epple, “Agentensysteme in der Leittechnik,” *atp - Automatisierungstechnische Praxis*, vol. 42, pp. 45–51, 2000.

[36] P. Göhner, P. G. de A. Urbano, and T. Wagner, “Softwareagenten - Einführung und überblcok über eine alternative Art der Softwareengwicklung teil 3: Agentensysteme in der Automatisierungstechnik: Aufbau, Strukture und Implementierung an einem Anwendungsbeispiel,” *atp - Automatisierungstechnische Praxis*, 2004.

[37] S. Eberle and P. Göhner, “Softwareentwicklung für eingebettete Systeme mit strukturierten Komponenten. teil 1+2,” *atp - Automatisierungstechnische Praxis*, vol. 46, 2004.

[38] S. Franklin and A. Graesser, “Is it an agent, or just a program?: A taxonomy for autonomous agents,” in *The Third International Workshop on Agent Theories, Architectures, and Languages*, 1996.

[39] M. Wooldridge, *An introduction to multiagent systems*. John Wiley & Sons, 2009.

[40] J. Odell, “Agent technology: An overview,” 2010.

[41] R. Guttman, A. Moukas, and P. Maes, “Agent-mediated electronic commerce: A survey,” *The Knowledge Engineering Review*, vol. 13, pp. 147–159, 1998.

[42] E. Ferreira, E. Subrahmanian, and D. Manstetten, “Intelligent agents in decentralized traffic control,” in *IEEE Intelligent Transportation Systems Conference Proceedings*, 2001.

[43] J. France and A. Ghorbani, “A multiagent system for optimizing urban traffic,” in *IEEE/WIC International Conference on Intelligent Agent Technology*, 2003.

[44] K. Mizuno, Y. Fukui, and S. Nishihara, “Urban traffic signal control based on distributed constraint satisfaction,” in *the 41st Hawaii International Conference on System Sciences*, 2008.

[45] X. Zhao and J. Zhao, “Research on model of resource management for traffic grid,” *Procedia Engineering*, vol. 15, pp. 1476–1480, 2011.

[46] J. Lagorse, D. Paire, and A. Miraoui, “A multi-agent system for energy management of distributed power sources,” *Renewable Energy*, vol. 35, pp. 174–182, 2010.

[47] G. Rohbogner and S. Fey, “What the term agent stands for in the smart grid definition of agents and multi-agent systems from an engineer’s perspective,” in *Proceedings of the Federated Conference on Computer Science and Information Systems*, 2012.

[48] J. Zeng, J. Liu, J. Wu, and H. Ngan, “A multi-agent solution to energy management in hybrid renewable energy generation system,” *Renewable Energy*, vol. 36, pp. 1352–1363, 2011.

[49] N. Jennings, “Agent-oriented software engineering,” in *Multiple Approaches to Intelligent Systems*, vol. 1611 of *Lecture Notes in Computer Science*, pp. 4–10, Springer Berlin Heidelberg, 1999.

[50] J. Bagherzadeh and S. Arun-Kumar, “Flexible communication of agents based on fipa-acl,” *Electronic Notes in Theoretical Computer Science*, vol. 159, pp. 23–39, 2006.

[51] F. Bellifemine, A. Poggi, and G. Rimassa, “JADE - a FIPA-compliant agent framework,” in *PAAM 1999: 4th International Conference on Practical Application of Intelligent Agents and Multi-Agent Technology*, vol. 99, pp. 97–108, IEEE, 1999.

[52] “Java Agent DEvelopment framework.” `http://jade.tilab.com/`. Accessed: 2014-09-16.

[53] S. Pech, *Agentenbasierte Informationsgewinnung für automatisierte Systeme*. PhD thesis, Institut für Automatisierungs- und Softwaretechnik, University Stuttgart, Germany, 2014.

[54] H. Mubarak and P. Göhner, “An agent-oriented approach for self-management of industrial automation systems,” in *INDIN 2010: 8th IEEE International Conference on Industrial Informatics*, 2010.

[55] H. Mubarak, *Agentenbasiertes Selbstmanagement von Automatisierungsanlagen*. PhD thesis, Institut für Automatisierungs- und Softwaretechnik, University Stuttgart, Germany, 2013.

[56] A. Wannagat and B. Vogel-Heuser, “Increasing flexibility and availability of manufacturing systems-dynamic reconfiguration of automation software at runtime on sensor faults,” *Journal of Automation, Mobile Robotics & Intelligent Systems*, vol. 3, pp. 47–53, 2009.

[57] D. Schütz, M. Schraufstetter, J. Folmer, B. Vogel-Heuser, T. Gmeiner, and K. Shea, “Highly reconfigurable production systems controlled by real-time agents,” in *ETFA2011: 16th IEEE Conference on Emerging Technologies Factory Automation*, pp. 1 –8, sept. 2011.

[58] U. Epple, “Agentenorientierte Modelle in der Anlagenautomation,” in *Agentensysteme in der Automatisierungstechnik* (P. Göhner, ed.), pp. 95–110, Springer-Vieweg, 2013.

[59] L. Yu, A. Schüller, and U. Epple, “On the engineering design for systematic integration of agent-orientation in industrial automation,” in *10th IEEE International Conference on Control & Automation*, 2013.

[60] *ISO/IEC 9126-1: Software engineering - Product quality, Part 1: Quality model*, 2001.

[61] A. Wannagat, *Entwicklung und Evaluation agentenorientierter Automatisierungssysteme zur Erhöhung der Flexibilität und Zuverlässigkeit von Produktionsanlagen*. PhD thesis, Lehrstuhl für Automatisierung und Informationssysteme, TU München, Germany, 2014.

[62] G. Bollella and J. Gosling, “The real-time specification for Java,” *Computer*, vol. 33, pp. 47–54, 2000.

[63] L. Yu, G. Quirós, T. Krausser, and U. Epple, “ACPLT + IEC 61131-3 = Dynamic Reconfigurable Models,” *Bamberg, Germany*, pp. 90–91, 2012.

[64] F. Uecker, *Konzept zur Prozessdatenvalidierung für die Prozessleittechnik*. PhD thesis, Chair of Process Control Engineering, RWTH Aachen University, 2005.

[65] R. Jorewitz, A. Münnemann, U. Epple, R. Böckler, W. Wille, and R. Schmitz, “Automated treatment of balances,” in *MATHMOD 2006: 5th Vienna Symposium on Mathematical Modelling*, vol. 30, pp. 4–1 – 4–13 (Vol. 2), AGRESIM-Verlag, 2006.

[66] T. Krausser, G. Quirós, and U. Epple, “An IEC-61131-based rule system for integrated automation engineering: Concept and case study,” in *IEEE INDIN 2011: 9th IEEE International Conference on Industrial Automation*, (Lisbon), IEEE, July 2011.

[67] S. Runde, A. Fay, S. Schmitz, and U. Epple, “Wissensbasierte Systeme im Engineering der Automatisierungstechnik - knowledge-based system for the engineering of automation systems,” *at - Automatisierungstechnik*, vol. 59, pp. 42–49, Jan. 2011.

[68] S. Schmitz, M. Schlütter, and U. Epple, “Automation of automation - definition, components and challenges,” in *ETFA 2009: 14th IEEE International Conference on Emerging Technologies and Factory Automation*, IEEE, Sept. 2009.

[69] W. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, 1943.

[70] E. Moore, “Gedanken-experiments on sequential machines,” in *Automata Studies, Annals of Mathematical Studies*, pp. 129–153, Princeton University Press, 1956.

[71] G. Mealy, “A method to synthesizing sequential circuits,” *Bell System Technical Journal*, vol. 34, pp. 1045–1079, 1955.

[72] D. Harel, “Statecharts: A visual formalism for complex systems,” *Science of Computer Programming*, vol. 8, pp. 231–274, 1987.

[73] D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, 1998.

[74] D. Witsch, M. Ricken, B. Kormann, and B. Vogel-Heuser, “PLC-statecharts: An approach to integrate UML-statecharts in open-loop control engineering,” in *INDIN 2010: 8th IEEE International Conference on Industrial Informatics*, pp. 120–125, July 2006.

[75] *ISO/IEC 19501: Information technology - open distributed processing - Unified Modeling Language (UML)*, 2005.

[76] C. Petri, *Kommunikation mit Automaten*. PhD thesis, Fachbereich für Mathematik und Physik, TU Darmstadt, 1962.

[77] K. Jensen and G. Rozenberg, *High-Level Petri Nets: Theory and Application*. Springer-Verlag, 1991.

[78] D. Abel, *Petri-Netze für Ingenieure: Modellbildung und Analyse diskret gesteuerter Systeme*. Springer-Verlag, 1990.

[79] *ISO/IEC 15909-1: Systems and software engineering - High-level Petri nets - Part 1: Concepts, definitions and graphical notation*, 2004.

[80] *ISO/IEC 15909-2: Systems and software engineering - High-level Petri nets - Part 2: Transfer format*, 2011.

[81] B. Falko, “Analysis of petri nets with a dynamic priority method,” *Application and Theory of Petri Nets 1997*, vol. 1248, pp. 359–376, 1997.

[82] G. Berthelot, “Transformations and decompositions of nets-,” *Petri Nets: Central models and their properties*, vol. 254, pp. 359–376, 1987.

[83] B. Graves, “Computing reachability properties hidden in a finite net unfolding,” *Foundations of Software Technology and Theoretical Computer Science*, vol. 1346, pp. 327–341, 1997.

[84] A. Kondratyev, M. Kishinevsky, A. Taubin, and S. Ten, “A structural approach for the analysis of petri nets by reduced unfoldings,” *Application and Theory of Petri Nets 1996*, vol. 1091, pp. 346–365, 1996.

[85] R. Lewis, *Programming Industrial Control Systems Using IEC 1131-3*. Institution of Electrical Engineeres: VCH, 1998.

[86] K. John and M. Tiegelkamp, *SPS-Programmierung mit IEC 61131-3, 4. neubearb. Aufl.* Springer-Verlag, 2009.

[87] N. Bauer, S. Engell, R. Huuck, S. Lohmann, B. Lukoschus, M. Remelhe, , and O. Stursberg, “Verification of PLC programs given as sequential function charts,” in *INT2004, LNCS3147*, 2004.

[88] N. Bauer, *Formale Analyse von Sequential Function Charts (In English: Formal Analysis of Sequential Function Charts)*. PhD thesis, Chair of Process Dynamics and Operations, TU Dortmond, Germany, 2004.

[89] N. Bauer, R. Huuck, B. Lukoschus, and S. Engell, “A unifying semantics for sequential function charts,” *Integration of Software Specification Techniques for Applications in Engineering*, vol. 3147, pp. 400–418, 2004.

[90] A. Hellgren, M. Fabian, and B. Lennartson, “On the execution of sequential function charts,” *Control Engineering Practice*, vol. 13, pp. 1283–1293, 2004.

[91] L. Yu, G. Quirós, T. Krausser, and U. Epple, “SFC-based process description for complex automation functionalities,” in *EKA2012: Entwurf komplexer Automatisierungssysteme, 12. Fachtagung*, (Magdeburg), pp. 13 – 20, ifak Institut für Automation und Kommunikation e.V., may 2012.

[92] S. Bornot, R. Huuck, Y. Lakhnech, and B. Lukoschus, “An abstract model for sequential function charts,” *Discrete Event Systems*, pp. 255–264, 2000.

[93] N. Bauer and R. Huuck, “A parameterized semantics for sequential function charts,” in *Proceedings of the semantic foundations of engineering design languages, Satellite Event of ETAPS*, 2002.

[94] *IEC 60848: GRAFCET specification language for sequential function charts*, 2013. 3rd Edition.

[95] F. Schumacher and A. Fay, “Konzept und werkzeugunterstützung zur automatischen generierung von IEC 61131-3 konformen steuerungsalgorithmen auf basis einer grafcet-spezifikation,” in *Automation 2013*, 2013.

[96] A. Schüller and U. Epple, “Ein referenzmodell zur prozedurbeschreibung: eine basis für industrie 4.0,” *at - Automatisierungstechnik*, vol. 63, no. 2, pp. 87–98, 2015.

[97] D. Meyer, *Objektverwaltungskonzept für die operative Prozessleittechnik*. PhD thesis, Chair of Process Control Engineering, RWTH Aachen University, 2001.

[98] H. Albrecht, *On Meta-Modeling for Communication in Operational Process Control Engineering*. PhD thesis, Chair of Process Control Engineering, RWTH Aachen University, 2003.

[99] R. Jorewitz, *Eine strukturelle Beschreibungsmethodik zur automatisierten Erzeugung von Prozessbewertungen in der operativen Prozessleittechnik*. PhD thesis, Chair of Process Control Engineering, RWTH Aachen University, 2011.

[100] G. Quirós, *Model-based Decentralised Automatic Management of Product Flow Paths in Processing Plants*. PhD thesis, Chair of Process Control Engineering, RWTH Aachen University, 2011.

[101] S. Schmitz, *Grafik- und Interaktionsmodell für die Vereinheitlichung grafischer Benutzungsschnittstellen der Prozessleittechnik*. PhD thesis, Chair of Process Control Engineering, RWTH Aachen University, 2010.

[102] “Mini-xml.” `http://www.msweet.org/projects.php?Z3`, 2013. Accessed: 2014-11-15.

[103] H. Jeromin and U. Epple, “Anwendungs- und herstellerneutrales Modell zur Darstellung und Interaktion mit leittechnischen Funktionen,” in *In Automation 2012: der 13. Branchentreff der Mess- und Automatisierungstechnik / VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik*, 2012.

[104] T. Krausser, L. Yu, and S. Schmitz, “Regelbasierte Vollständigkeitsüberprüfung von Automatisierungslösungen,” in *VDI-Berichte 2092, Automation 2010: Leading through Automation*, (Düsseldorf), pp. Kurzfassung: S. 55–58, Langfassung: auf beiliegender CD, VDI Verlag, June 2010.

[105] *IEC 62424: Representation of process control engineering - Requests in P&I diagrams and data exchange between P&ID tools and PCE-CAE tools*, 2008. 1st Edition.

[106] “Tcl developer xchange site.” `http://www.tcl.tk/`, 2013. Accessed: 2014-09-10.

Online-Buchshop für Ingenieure

VDI nachrichten **BUCHSHOP**

Online-Shops

Fachliteratur und mehr - jetzt bequem online recherchieren & bestellen unter: www.vdi-nachrichten.com/Der-Shop-im-Ueberblick

Täglich aktualisiert: Neuerscheinungen VDI-Schriftenreihen

Im Buchshop von vdi-nachrichten.com finden Ingenieure und Techniker ein speziell auf sie zugeschnittenes, umfassendes Literaturangebot.

Mit der komfortablen Schnellsuche werden Sie in den VDI-Schriftenreihen und im Verzeichnis lieferbarer Bücher unter 1.000.000 Titeln garantiert fündig.

Im Buchshop stehen für Sie bereit:

VDI-Berichte und die Reihe **Kunststofftechnik:**

Berichte nationaler und internationaler technischer Fachtagungen der VDI-Fachgliederungen

Fortschritt-Berichte VDI:

Dissertationen, Habilitationen und Forschungsberichte aus sämtlichen ingenieurwissenschaftlichen Fachrichtungen

Newsletter „Neuerscheinungen":

Kostenfreie Infos zu aktuellen Titeln der VDI-Schriftenreihen bequem per E-Mail

Autoren-Service:

Umfassende Betreuung bei der Veröffentlichung Ihrer Arbeit in der Reihe Fortschritt-Berichte VDI

Buch- und Medien-Service:

Beschaffung aller am Markt verfügbaren Zeitschriften, Zeitungen, Fortsetzungsreihen, Handbücher, Technische Regelwerke, elektronische Medien und vieles mehr – einzeln oder im Abo und mit weltweitem Lieferservice

Die Reihen der Fortschritt-Berichte VDI:

1 Konstruktionstechnik/Maschinenelemente
2 Fertigungstechnik
3 Verfahrenstechnik
4 Bauingenieurwesen
5 Grund- und Werkstoffe/Kunststoffe
6 Energietechnik
7 Strömungstechnik
8 Mess-, Steuerungs- und Regelungstechnik
9 Elektronik/Mikro- und Nanotechnik
10 Informatik/Kommunikation
11 Schwingungstechnik
12 Verkehrstechnik/Fahrzeugtechnik
13 Fördertechnik/Logistik
14 Landtechnik/Lebensmitteltechnik
15 Umwelttechnik
16 Technik und Wirtschaft
17 Biotechnik/Medizintechnik
18 Mechanik/Bruchmechanik
19 Wärmetechnik/Kältetechnik
20 Rechnerunterstützte Verfahren (CAD, CAM, CAE CAQ, CIM ...)
21 Elektrotechnik
22 Mensch-Maschine-Systeme
23 Technische Gebäudeausrüstung

ISBN 978-3-18-5**24808**-5