A Sequential Control Language for Industrial Automation

Theorin, Alfred

2014

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

A Sequential Control Language for Industrial Automation

Alfred Theorin

LUND
UNIVERSITY

Department of Automatic Control

PhD Thesis
ISRN LUTFD2/TFRT--1104--SE
ISBN 978-91-7623-110-4 (print)
ISBN 978-91-7623-111-1 (web)
ISSN 0280–5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Printed in Sweden.
Lund 2014

To Lisa and Rikard

Abstract

Current market trends for industrial automation are the need for customizable production, shorter time to market, and powerful global competitive pressure. Based on these trends two challenges have been identified: 1) flexible production systems and 2) integration and utilization of devices and software. Applications from both process automation, manufacturing, and robotics have been considered.

More flexible languages and tools are needed to get a flexible production system. The graphical programming language Grafchart, based on the IEC 61131-3 standard language Sequential Function Charts (SFC), is considered with the aim to make both the language and its implementation more flexible. In particular, new constructs have been added to the Grafchart language and modern compiler techniques are evaluated for JGrafchart, a Grafchart implementation, with focus on an extensible language implementation. A first step toward real-time execution of Grafchart applications is also taken to make it possible to use Grafchart for hard real-time control. High execution rates often reveal concurrency issues and thus execution concurrency has also been investigated.

Access to more data from industrial devices and software can be used to optimize production. Architectures for factory integration have been considered as this is the foundation to connect all devices and thus address the challenge of integrating and utilizing devices and software. Service Oriented Architecture (SOA) is a flexible software design methodology widely used in IT systems and for business processes. SOA service orchestration is brought to industrial automation by integrating support for both Devices Profile for Web Services (DPWS) and OPC Unified Architecture (OPC UA) in JGrafchart. Looking further, SOA 2.0 is event driven and features extremely loose coupling between components. An architecture based on SOA 2.0 where it is easy to integrate any device or software, in particular legacy devices with limited knowledge and capabilities, has been developed with focus on service choreography in industrial manufacturing. Another step toward real-time execution of Grafchart applications is integrated support for the high performance communication protocol LabComm. Additionally, it is investigated how Grafchart can be connected to Functional Mock-up Interface (FMI) for co-simulation to further address the shorter time to market trend by introducing simulation support.

The PID controller is the most common controller for industrial automation. A PID implementation has been added to a Grafchart library and a flaw with the PID algorithm has been discovered. The problem occurs for PID controllers with a derivative part when the process value saturates. The derivative part then backs off which leads to undesired changes in the control signal. This issue has been analyzed and a solution to the problem is proposed.

Acknowledgments

First of all I would like to thank my supervisor Charlotta Johnsson for always being enthusiastic about my work, and for helping me outline and proofreading this thesis.

I would also like to thank my co-supervisor Karl-Erik Årzén for helping me understand the thoughts behind the previous work and the JGrafchart implementation and for proofreading this thesis.

I also want to thank the Department of Automatic Control for hiring me and supporting me as a Ph.D. student, and for helping me grow both as a researcher and as a person.

My thanks also go to Lisa Ollinger, Tobias Gerber, and Johan Hagsund for interesting and fun collaboration.

I would also like to thank the users of JGrafchart for all the valuable discussions and feedback regarding the tool.

I am also very grateful to the developers of all the free tools that I have used both to do the work and to write this thesis. I am especially grateful for Eclipse, Apache Ant, reStructuredText (Docutils) [1], and LaTeX.

I would also like to thank Leif Andersson for sharing his LaTeX expertise and for helping me resolve some intricate LaTeX issues.

Special thanks go to my family and friends. In particular I would like to thank my mother, Iréne, for always believing in me.

Finally, my very special thanks go to my loving wife, Lisa, for being a wonderful person and for proofreading this thesis.

Abbreviations

API	Application Programming Interface, a software interface for an application.
AST	Abstract Syntax Tree, a common way for compilers to represent applications.
BPMN	Business Process Model and Notation, a graphical language to model business processes.
BV	Basic Version, one of the two Grafchart versions.
DPWS	Devices Profile for Web Services, a minimal set of mandatory web service extensions targeted for resource constrained devices.
EDA	Event-driven Architecture, a flexible and extremely loosely coupled software architecture.
EIP	Enterprise Integration Patterns, best practices for enterprise integration.
ESB	Enterprise Service Bus, a component for message routing to distributed applications.
FBD	Function Block Diagram, one of the graphical IEC 61131-3 standard programming languages.
FC	Function Chart, an SFC or Grafchart application.
FMI	Functional Mock-up Interface, an interface standard for simulations.
FMU	Functional Mock-up Unit, an exported FMI model.
FSM	Finite State Machine, a modeling language for machines with a finite number of states.
HLV	High-Level Version, one of the two Grafchart versions.

I/O	Inputs and Outputs, how applications interact with the external environment.
IDE	Integrated Development Environment, a software in which applications can be written, compiled, executed, and debugged.
IEC	International Electrotechnical Commission, a standards organization.
ISA	The International Society of Automation, a standards organization.
ISO	International Organization for Standardization, a standards organization.
IL	Instruction List, one of the textual IEC 61131-3 standard programming languages.
LD	Ladder Diagram, one of the graphical IEC 61131-3 standard programming languages.
LISA	Line Information System Architecture, a research project and an event driven architecture.
OPC	The classic interoperability standard for industrial automation.
OPC AE	OPC Alarm and Events, a classic OPC standard for alarms and events.
OPC DA	OPC Data Access, a classic OPC standard to read and write data.
OPC HDA	OPC Historical Data Access, a classic OPC standard to read historical data.
OPC UA	OPC Unified Architecture, an interoperability standard for industrial automation systems.
PLC	Programmable Logic Controller, a control system used for industrial automation.
PN	Petri Nets, a mathematical language for system modeling.
PtP	Point-to-Point, the traditional integration approach.
ReRAGs	Rewritable Reference Attribute Grammars, a declarative way to implement compiler semantics.
SFC	Sequential Function Charts, one of the graphical IEC 61131-3 standard programming languages.
SLOC	Source Lines Of Code, a metric for the size of software programs.
SOA	Service Oriented Architecture, a flexible and loosely coupled software architecture.

SOA-AT	SOA in Automation Technologies, SOA used for automation.
ST	Structured Text, one of the textual IEC 61131-3 standard programming languages.
UML	Unified Modeling Language, a modeling language for software engineering.
WSDL	Web Services Description Language, a language to define the interface of web services. Also used to refer to the interface of a particular web service.
XML	eXtensible Markup Language, a textual data format for structured data.

Contents

1

Introduction

One definition of automation is "the creation and application of technology to monitor and control the production and delivery of products and services" [2]. Some examples where automation is used are for production of cars, consumer electronics, medicine, plastics, paper, gasoline, and chemicals. They are all produced in factories where the production is highly automated. To produce with less manual labor often means cheaper production, higher production rate, and a more stable product quality. Companies need a high level of automation to be competitive and stay in business. A company that can automate more or do it better than its competitors has a competitive advantage. Even small improvements can potentially generate or save a considerable amount of money. To improve existing automation and increase the level of automation is thus always a relevant topic.

One current market trend is that the products are expected to be more customizable. Take buying a new car for example. A few years back the only options were the brand, the model, one of a few model configurations, and the color. Now it is possible to choose freely among all the available options and colors and get a completely customized configuration. This is often cheaper than buying one of the default configurations of the product [3]. You can skip all the options that you do not care about and do not have to buy the supreme configuration to get the less common option that you want. For the manufacturer this means that more flexible production systems are needed. However, the control systems and languages used for control were developed with a more static production in mind.

At the same time there is an increasing demand for shorter time to market, that is, to set up and make changes to the production systems faster. New environmental impact legislation also introduce new boundaries for what is allowed which in turn require changes to the production systems. The earlier a production system can be up and running, the earlier it is possible to make money from the production. Similarly, if a change can be applied faster there are less production losses and it is possible to benefit from the change earlier. Considering the powerful global competitive pressure this is important to stay in business.

Based on these market trends the following challenges have been identified:

1. Flexible production systems
2. Integration and utilization of devices and software

With a flexible production system it is easier to produce customizable products and to reduce the effort needed to add new options. Flexible languages and tools are required to realize flexible production systems.

The smart factory is a concept which refers to future factories where all devices and software are well integrated and utilized. The benefit of this is that the data from devices and software is readily available and can be used to improve or optimize the production. To make it possible to access the data, all devices and software must first be integrated. To accomplish this, a flexible architecture is needed where it is easy to integrate any application or device. Factories typically contain a wide range of devices that are from different eras and based on different technologies. Some devices are from when the factory was built and devices have then been added as part of continuous improvements of for example quality, reliability, or efficiency. It is thus particularly important that legacy devices regardless of age or capabilities can be integrated in such an architecture.

1.1 Methodology

In this thesis the inductive research methodology has primarily been used. The starting point has been the observation of current trends in industrial automation. Applications from both process automation (continuous, for example chemical plants and paper mills), manufacturing (discrete, for example car factories), and robotics (discrete, for example assembly and packaging) have been considered. Based on the current trends, challenges related to control languages have been identified and the research question has been how these challenges can be addressed through improvements and use of a sequential control language.

1.2 Publications

This thesis is based primarily on the following publications:

- A. Theorin, K.-E. Årzén, and C. Johnsson. “Rewriting JGrafchart with Rewritable Reference Attribute Grammars”. In: *Industrial Track of Software Language Engineering 2012*. Dresden, Germany, 2012.
- A. Theorin, L. Ollinger, and C. Johnsson. “Service-oriented process control with Grafchart and the Devices Profile for Web Services”. In: T. Borangiu et al. (Eds.). *Proceedings of the 14th IFAC Symposium on Information Control Problems in Manufacturing (INCOM’12)*. Elsevier Ltd, Bucharest, Romania, 2012, pp. 799–804. DOI: 10.3182/20120523-3-RO-2023.00131.

- A. Theorin, L. Ollinger, and C. Johnsson. “Service-oriented process control with Grafchart and the Devices Profile for Web Services”. In: T. Borangiu et al. (Eds.). *Service Orientation in Holonic and Multi Agent Manufacturing and Robotics*. Vol. 472. Studies in Computational Intelligence. Springer Berlin Heidelberg, 2013, pp. 213–228. ISBN: 978-3-642-35851-7. DOI: 10.1007/978-3-642-35852-4_14. URL: http://dx.doi.org/10.1007/978-3-642-35852-4_14.

- A. Theorin and C. Johnsson. “Polymorphism for state machines”. In: *ISA Automation Week 2012*. Orlando, FL. USA, 2012.

- T. Gerber, A. Theorin, and C. Johnsson. “Towards a seamless integration between process modeling descriptions at business and production levels - work in progress”. In: T. Borangiu et al. (Eds.). *Proceedings of the 14th IFAC Symposium on Information Control Problems in Manufacturing (INCOM’12)*. Elsevier Ltd, Bucharest, Romania, 2012, pp. 1537–1542. DOI: 10.3182/20120523-3-RO-2023.00309.

- T. Gerber, A. Theorin, and C. Johnsson. “Towards a seamless integration between process modeling descriptions at business and production levels: work in progress”. English. *Journal of Intelligent Manufacturing* (2013), pp. 1–11. ISSN: 0956-5515. DOI: 10.1007/s10845-013-0754-x. URL: http://dx.doi.org/10.1007/s10845-013-0754-x.

- L. Ollinger, D. Zühlke, A. Theorin, and C. Johnsson. “A reference architecture for service-oriented control procedures and its implementation with SysML and Grafchart”. In: *18th IEEE International Conference on Emerging Technologies and Factory Automation*. Cagliari, Italy, 2013.

- A. Theorin and C. Johnsson. “An interactive PID learning module for educational purposes”. In: *Proceedings of the 19th IFAC World Congress (IFAC’14), Cape Town, South Africa*. 2014.

- A. Theorin and C. Johnsson. “On extending jgrafchart with support for FMI for co-simulation”. In: *10th International Modelica Conference*. Lund, Sweden, 2014.

- A. Theorin, J. Hagsund, and C. Johnsson. “Service orchestration with OPC UA in a graphical control language”. In: *19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA’2014)*. Barcelona, Spain, 2014.

- A. Theorin, K. Bengtsson, J. Provost, M. Lieder, C. Johnsson, T. Lundholm, and B. Lennartsson. “An event-driven manufacturing information system architecture”. In: *In submission*.

- A. Theorin and T. Hägglund. “Derivative backoff: a process value saturation problem for PID controllers”. *In journal submission* (2014).

1.3 Contributions

The main contributions of this thesis are:

- A way to do service orchestration in the smart factory using the service technologies Devices Profile for Web Services (DPWS) and OPC Unified Architecture (OPC UA).
- Establishing a foundation for service choreography in the smart factory with Service Oriented Architecture (SOA) 2.0.
- Demonstrating the applicability of state of the art compiler techniques for an industrial based sequential control language.
- A step toward real-time execution of a flexible industrial based sequential control language.
- The discovery and solution of a PID algorithm issue regarding the derivative part when the process value saturates.

1.4 Research Projects

The research presented in this thesis has been carried out in three research projects: Line Information System Architecture (LISA), LCCC, and ELLIIT.

The aim of the LISA project is to support knowledge-based competitive production through the development of methods for control, optimization, and maintenance of sustainable discrete manufacturing processes. The LISA project is funded by VINNOVA, the Swedish government agency for innovation, and is part of the FFI initiative Sustainable Production Technology.

The aim of the LCCC Linnaeus Center is to develop theory, methods, and tools for control of large scale engineering systems. Our society depends on flexible infrastructure for industrial production, energy supply, and communication systems. This requires research and innovations on control of complex systems. Many challenges are common for a wide range of application areas and need to be addressed with a combination of competences from control, communications, and computer science. LCCC is facing these challenges. LCCC is funded by the Swedish Research Council (VR).

The ELLIIT excellence center is a network organization for information and communication technology. Its objective is scientific excellence in combination with industrial relevance and impact. It is organized within the Swedish government's strategic research support initiative and is funded by VINNOVA and the Swedish Research Council (VR).

Financial support from VINNOVA and VR through the research projects LISA, LCCC, and ELLIIT is gratefully acknowledged. The research platform

Figure 1.1 Logos for the research projects.

SmartFactory^{KL} [4] which has provided collaboration and evaluation opportunities is also acknowledged.

1.5 Thesis Outline

The thesis is divided into chapters where the first chapters present the background for the work and the following chapters describe the contributions in detail.

Chapter 2–4 are background chapters. In Chapter 2 the industrial automation field and Industry 4.0 are presented. In Chapter 3 automation languages related to this thesis are presented and graphical programming is compared to textual programming. Chapter 4 describes the sequential control language Grafchart and its implementation JGrafchart in detail.

In Chapter 5 improvements to the Grafchart language and the Grafchart implementation JGrafchart are presented. In particular, new language constructs are described, modern compiler techniques are evaluated for JGrafchart, and real-time execution and execution concurrency for Grafchart applications are addressed. In Chapter 6 factory integration is addressed by considering different architectures. Integrated JGrafchart support for the SOA service technologies DPWS and OPC UA, as well as the high performance communication protocol LabComm are presented. The SOA 2.0 architecture LISA as well as an investigation of how JGrafchart can be connected to Functional Mock-up Interface (FMI) for co-simulation are also presented. In Chapter 7 a newly discovered problem with the PID algorithm is described in detail and a solution is proposed. Finally, Chapter 8 contains a summary and future work.

2

Industrial Automation

This chapter gives an overview of the industrial automation field and how it relates to other fields.

2.1 Terminology

Industrial automation Industrial automation is a wide term which includes all sciences and technologies used to implement an industrial production system. It includes for example mechanical systems, chemical reactions, sensor and actuator technology, order handling, and logistics. In this thesis the term industrial automation is used to refer to this context and the focus is on control systems, control languages, control algorithms, and control applications as well as their interaction with other systems.

Field device The physical sensors and actuators used in a production machine are known as field devices. They are the interface between the physical world and the control systems.

Embedded system Many stand-alone embedded devices such as smart phones or traffic lights are controlled internally by a dedicated microcontroller. This is known as an embedded system.

Cyber-physical systems Cyber-physical systems are considered the next generation of embedded systems. Instead of focusing on a single stand-alone embedded device, a network of distributed embedded devices with actuators and sensors and with computing and communication capabilities is considered [5]. One example is distributed control applications where the control is implemented through collaboration between embedded devices.

Internet of Things The Internet of Things is a concept where embedded devices are uniquely identified and can communicate with each other over the Internet [6].

2.2 Functional Hierarchy

IEC 62264 is an international standard based on ISA95 [7] which describes integration of enterprise and control systems. Part 1 of the standard describes models and terminology [8]. The standard classifies the production tasks in levels, see Figure 2.1.

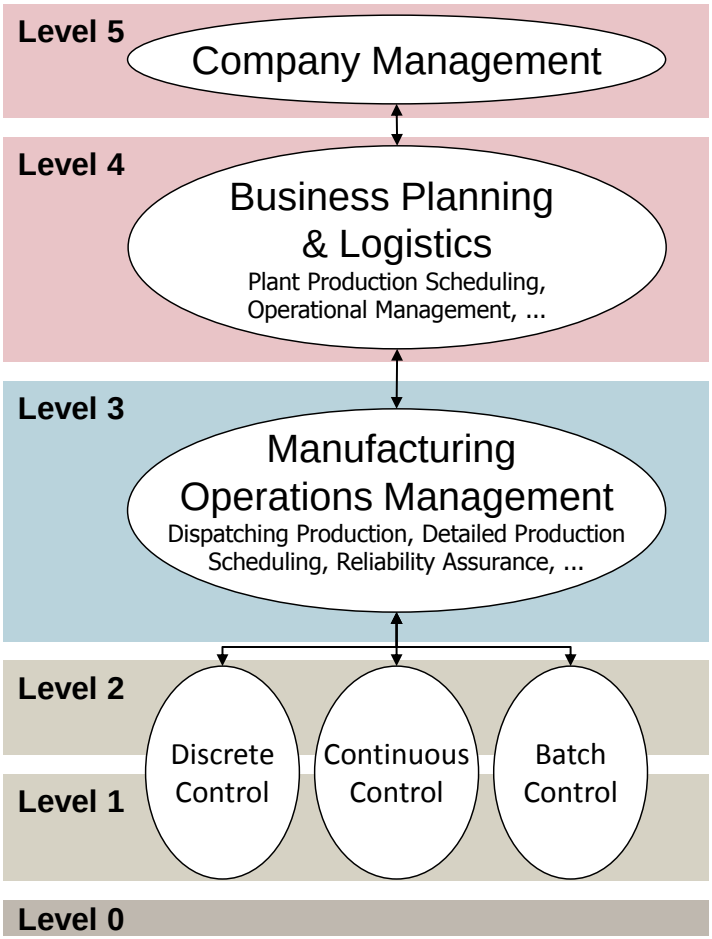


Figure 2.1 Functional hierarchy of production according to IEC 62264-1.

Level 0 is the physical production process.

Level 1 and 2 contain tasks related to sensing and actuation of the physical process, that is, real-time and supervisory control. This is also referred to as the device level. The time frame ranges from hours to milliseconds. The implementation

of the tasks on this level should ensure stable product quality, production reliability, and energy efficiency.

Level 3 contains tasks such as detailed scheduling, data collection, performance, and dispatching. The time frame ranges from work shifts to seconds. The implementation of the tasks on this level should ensure high utilization of the production equipment.

Level 4 contains tasks related to business processes such as basic production planning, delivery, and inventory handling. This is also referred to as the business level. The time frame ranges from months to work shifts.

Level 5 contains tasks related to company management such as order handling and customer relations. The time frame ranges from years to months.

The communication on level 2 and below is often integrated and data can be fetched at any time. Communication between the upper and the lower levels on the other hand is in many cases manual and thus data is not readily available. Integrated support for communication between all levels is known as vertical integration. With vertical integration it is possible to access and utilize all existing data at all levels. This is important as more data means more opportunities for production optimization.

2.3 Field Buses

To support hard real-time control, field device communication must be deterministic and reliable. The traditional physical architecture of control systems is rigid, with each field device connected to one field bus which in turn is connected to one controller, for example a Programmable Logic Controller (PLC). Industrial Ethernet is the next generation of field buses which are based on Ethernet technology. When the Ethernet infrastructure also includes the field devices there is an opportunity to use Ethernet for control. As the communication with field devices no longer has to go through a specific controller, the architectural flexibility can be increased by decoupling the physical and functional hierarchy of field devices. Hence loose physical coupling can be attained, which simplifies vertical integration.

2.4 Characteristics

Control logic in automation is almost exclusively implemented on computers. Like ordinary computer programs are implemented in Java or C++, automation is implemented in the languages defined in the international standard IEC 61131-3 [9] for PLCs. Practically all languages for ordinary computer programs are textual. The programs are written in plain text and are compiled to executable binaries. To troubleshoot and inspect what is going on, a debugging environment is needed.

Implementing ordinary consumer applications for conventional computers, smart phones, or tablets is similar to implementing automation in some aspects and

completely different in other, see Figure 2.2. One similarity is that the applications are executed on similar computer hardware and hardware architecture. Some characteristics are also shared with embedded systems. An embedded system is closer to the hardware and is often required to execute its applications in real-time, for example to respond to an event within a given time to behave correctly.

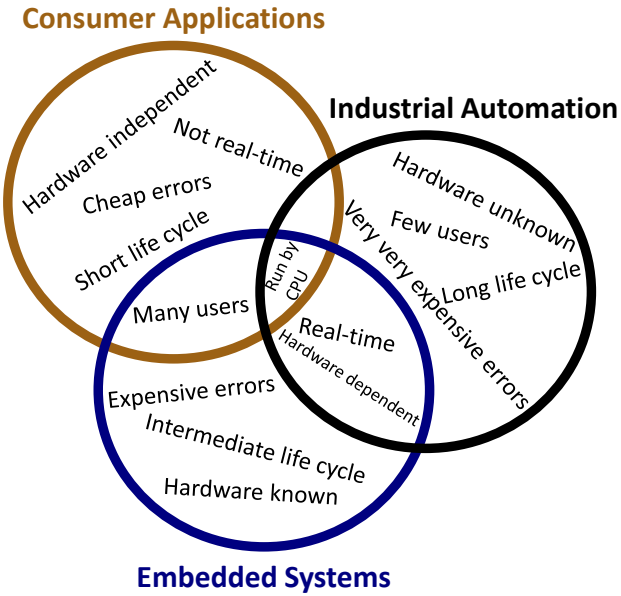


Figure 2.2 Comparison of implementation aspects for ordinary consumer applications, embedded systems, and automation.

The life cycle for consumer applications is roughly as long as the time between the releases and can typically be measured in months. The life cycle for an embedded system is roughly the life span of the product, typically a few years. The life cycle for automation is roughly how long the production machine is running, which is for as long as it is profitable to run it. For example, paper machine 1 at Stora Enso Hylte Mill was shut down after operating for 41 years due to lower market demand [10]. There is a demand for spare parts during the entire lifetime of production machines and control system manufacturers need to make long term commitments to attract customers. For example, ABB guarantees that they will actively produce spare parts for the previous generation for at least 10 years [11].

For consumer applications the underlying hardware does not have to be considered, abstraction layers between the application and the hardware take care of this. When implementing an embedded system the target hardware is known and it is sufficient to make the software work on this particular hardware. In automation there is

much more uncertainty. For example, it is impossible to know if the hardware will need to be changed in 20 years due to a lack of spare parts.

Another difference is the amount of hardware used. Embedded systems are only concerned with its particular hardware while the primary objective for automation is to use sensors and actuators to get the machines to behave properly. Typically many different types of sensors and actuators are required and they all need to be configured properly.

Since there is a considerable cost to start up a production machine, shutting it down is avoided as far as possible. Changes may be applied during maintenance stops or, if possible, on the fly. This is one reason why the languages used for automation differ from other fields. In other fields, debugging mode is only used during development. In automation a mode close to debugging mode is always used. It is not possible to set breakpoints and step through the code, but execution details such as variable values are available during execution. This is needed to be able to troubleshoot odd behavior without shutting down the production machine.

Considering the cost of errors, take for example a race condition that causes a crash one time out of ten thousand and is otherwise harmless. For a consumer application or an embedded system it is enough to restart the application or the device. For a consumer application a fix can simply be rolled out with the next version. Embedded systems are harder to update, often special equipment is required. If an application stops in industrial automation, so does the production. Many production machines take a long time to start up and they may consume raw material without producing sellable products during startup. Also, many production machines are hazardous and an error in the program can cause worker injuries.

2.5 Control Loops

An important part of industrial automation are the feedback loops which are executed in real-time to give production processes desired behavior. For example, the control loops handle disturbances and ensure stable product quality. Figure 2.3 shows an overview of a simple feedback loop. The input to the controller is the control error, $e(t)$, which is the difference between the desired (reference/set-point) process state, $y_{sp}(t)$, and the measured process state, $y(t)$ is

$$e(t) = y_{sp}(t) - y(t) \tag{2.1}$$

The output of the controller is the manipulated variable (control signal), $u(t)$.

The PID controller is by far the most commonly used controller in industry. There are billions of control loops [12] and the PID controller is used for more than 95% of all control loops [13].

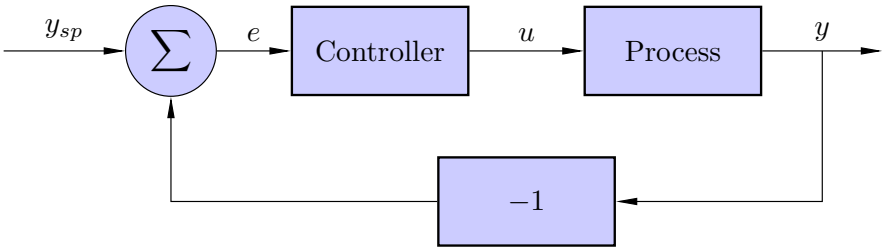


Figure 2.3 A feedback loop where a controller is used to control the process by considering the control error.

2.6 Service Oriented Architecture

As new functionality and systems are added they need to be rapidly integrated with existing systems. The traditional integration approach is to connect applications on a Point-to-Point (PtP) basis with the client/server pattern. The pattern mandates that the client knows about the server and that the server knows about the client. The number of connections in a fully connected network increases quadratically with the number of applications. This is known as "spaghetti integration" and makes the system rigid and hard to maintain [14]. Each time an application is added, all other applications must also be updated.

It is also common that applications are only able to communicate through proprietary or limited protocols and applications may require external message translators to communicate with each other at all. This is for example the normal case for communication between PLCs from different vendors.

The PtP approach is almost useless in supporting the expected business requirements [15]. Yet industry has been slow to migrate to new approaches, mainly due to the cost to replace their established legacy systems based on PtP [14]. However, migration has been significantly accelerated by the advent of SOA [14].

SOA is a component-based distributed software architecture. Each component encapsulates a specific functionality and is called a service. Services are unassociated, loosely coupled, and self-contained. Preferably they are also discoverable. Services are described with metadata so that they can be both language and platform independent. To write an application which combines SOA services is known as service orchestration. SOA enables a high degree of reusability and flexibility. To use SOA for automation has potential to significantly reduce the integration effort.

SOA is widely used for business processes [16, 17] on level 4 of Figure 2.1. It has also been recognized for use in automation in several research projects [18, 19, 20].

The term SOA in Automation Technologies (SOA-AT) is used to distinguish SOA used for business processes from SOA used in automation since they differ in many ways [21]. One difference is the execution environment. SOA for business

processes is implemented on ordinary computers with practically unlimited memory and processing power. In SOA-AT services are running on resource constrained embedded devices with little memory and processing power. Only a minimal set of features can then be supported. Another difference is that in SOA-AT the service execution timing is more important.

In SOA-AT the functionality of a field device is encapsulated as a service and the control application is implemented as orchestration of these services.

Even though SOA conceptually offers loose coupling and is intended to be distributed, service orchestration is typically done centrally with the orchestrator taking control of the involved services.

SOA 2.0 [22], also known as advanced SOA or event-driven SOA, is the next generation of SOA which is inspired by Event-Driven Architecture (EDA) [23] and is more focused on events. SOA 2.0 enables service choreography, where each service reacts to published events on its own, rather than being requested to do so by a central orchestrator.

EDA is extremely loosely coupled and distributed by design. The creator of an event only needs to know that the event occurred, it does not need to know anything about who is interested in the event or how it will be processed [23]. Event data should be immutable and it is then always (thread-)safe to pass events around both within and between applications. Thus applications turn from being synchronized and blocking to being asynchronous and non-blocking [24].

2.7 Industry 4.0

Industry 4.0 [25] is a term that refers to the fourth industrial revolution. Industrie 4.0 is also a German strategic research project with a time frame of more than 20 years [26, 27]. The first industrial revolution began in the late 18th century. It introduced mechanical automation, powered by steam and water, to replace manual labor. The second industrial revolution began in the early 20th century. It introduced mass production through the use of electricity. The third industrial revolution began in the second half of the 20th century through digitalization, that is, use of electronics and IT for further automatization. This was also when industrial robots were introduced.

Industry 4.0 envisions a factory where everything is connected, similar to Internet of Things but for producing industries. Such a factory is often referred to as a smart factory. Industry 4.0 includes moving from isolated embedded systems to networked cyber-physical systems, which are easy to use and compose. Software and software integration play important roles. With everything connected there is an opportunity for improved production profitability [28].

It must be a low effort to integrate and use the many cyber-physical systems in the smart factory. Loosely coupled components and an architecture which features loose coupling will be needed to tackle the increased complexity. Hence SOA is recognized as a key enabler for Industry 4.0 [26].

3

Automation Languages

This chapter describes languages used for industrial automation, languages similar or related to Grafchart, and other languages of relevance for this work.

3.1 Graphical Programming Languages in PLCs

The IEC 61131-3 standard defines five programming languages for PLCs [9]: the three graphical languages Ladder Diagram (LD), Function Block Diagram (FBD), and Sequential Function Charts (SFC) and the two textual languages Structured Text (ST) and Instruction List (IL). These languages can also be combined.

The code for the textual languages is executed line by line from the top down. On the other hand, the code for the graphical languages consists of graphical elements which are connected to make up the program. This can be a more suitable way to implement applications. Specifically, it suits automation very well.

Function Blocks

Function blocks are used to encapsulate reusable functions in IEC 61131-3. A function block may have input variables, output variables, and internal variables. They are called similar to Java methods but may have internal states and must be instantiated similar to Java classes.

Ladder Diagram

LD is a replacement for implementation of relay logic with physical relays. Engineers thinking in terms of relay logic draw their applications as LD diagrams, see Figure 3.1. Instead of then wiring relays the diagrams can be executed directly in a PLC. The vertical lines on the sides are the power rails. The power of the diagram flows from the left power rail to the right power rail. Coils are used to assign outputs and are drawn as a pair of round brackets, for example `D` in Figure 3.1. A coil's output is true whenever there is power flow through the coil. Contacts are used to implement the logic and are drawn as a pair of vertical lines, for example `A` in Figure 3.1. Power can flow through a contact when its input variable is true. A diagonal

line between the vertical lines means that the contact is inverted, see for example `C` in Figure 3.1. Power can flow through an inverted contact when its input variable is false.

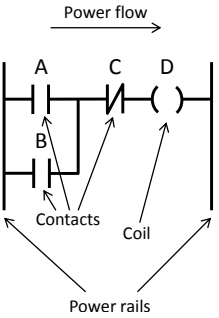


Figure 3.1 An LD diagram equivalent to `D = (A | B) & !C`.

Function Block Diagram

FBDs consist of function block instances whose inputs and outputs are connected graphically, see Figure 3.2. This way it is easier to get an overview of the application.

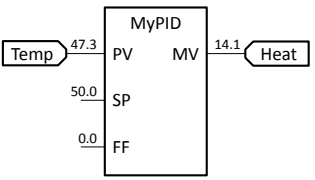


Figure 3.2 An FBD implementation of a PID control loop. The inputs to the function block instance `MyPID` are the process value `Temp` (PV), a constant set point of `50.0` (SP), and no feed forward (FF). The manipulated variable (MV) of `MyPID` is connected to the output `Heat`.

Sequential Function Charts

SFC is based on GRAFCET, which is a standardized specification language to describe the functional behavior of the sequential part of a control system. The standard that describes GRAFCET is IEC 60848 [29].

SFC consists of steps representing states, and transitions representing the change of state. It is supported by most industrial automation systems, for example 800xA by ABB, SIMANTIC S7 by Siemens, RSLogix 5000 by Rockwell Automation, DeltaV by Emerson, and CENTUM CS by Yokogawa. SFC is used to

implement sequential, parallel, and general state-transition oriented applications. It is hard to get a non-trivial state machine right in a textual language and once written it is practically impossible to get an overview of the state machine. Typically, it is first drawn on paper and then translated into textual code. Similarly, to understand how things are connected the code is translated into a graphical representation. A textual language also requires that all steps are named, otherwise they cannot be referred to which is required to connect them to transitions. Working directly with a graphical representation is more intuitive.

Figure 3.3 shows a comparison of a small application implemented both in SFC and a minimal textual language. In the SFC application it is easier to follow the flow and get an overview of the application. The textual implementation is more compact but it is harder to get an overview.

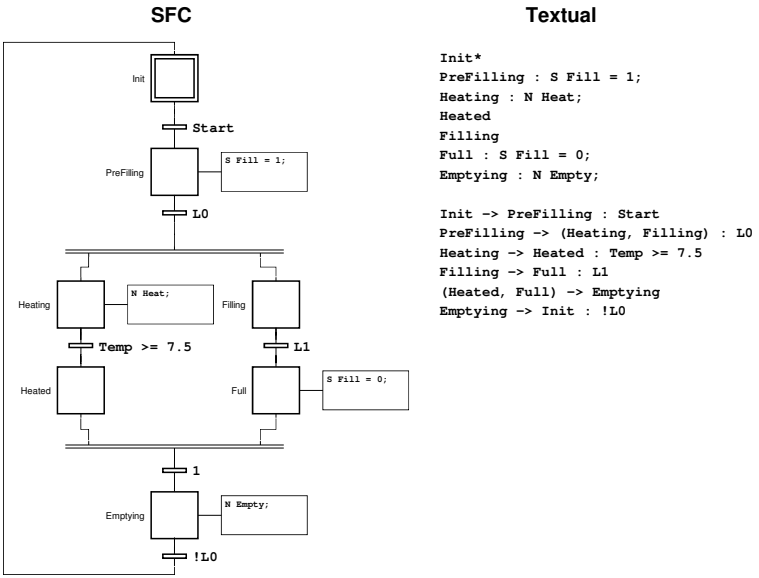


Figure 3.3 An application implemented in both SFC and a minimal textual language.

Visualization

Visualization of the current execution state can be done for all these graphical languages and makes it easier to see what is happening. In SFC the current state can be highlighted. In LD connections can be highlighted depending on if they are true or false. In FBD the values of function block inputs and outputs can be written next to the connections and, as for LD, Boolean connections can be highlighted depending on their value.

3.2 Petri Nets

Petri nets (PN) is a mathematical language for system modeling [30]. It consists of places and transitions as well as directed arcs between places and transitions. The places contain tokens which model a property of the current system state. All tokens together represent the whole system's state and is called a marking. The initial marking corresponds to the system's initial state.

The marking can be changed by firing transitions, one at a time. The places with an arc leading to the transition are called the transition's input places and the places with an arc leading from the transition are called the transition's output places. A transition may fire if there is at least one token in each input place. When a transition fires, a token in each input place is consumed and a token is produced in each output place.

With a Petri net model it is possible to analyze a system to determine if it is for example deadlock free, live, bounded, or if a certain marking is reachable. Deadlock free means that it is not possible to reach a marking where no transition is fireable. A transition is live if there from each valid marking exists a sequence of firings which includes the transition. A Petri net is live if all its transitions are live. A place is bounded if there is an upper limit on the number of tokens it may contain. A Petri net is bounded if all its places are bounded. A marking is reachable if there exists a sequence of firings that brings the initial marking to that marking.

Properties of the Petri net have a corresponding meaning for the modeled system. It can for example be guaranteed that a forbidden state cannot occur if its corresponding marking is not reachable. Another example is that a deadlock free Petri net guarantees that the system will not freeze.

An example Petri net is shown in Figure 3.4. The modeled system is an assembly station where the tokens represent workers. Workers in place p_1 are unallocated, workers in place p_2 are producing part A, workers in place p_3 are producing part B, and workers in place p_4 are assembling part A and B. An unallocated worker may start producing part A, modeled by transition t_1, or part B, modeled by transition t_2. When there is both a part A and a part B available they can be put together and then one worker assembles the parts and the other becomes unallocated, modeled by transition t_3. Finally, when the assembling is complete, that worker also becomes unallocated, modeled by transition t_4. With the marking in Figure 3.4 there is one worker producing part A, one worker producing part B, and one worker assembling. Next, either another assembly can begin, that is, transition t_3 is fired, or the current assembling is finished, that is, transition t_4 is fired as shown in Figure 3.5. By analyzing Petri net properties it can be found that the net is not deadlock free which corresponds to stalled production. This happens for example for the firing sequence $\{t_3, t_4, t_1, t_1, t_1\}$ applied to the marking in Figure 3.5 which results in the marking in Figure 3.6.

There exist many extensions to Petri nets. One example is colored Petri nets where each token may contain individual data (its "color"). The data may be used in

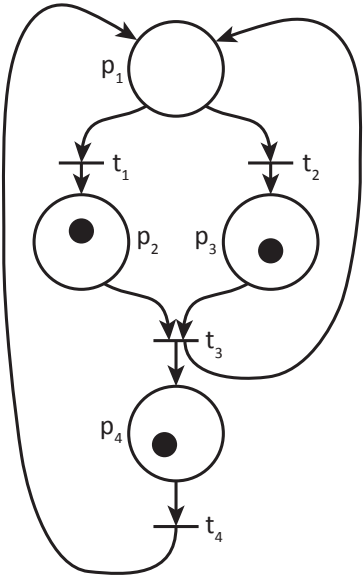


Figure 3.4 A Petri net consisting of the places p_1, p_2, p_3, and p_4 and the transitions t_1, t_2, t_3, and t_4. In the current marking there is one token in p_2, one in p_3, and one in p_4.

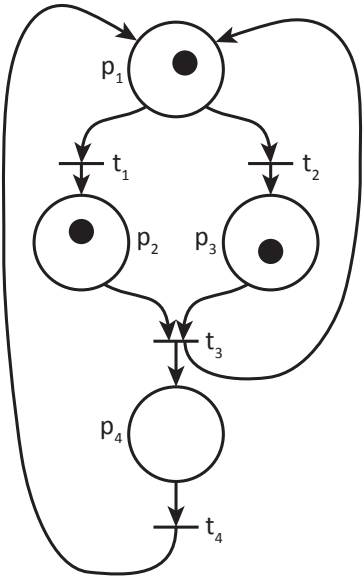


Figure 3.5 The Petri net in Figure 3.4 after t_4 has fired.

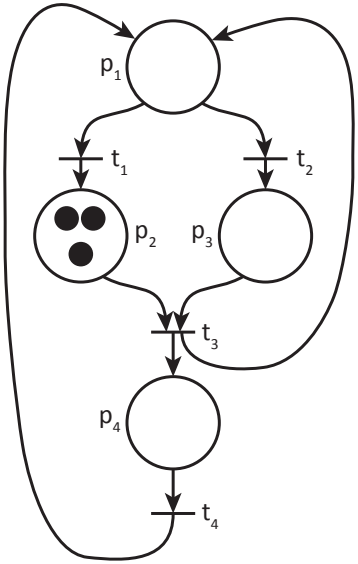


Figure 3.6 The Petri net in Figure 3.5 after firing sequence $\{t_3, t_4, t_1, t_1, t_1\}$, resulting in a deadlock.

guard conditions in transitions which means that the transition is only fireable when the condition is true. The data may also be changed for example when a transition is fired or when the token enters a place. Another example is timed Petri nets in which each transition or place may have timing constraints, for example the transition may not fire until it has been enabled for 3 seconds.

3.3 Finite State Machines

A finite state machine (FSM) [31], also known as finite state automaton or simply state machine or automaton, consists of a set of possible states, inputs, and outputs. FSMs are widely used for design of digital circuits and implementation of computer applications with states. An FSM always has exactly one current state and can thus be seen as a Petri net restricted to one token. For each state the next state and the outputs are pre-specified, possibly depending on the input values. The two classic FSMs are the Mealy machine and the Moore machine. For a Moore machine the outputs are determined by the current state only. For a Mealy machine the outputs are determined by both the current state and the current input values.

As an example, consider a state machine with one digital input, `i`, and one digital output, `o`. The output value should be 1 when the two most recent input values are 1 and in all other cases it should be 0. This can be implemented with a Moore machine represented either graphically, see Figure 3.7, or as a table, see Table 3.1.

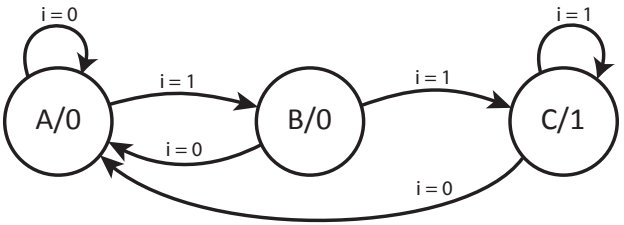


Figure 3.7 A Moore machine represented graphically.

Table 3.1 A Moore machine represented with a state transition table.

Current State/Output	Next State, Input = 0	Next State, Input = 1
A/0	A	B
B/0	A	C
C/1	A	C

3.4 Statecharts

Statecharts [32] are known by many names such as Harel statechart, state diagram, Unified Modeling Language (UML) state machine, and UML statechart and are used to describe the behavior of systems. It is an extension to FSMs and consists of states and transitions. Statecharts are, like state machines, event driven. States may have entry and exit actions and transitions may have both guard conditions and associated actions. In statecharts there is also a concept of composite states, that is, states which have internal substates. A composite state may be divided into orthogonal regions, then one substate in each orthogonal region is active simultaneously. Finally, a composite state may contain a history pseudostate which stores the most recent substate history which can then later be restored. A shallow history pseudostate only stores the active substates (non-recursive) while a deep history pseudostate stores the complete configuration of a state (recursive).

3.5 Business Process Model and Notation

Business Process Model and Notation (BPMN) is a graphical language for business process modeling based on flowcharts and similar to UML activity diagrams [33]. Business processes are used to formalize how various business related tasks on level 4 in Figure 2.1 proceed, for example the sequence of steps involved in a product release.

BPMN consists of flow objects, connecting objects, swim lanes, and artifacts. The flow objects are: event, activity, and gateway. An event denotes that something happens, an activity denotes something that should be done, and a gateway is used to split and join paths. Connecting objects are used to connect the flow objects. Swim lanes is a way to detail the flow between the participants involved in the process. Finally, artifacts contain additional relevant information.

4

Grafchart

In this chapter the graphical programming language Grafchart and its implementations are described. First there is a short section about the history of Grafchart. Then the language is described in detail followed by implementation details for JGrafchart, the Java implementation of Grafchart. After this, a task is implemented in both Grafchart and SFC to exemplify the usefulness of some of Grafchart's and JGrafchart's additional features. Finally, a few implementation details for the old Grafchart implementation are mentioned.

4.1 History

Grafchart has been developed by the Department of Automatic Control at Lund University since 1991 [34]. It is based on SFC which is well-accepted by the automation community. Like SFC, Grafchart uses the state-transition paradigm. Grafchart has been inspired by ideas from statecharts, colored Petri nets, and ordinary object-oriented programming languages among others. The goal is to make it possible to write well structured, flexible, and maintainable applications. Grafchart was initially designed to be a language suited for batch control [35]. The main extensions facilitate exception handling and enable hierarchical structuring and code reuse. Applications written in Grafchart are often referred to as Function Charts (FC) or step sequences.

Grafchart has been shown to be a language well suited also for various other automation applications. The state-transition paradigm does not target any specific level in Figure 2.1 and thus Grafchart can be used to implement sequential applications on any level. It has been successfully used for a wide variety of applications, for example batch control, discrete control, and diagnosis. Grafchart also has potential for formal description, validation, and analysis [35].

There are two implementations of Grafchart, see Figure 4.1. The first one is also called Grafchart and was implemented in Gensym's knowledge-based system development environment G2 [36]. Here, this implementation is always referred to as G2Grafchart to distinguish it from the language itself. It was desirable to have

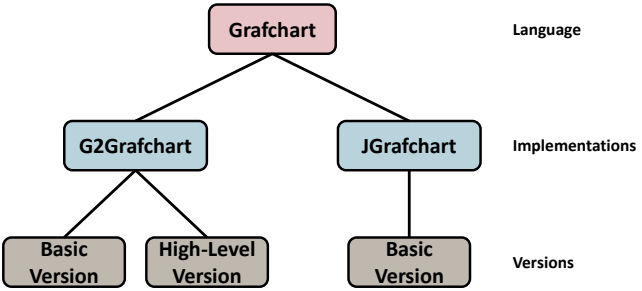


Figure 4.1 There are two implementation of Grafchart, namely G2Grafchart and JGrafchart. There are two versions of G2Grafchart and JGrafchart corresponds to the Basic Version.

a freely available tool built on an open platform. Hence, G2Grafchart is no longer developed or maintained.

The second implementation of Grafchart is written in Java and is called JGrafchart. It is actively developed and publicly available for free [34]. JGrafchart is a research tool that has been used in, for example, the EU/GROWTH project CHEM for control in process industry [37], the EU FP7 project ROSETTA [38] for robotic assembly [39], and several master's theses for modeling [40] and code generation [41, 42]. JGrafchart is also used for teaching, for example in laboratory exercises on sequential and batch control, as well as for control of real industrial processes [43].

4.2 Function Chart Elements

Grafchart will be introduced through examples to highlight the main ideas before describing the details.

Core Features

Grafchart's two main building blocks are steps and transitions. Steps represent possible application states and transitions represent the change of application state. Associated with the steps are actions which specify what to do. Associated with the transitions are Boolean guard conditions which specify when the application is allowed to change state.

A part of a running Grafchart application is shown in Figure 4.2. Two steps are connected by a transition and there are two variables `var` and `cond` where `cond` is set by another part of the application. When the first step is activated its S action is executed, meaning that `var` is set to 7. That the step is active is indicated by a token, drawn as a black dot. When `cond` becomes 4 the transition's guard condition becomes true. Then the first step is deactivated and the second step is activated, thus setting `var` to 12.

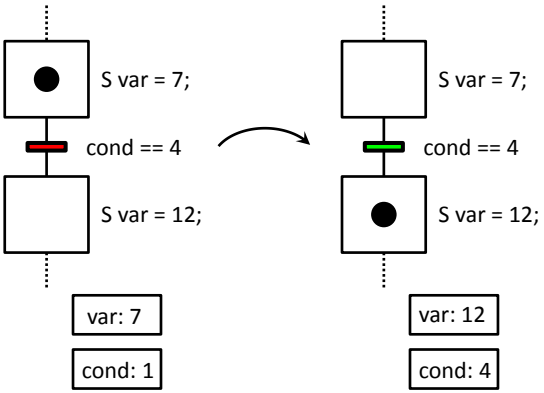


Figure 4.2 A part of a running Grafchart application showing the main building blocks of Grafchart, steps and transitions. The left part of the figure shows one application state and the right part shows a later application state.

Figure 4.3 shows how to express alternative and parallel paths. At any time only one alternative path may contain active steps. On the other hand, parallel paths are executed in parallel and always contain active steps at the same time. To create alternative paths a step is connected to several transitions. To create parallel paths a Parallel Split is used to split the execution. A Parallel Join is then used to merge the execution paths again.

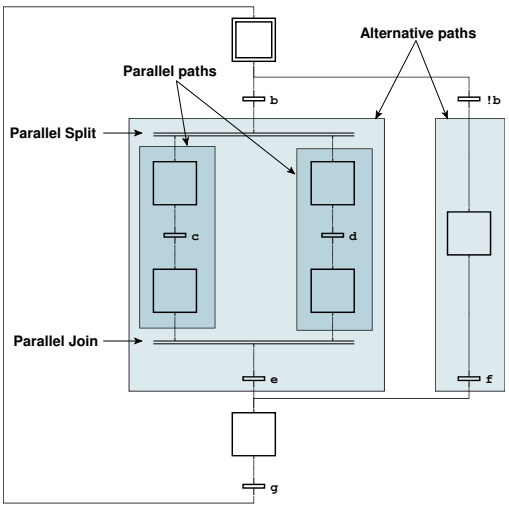


Figure 4.3 A Grafchart application showing how to express alternative and parallel paths.

Hierarchical Structuring

A common extension to SFC that is not included in the IEC 61131-3 standard is the Macro Step [44] which contains an internal FC, see Figure 4.4. It is, however, defined in GRAFCET [29] and is also available in Grafchart to make it possible to split up applications into understandable parts. When a Macro Step is activated its Enter Step is also activated. The Exit Step is the final step of a Macro Step. A Macro Step may only be deactivated when its Exit Step is active. When the Macro Step is deactivated the Exit Step is also deactivated.

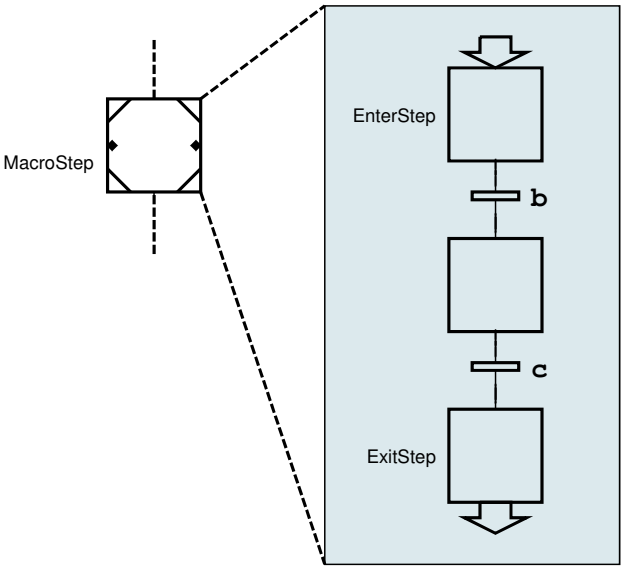


Figure 4.4 A Grafchart Macro Step and its internal FC.

Often, the same code is needed in more than one place. In Grafchart, reusable Procedures can be used to avoid redundant code. Like Macro Steps, Procedures also have an internal FC with an Enter Step and an Exit Step. In addition, Procedures may also return values and take parameters. Procedure Steps and Process Steps are used to call Procedures, see Figure 4.5. A Procedure Step may be deactivated once its procedure call reaches the Exit Step, that is, the same as for the Macro Step. The Process Step, on the other hand, spawns a new procedure call each time it is activated and does not wait for the call to reach the Exit Step before proceeding. Spawned procedure calls terminate automatically when the Exit Step is reached.

Yet another way to structure Grafchart applications is with a Workspace Object. It also has an internal FC but no Enter or Exit Steps. It can be used to represent objects with variables and methods, to group variables, or to structure applications.

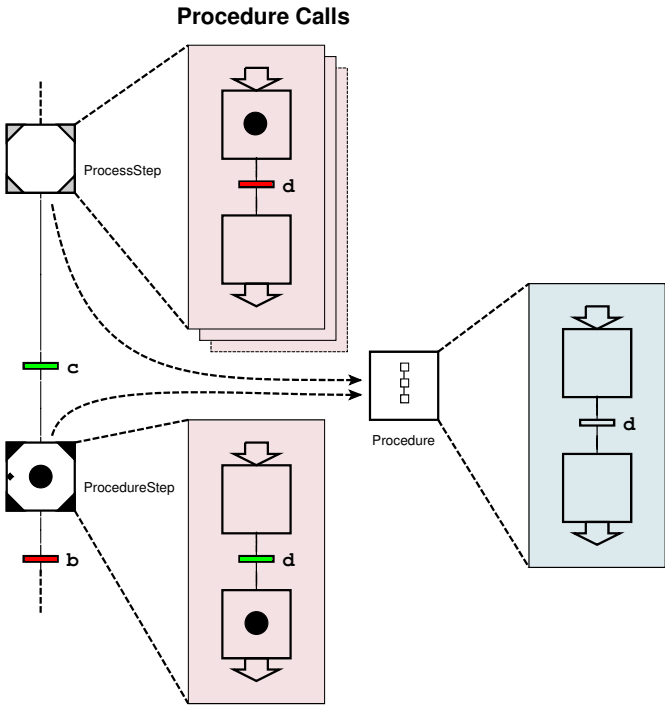


Figure 4.5 A Procedure can be thought of as a reusable Macro Step which can be called from Procedure Steps and Process Steps. Each Procedure Step and Process Step specifies which Procedure to call when activated. When `ProcedureStep` is active it will proceed when the procedure call has reached the Exit Step and `b` is true. When `ProcessStep` is active it will proceed as soon as `c` is true. The spawned procedure call is unaffected by this.

Exception Handling

A common misconception is that exception handling is only a small part of the total application. It is actually the other way around, the normal case is just one case to handle while each fault condition is a separate case to handle [45].

FC elements have ports through which they are connected to other FC elements. Ordinary Steps for example have an in port and an out port which have been used in previous examples. Macro Steps also have two additional ports on the sides, namely the exception port and the history port, which are used for exception handling. An Exception Transition is a special kind of transition that can be connected to an exception port, see Figure 4.6. If `faultA` or `faultB` becomes true while the Macro Step is active the Macro Step will be aborted. The internal state of the Macro Step is then stored and can later be restored through the history port. This is similar to the history pseudostate of statecharts that was described in Section 3.4.

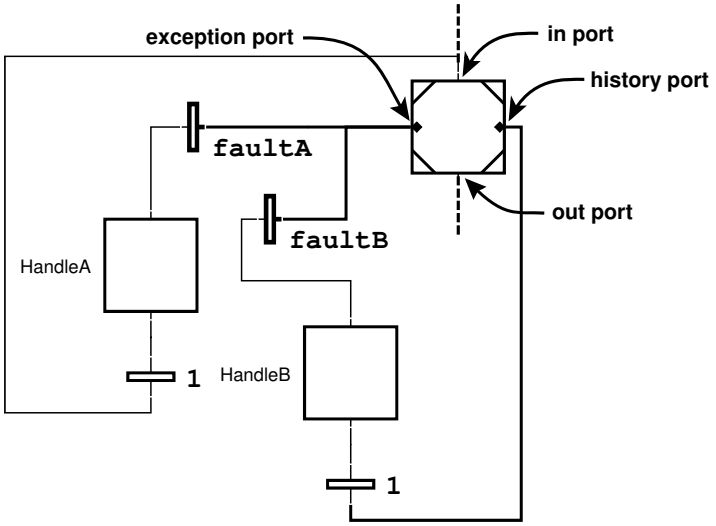


Figure 4.6 Exception Transitions can be connected to Macro Steps and Procedure Steps and are conceptually connected to all steps in the internal FC. When `faultA` has been handled the Macro Step is restarted from its Enter Step. When `faultB` has been handled the Macro Step resumes its execution from the stored state.

Step Fusion Set

The Step Fusion Set is another construct for exception handling which gives a step multiple views, that is, to have one conceptual step appear as several steps. The steps in a fusion set are all active at the same time. All steps in a fusion set are activated together, that is, activate one step in a fusion set triggers the activation of all steps in the fusion set. A fusion set may be either non-abortive and abortive, which determine the deactivation semantics. For non-abortive fusion sets, all steps are deactivated together and for deactivation to be allowed it is required that all steps in the fusion set are allowed to be deactivated. For example, a Macro Step that has not reached its Exit Step is not allowed to be deactivated. For abortive fusion sets the deactivation of one step in the fusion set causes all other steps in the fusion set to be aborted. An example where a fusion set is used for exception handling is shown in Figure 4.7.

Connection Posts

Connection Posts is a way to connect elements without the whole graphical link visible, see Figure 4.8. They can be used to make the FC clearer. Each Connection Post In is connected to a Connection Post Out and each link that goes to the Connection Post In is considered connected to all links from the connected Connection Post Out.

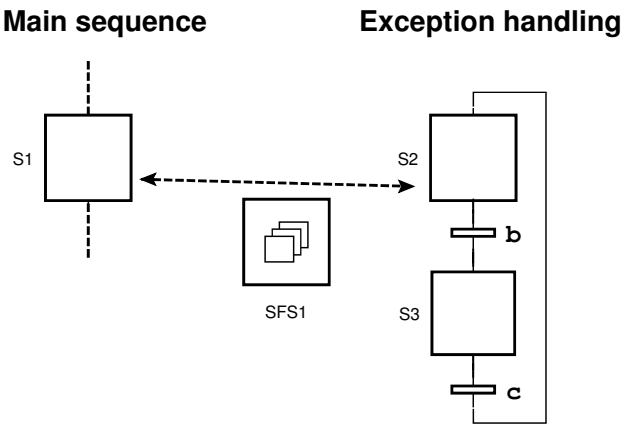


Figure 4.7 In this example it is an exception if `b` becomes true while `S1` is active. The exception handling is implemented in `S3`, completely separate from the main sequence. This is accomplished by using a Step Fusion Set, `SFS1`, which contains the steps `S1` and `S2`. When `S1` is activated, `S2` is also activated. If `b` becomes true while `S2` is active, `S1` and `S2` are deactivated and `S3` is activated. When the exception handling has completed, `S2` is activated which also triggers the activation of `S1`.

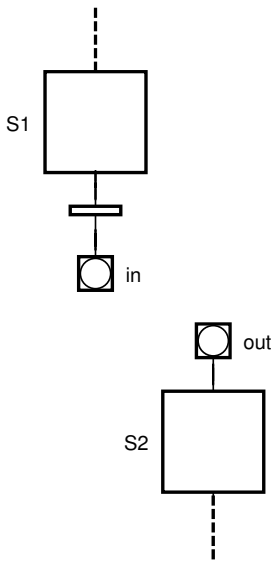


Figure 4.8 The transition is connected to the Step `S2` through Connection Posts.

Function Chart Element Summary

Figure 4.9 shows the main FC elements of Grafchart. The Initial Step is activated when the application starts and thus defines the initial application state. It is otherwise the same as the ordinary Step.

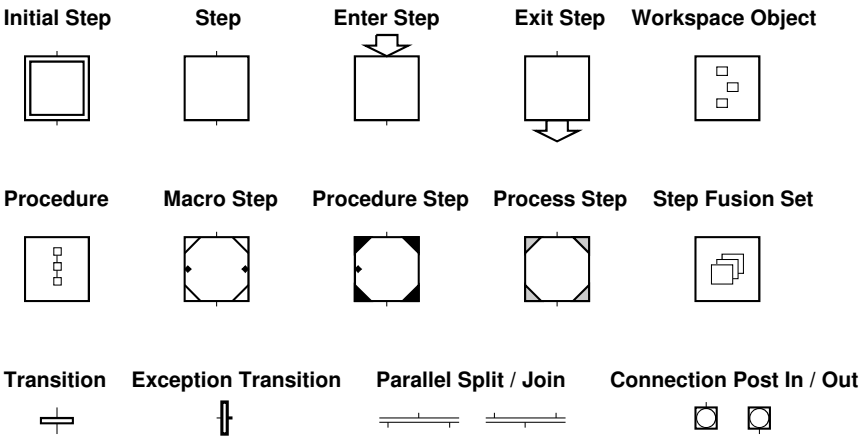


Figure 4.9 Summary of the FC elements of Grafchart.

4.3 Syntax and Semantics

Variables and Scoping

In Figure 4.2 variables were mentioned. Variables can be declared at any level, for example at the top level or inside a Macro Step, Workspace Object, or Procedure. Scoping in Grafchart is similar to ordinary programming languages, see Figure 4.10.

Languages

Grafchart consists of three parts which may be considered separate sub-languages: the FC language (graphical), the action language (textual), and the condition language (textual). The FC language consists of the graphical elements such as steps, transitions, and variables. The action language is used for the actions of steps. The condition language is used for the guard condition of transitions.

The action language uses prefixes to specify the action type of each statement, that is, when it should be executed. In Figure 4.2 the S prefix was introduced. Table 4.1 shows the complete list of prefixes.

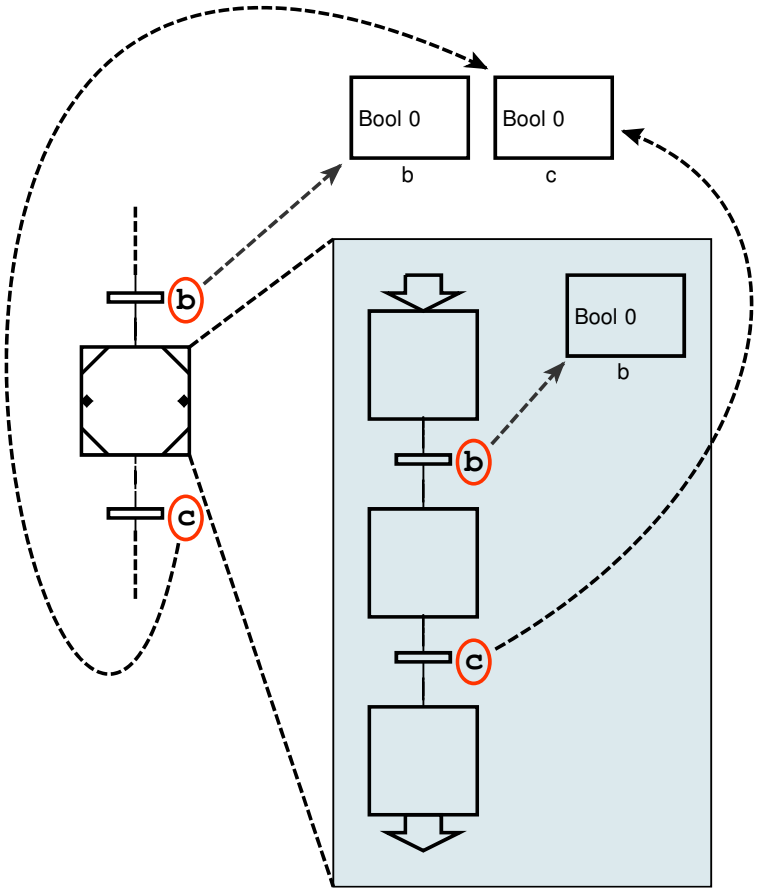


Figure 4.10 Scoping in Grafchart. The arrows indicate which variable that is used.

Execution

For programming languages it is important to have deterministic execution. Grafchart applications are like SFC applications executed periodically, one scan cycle at a time.

To explain the Grafchart execution model it is useful to introduce a few definitions first. A transition is *enabled* when all immediately preceding steps are active. An *enabled* transition *fires* if its guard condition is true. *Firing* a transition involves deactivating the immediately preceding steps and activating the immediately succeeding steps. In the left part of Figure 4.2 the transition is *enabled* since its only immediately preceding step is active, but it cannot *fire* since its guard condition is false. In the right part the transition's guard condition is still true but it cannot

Table 4.1 A complete list of Grafchart and JGrafchart prefixes.

Prefix	Description
S	Action type. Executed when the step is activated.
X	Action type. Executed when the step is deactivated.
P	Action type. Executed periodically while the step is active.
N	Action type. Associates a variable with the status of the step. The variable is set to true when the step is activated and to false when the step is deactivated.
A	Action type. Executed when the step is aborted.
V	Procedure call parameter. Call-by-value (JGrafchart specific)
R	Procedure call parameter. Call-by-reference (JGrafchart specific)

fire again since the immediately preceding step is now inactive and thus the transition is not *enabled*. For a transition following a Macro Step or Procedure Step to be *enabled* the internal Exit Step must also be active. Exception Transitions have priority over ordinary Transitions and are always *enabled* when the Macro Step or Procedure Step is active.

Steps also have some additional properties, namely `x`, `t`, and `s` which are accessed through the step's name, for example `MyStep.s`. `x` is true if the step is active and false if the step is inactive. `t` stores the number of scan cycles that the step has been active since the previous activation if the step is active. For inactive steps `t` is 0. `s` works the same as `t` but counts seconds instead of scan cycles.

A naive execution model could include "Iterate over the transitions and fire a transition if it is enabled and its guard condition is true.". The application behavior then depends on the transition iteration order, as shown in Figure 4.11.

If this behavior is avoided it is guaranteed that activated steps are active during at least one scan cycle which makes it easier to reason about an application. For example, for an N action, the variable will always be true during at least one scan cycle. The same applies to the `x` property of steps.

Another issue is transitions for alternative paths which have conditions that may be true at the same time. This is called a transition conflict and the transitions involved in a transition conflict are called conflicting transitions. The semantics of conflicting transitions in Grafchart is that they all fire. However, this is rarely the intended behavior and is thus practically always a bug in the user application. Note that there cannot be conflicts between Exception Transitions and ordinary Transitions since Exception Transitions have priority over ordinary Transitions.

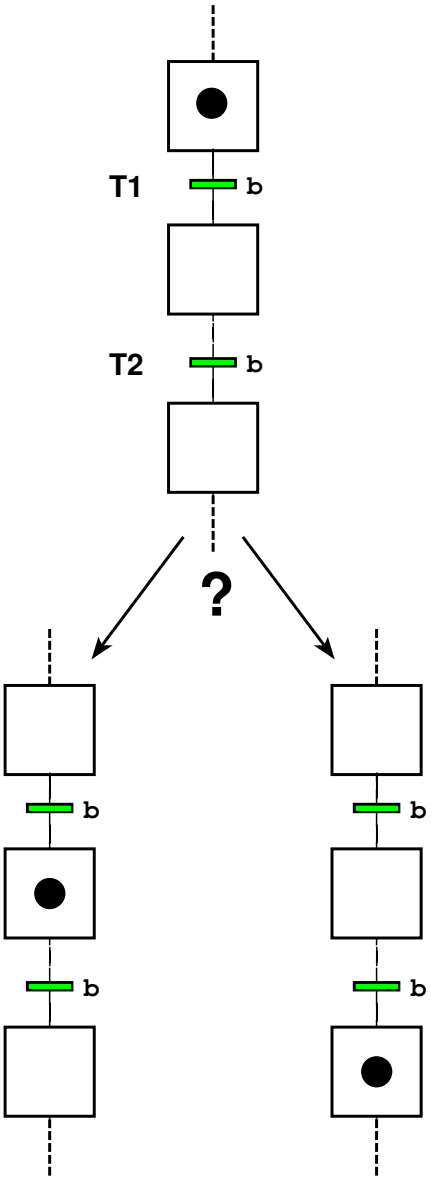


Figure 4.11 With a naive execution model, what happens when `b` becomes true depends on the transition iteration order. If `T2` is checked before `T1` the scan cycle will end in the state to the left. If `T1` is checked first it will fire and when `T2` is then checked it will also fire meaning that the scan cycle will end in the state to the right.

The Grafchart execution model is to do the following during each scan cycle:

1. Read inputs.
2. Mark fireable transitions.
3. Remove mark for conflicting transitions of lower priority.
4. Fire marked transitions.
5. Update step properties `t` and `s`.
6. Execute `P` actions.
7. Mark variables subject to `N` actions.
8. Update marked variables.
9. Sleep until the start of the next scan cycle.

The execution model is compatible with GRAFCET [35], the language which SFC is based on. It gives deterministic behavior for most cases and has the property that a step which is activated always remains active for at least one scan cycle. The remaining non-determinism is for cases where the application should not depend on the execution order anyway. An example of this is the firing order of transitions, affecting which step's `S` and `X` actions are executed first. Another example is which step's `P` actions are executed first.

4.4 JGrafchart Specifics

JGrafchart has many additional features such as load/save in XML format and printing. Some JGrafchart specific extensions and implementation details are described.

Data Types

There are four data types in JGrafchart: Boolean for Boolean values, Integer for integer values, Real for float values, and String for strings. Boolean values are written `1` for true and `0` for false, Integer values are written like ordinary whole numbers, Real values are written with decimal notation, and String values are written as the string value enclosed by quotation marks.

JGrafchart uses loose typing and automatic type casting. The target data type is determined by the context. For example, in Figure 4.12 the value of the Real variable `b` is cast to a Boolean value since that is the data type for transition guard conditions.

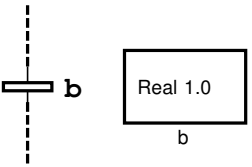


Figure 4.12 The target data type for a transition guard condition is Boolean. Therefore the value of the Real variable `b` will automatically be cast to a Boolean. In this case it is the value `1.0` that is cast and the condition will therefore be true. A complete list of how values are reinterpreted by a cast is included in the JGrafchart documentation.

Variables and I/O

Variables are declared by adding Variable elements to the application. There is a Variable element for each data type. Variables may have an initial value and may also be constant or configured to be updated automatically at the beginning of each scan cycle according to an expression. There is also a List element for each data type. Both Variables and Lists are internal to the application. To interact with an external environment Inputs and Outputs (I/O) are needed.

Custom I/O is one I/O possibility in JGrafchart. It consists of the I/O elements Digital In, Digital Out, Analog In, and Analog Out as well as inverted variant of Digital In and Digital Out. At the beginning of each scan cycle each Analog and Digital In is updated from the external environment. An Analog or Digital Out is written to the external environment whenever assigned. How the I/Os interact with the external environment depends on the chosen I/O implementation. A custom I/O implementation is created by implementing a set of Java interfaces. With a custom implementation it is possible to communicate with practically any external environment. However, note that it is limited to Boolean and float values.

Another I/O possibility is the *Socket I/O* elements. JGrafchart can connect to a TCP server and communicate Boolean, Real, Integer, and String values over a socket with the message protocol: `<identifier> '|' <value> '\n'`. The TCP server is then responsible for interaction with the external environment. *Socket I/O* is often sufficient for implementation of adapters to other communication protocols.

Dynamic References

In most programming languages there are dynamic references which can be used to create higher level applications. In C there are pointers that can be used as dynamic references. In Java any variable used to refer to an object is a dynamic reference. In JGrafchart String Variables can be used as dynamic references, similar to variable variables in PHP. The String's value is then the name of the element which it references. To dereference a String Variable, the `^` operator is used, see Figure 4.13. The result of a dereference operation can be any element with a name and it is used as when the element is named explicitly.

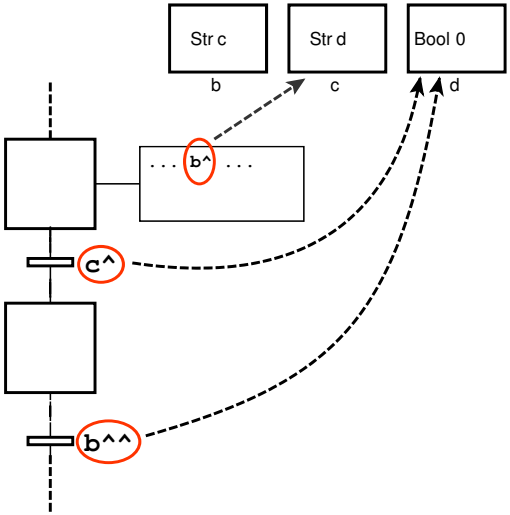


Figure 4.13 The String Variable `b` has the value `c`. As a dynamic reference it thus refers to the String Variable `c`. Similarly the String Variable `c` refers to the Boolean Variable `d`. The expression `b^` will dereference the String Variable `b` and return the String Variable `c`. Similarly `c^` will return the Boolean Variable `d`. Finally `b^^` is equivalent to `(b^)^` which, as previously described, is dereferenced to `c^` which in turn is dereferenced to the Boolean Variable `d`.

If Statement

Conditional execution and evaluation are possible to express with states and transitions. After all, this is the core concept of Grafchart. However, for simple conditional expression evaluations this creates overhead that makes both the expressions and the application appear more complicated. Also, since each step is active at least one scan cycle the evaluation of consecutive conditional expressions takes several scan cycles.

Inline if, also known as the conditional operator and ternary if, is supported by JGrafchart. Figure 4.14 shows a small FC written with and without *inline if*. The implementations behave slightly different: The left part executes the initialization in a separate scan cycle while the right part executes everything in the same scan cycle.

Graphical Elements

The online view of the running application is useful for developers and maintenance staff. For the operators of the production machine on the other hand, it is more intuitive with an interactive interface that resembles the machine. In JGrafchart it is possible to create interactive operator interfaces with graphical elements. Figure 4.15 shows a JGrafchart operator interface.

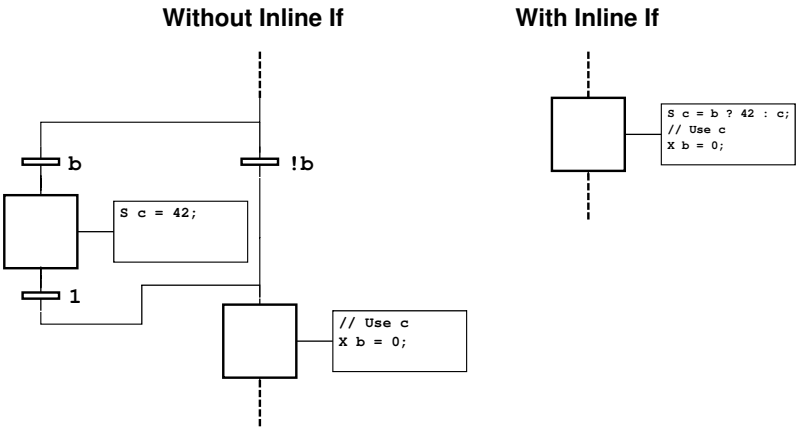


Figure 4.14 The variable `c` needs to be initialized to 42 if `b` is true. The left part shows how to implement this without *inline if*. The right part shows how it can be implemented with *inline if*.

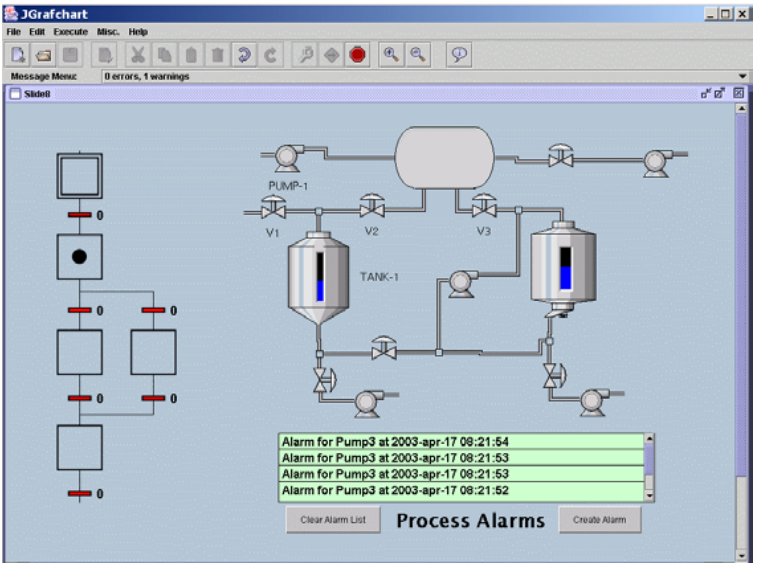


Figure 4.15 An operator interface with process alarms, the executed application, and animated tanks and piping that resemble the controlled process.

Basic graphical elements available are rectangles, ellipses, and lines. Custom images, interactive buttons, and live plotters are also available. In fact, most figures in this chapter were created with JGrafchart. It is also possible to manipulate the graphical elements from the application to create animations.

Functions, Methods, and Properties

JGrafchart provides a library of built-in functions, for example mathematical functions like `sqrt`, `sin`, and `abs`. Most elements also have methods. For example, a graphical element has methods to get and set the element's size and location. These methods can be called from a Grafchart application to create animations. Some elements also have properties, for example steps have the `x`, `t`, and `s` properties.

4.5 Comparative Example

As mentioned in Section 4.2, Macro Steps can make larger applications understandable. Here this is shown by comparing an application implemented in both standard SFC, SFC with Macro Steps, and JGrafchart. Note that the example is rather small. For larger applications Macro Steps and the additional JGrafchart features are even more beneficial.

A bead sequencing and sorting process is shown in Figure 4.16. It consists of two compartments, one with yellow beads and one with black beads. In sequencing mode the task is to sequence a pattern of yellow and black beads. Turned upside down the process is in sorting mode and the task is then to sort the beads back into their corresponding compartments.

The sequencing process can be controlled by the SFC application in Figure 4.17. In Figure 4.18 Macro Steps have been used to structure the application. The inner loop has been moved into a separate Macro Step and the alternative paths have been moved into Macro Steps to make the flow appear more linear. The overall control logic is then considerably clearer. The internal FC of the Macro Step `ReleaseBeads` is shown in Figure 4.19. The other two Macro Steps contain the black/yellow alternative paths in the SFC application and are omitted. Note that the exception handling, the transitions concerning `nAttempts`, must be implemented twice to move the inner loop into a Macro Step. The structured application is much easier to overview, making it easier to understand and maintain.

In Figure 4.20 and Figure 4.21 additional Grafchart and JGrafchart features have been used. With two exits from the Macro Step, one for the normal case and one for the exceptional case, it was possible to remove the redundant exception handling. With use of dynamic references (`^`) and inline if (`?:`) several alternative paths have been removed. The resulting application has a linear flow with a minimal number of loops. The state flow is thus as easy as possible.

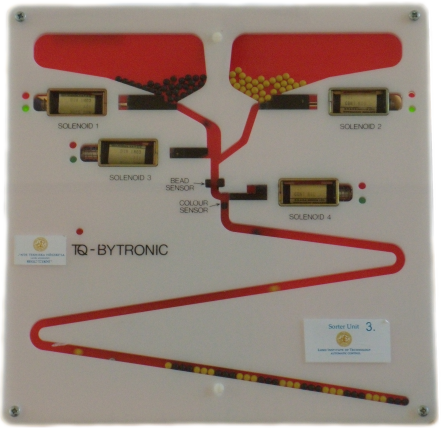


Figure 4.16 The bead sequencing and sorting process. In sequencing mode yellow and black beads are sequenced into a pattern, here 3 black 3 yellow. Turned upside down the process is in sorting mode and the task is then to sort the beads back into their corresponding compartments.

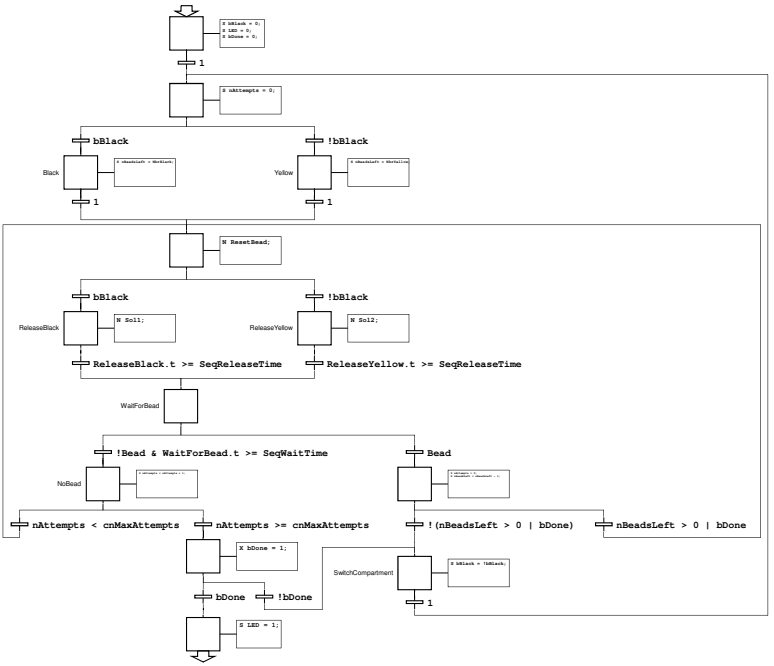


Figure 4.17 An SFC application to control the bead sequencing process. Variables are omitted and the code is not intended to be readable in the printed version.

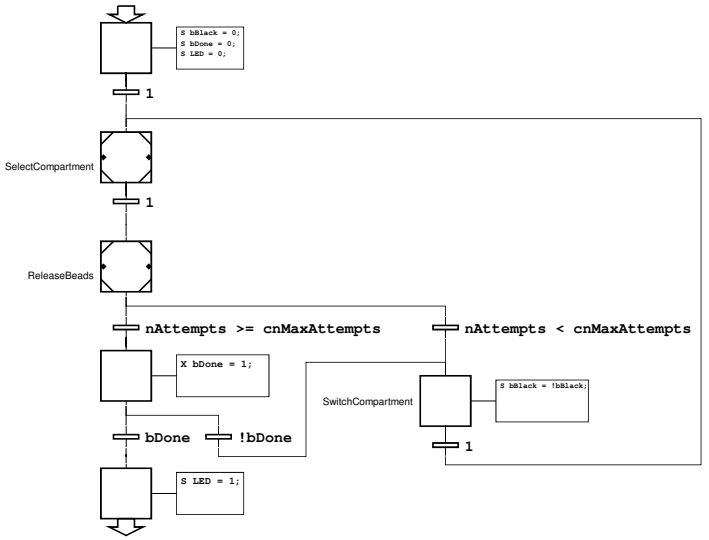


Figure 4.18 The application in Figure 4.17 structured with Macro Steps. The internal FC of `ReleaseBeads` is shown in Figure 4.19.

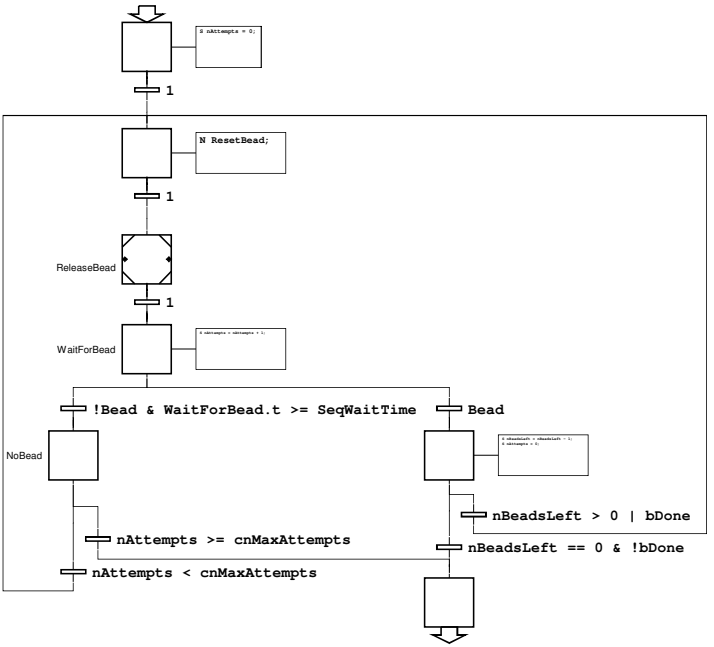


Figure 4.19 The internal FC of the Macro Step `ReleaseBeads` in Figure 4.18.

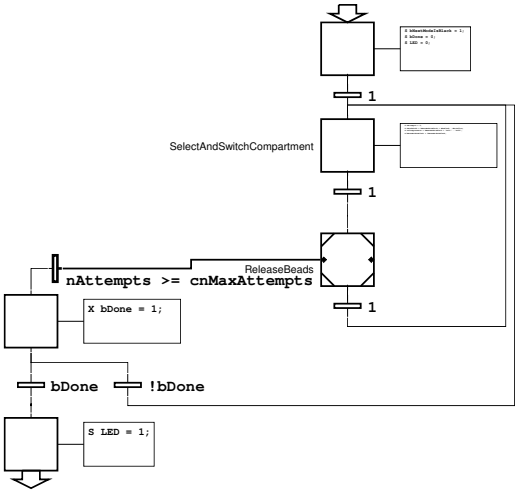


Figure 4.20 The application in Figure 4.18 simplified with additional features of Grafchart and JGrafchart. The internal FC of `ReleaseBeads` is shown in Figure 4.21.

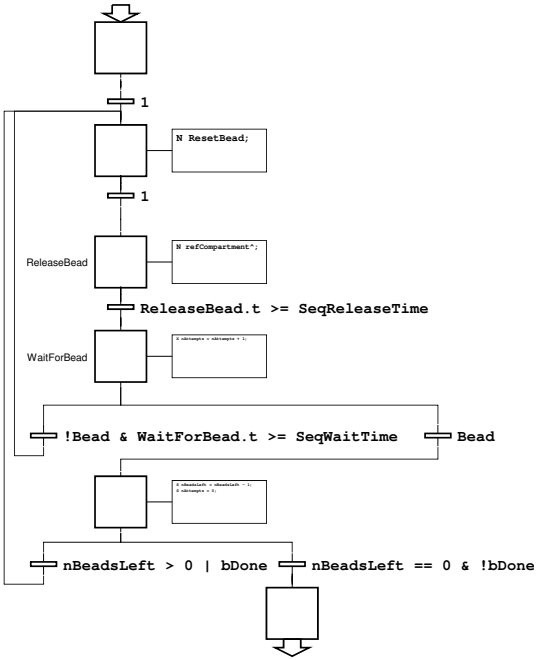


Figure 4.21 The internal FC of the Macro Step `ReleaseBeads` in Figure 4.20.

4.6 G2Grafchart Specifics

Everything so far describes the Basic Version (BV) of Grafchart which is implemented in both G2Grafchart and JGrafchart. In G2Grafchart there is also a High-Level Version (HLV) which is a superset of BV. In BV there is essentially only one token while in HLV it is possible to have several tokens and to spawn and consume tokens dynamically. Tokens in HLV may also contain data (compare to colored Petri nets). They may also contain internal FCs, a feature called multi-dimensional charts. Objects in G2Grafchart may also have parameters and attributes. The notation to access token and object data in actions and conditions is `inv` for token data, `sup` for object parameters, and `self` for object attributes. For more details about G2Grafchart, see [35].

5

Grafchart Language Improvements

As shown in Section 4.5, Grafchart's hierarchical structuring and exception handling make it possible to create more well structured and conceivable applications. Together with Grafchart's features for reusability this leads to better flexibility. If JGrafchart would support HLV or other new language constructs it would be possible to create even more flexible applications. JGrafchart would thus benefit from a more extensible language implementation where new constructs are easier to add and evaluate.

Currently, JGrafchart applications can only be executed in an interpreted manner. Also, the same Java object instances are used both for execution and application visualization. Hence, there are no timing guarantees which means that JGrafchart cannot be used for hard real-time control. For core robotics applications, a scan cycle time of a few milliseconds is often required. Forces build up quickly and a missed deadline can be the difference between smooth and harmful behavior. With the current version of JGrafchart (2.6.1) not even tiny applications execute reliably at this rate.

In this chapter the challenge of flexible production systems is addressed by improving the Grafchart language and its implementation JGrafchart to make them more flexible. This is achieved by adding new constructs to Grafchart and by applying modern compiler techniques to JGrafchart to make the language implementation extensible. To make Grafchart usable for industrial automation on device level, a step toward real-time execution is made. The new Grafchart constructs and real-time execution are useful for both process automation, manufacturing, and robotics. Hard real-time execution is particularly useful for robotics where the execution rate is higher.

5.1 New Constructs

Transition Priorities

To ensure that there are no conflicting transitions is tedious, error prone, and makes the code verbose and hard to overview. It is tedious since it is repetitive and has to be done everywhere where there are alternative paths. It is error prone since it has to be done manually. It makes the code verbose since additional terms need to be added to the otherwise conflicting transitions. An example of this is shown in Figure 5.1.

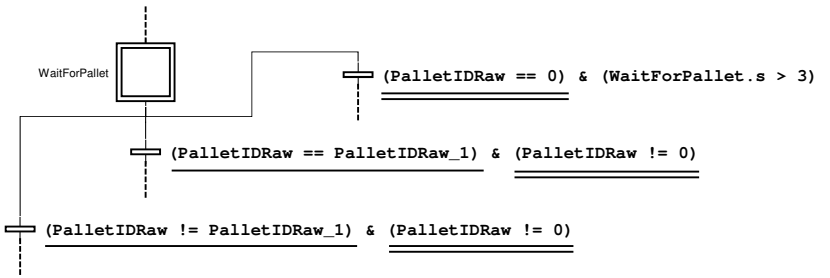


Figure 5.1 An example of an application where the guard conditions to the alternative paths have been manually resolved to avoid conflicting transitions. The intention for this application was to first evaluate the left transition, then the middle transition, and finally the right transition. The term `(PalletIdRaw == PalletIDRaw_1)` has been added to the transition in the middle to make the left and the middle transitions mutually exclusive. The term `(PalletIDRaw == 0)` has been added to the transition to the right to make all transitions mutually exclusive.

Special care is needed for non-deterministic guard condition expressions, for example expressions that use random number generators. Such expressions cannot be directly used in alternative transitions. One solution is to use an intermediate variable that is assigned with a `P` action in the immediately preceding step. Note, however, that the transition will then be fired during the next scan cycle. The I/O inputs have then been updated between the assignment of the intermediate variable and the execution of `S` and `X` actions. This must also be taken into account. Similarly, special care is required for expressions with side effects. In general, expressions with side effects should be avoided in guard conditions.

A more robust way to resolve conflicting transitions is to use transition priorities. Previously, step 3 of the Grafchart execution model, "Remove firing mark for conflicting transitions of lower priority", only gave exception transitions priority over ordinary transitions. Similarly, internal priorities among transitions of the same kind can be introduced. Exception transitions still have priority over ordinary transitions but an exception transition may have priority over another exception transition and an ordinary transition may have priority over another ordinary transition.

Priority 1 is the highest priority, priority 2 the second highest and so on. Transitions with no priority specified have the lowest possible priority. With priorities the application fragment in Figure 5.1 can be simplified, see Figure 5.2. Here only the core of the guard conditions are included and the code is more concise. As the transitions have different priority it is trivial to conclude that they will never be conflicting. Transitions with the same priority may still be conflicting and, like before, all conflicting transitions will then fire.

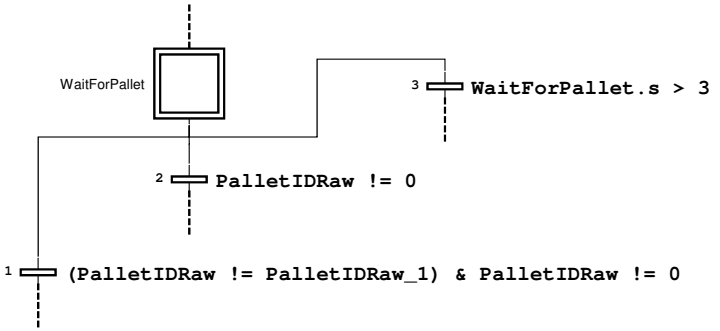


Figure 5.2 A more concise implementation of Figure 5.1 where transition priorities have been used. The transition priority is written to the left of the transition.

Another use of transition priorities is else paths, that is, a path which should always be taken if no other path is taken. Without transition priorities, the guard condition for the else path would be the and of the negation of the guard condition of each otherwise conflicting transitions. For example, if the guard condition of three otherwise conflicting transitions are `c1`, `c2`, and `c3` the guard condition for the else path is `!c1 & !c2 & !c3`. With transitions priorities, the transition to the else path is simply given lowest priority and guard condition `1`. The `ReleaseBeads` Macro Step for the bead sequencing task in Figure 4.21 has alternatives in two places, where one is a typical else path. With transition priorities it can be made more concise, see Figure 5.3.

It is good practice to always assign priorities to alternative transitions to eliminate the risk of conflicting transitions. Consider for example the transition with the guard condition `Bead` in Figure 5.3. Without the priority there will be a transition conflict in the rare case when the bead is detected in the same scan cycle as the timeout occurs. This is also a case that is easy to forget.

To add transition priorities increases the expressive power of the Grafchart language slightly but it is mainly syntactic sugar which makes it easier to deal with conflicting transitions. Most applications which use transition priorities can be transformed to equivalent applications which do not use transition priorities. For each transition, the negation of its condition is added to each alternative transition with lower priority, see Figure 5.4. However, this only works for deterministic expres-

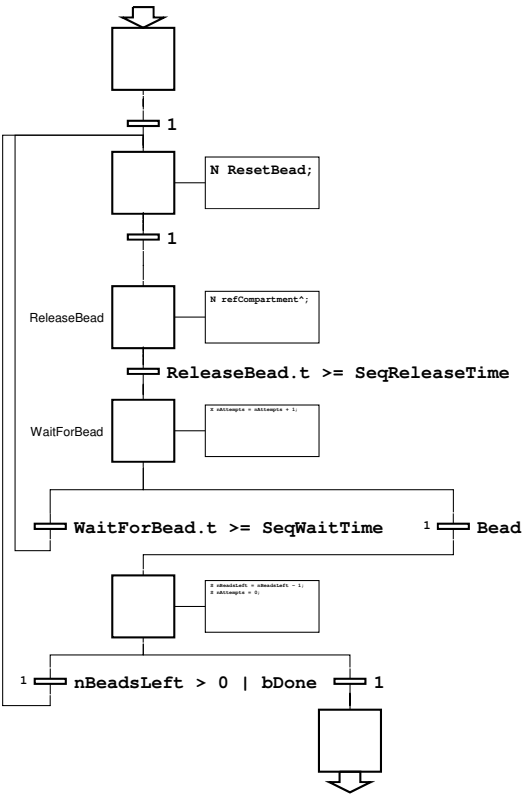


Figure 5.3 The Grafchart application for bead sequencing in Figure 4.21 simplified with transition priorities. The `WaitForBead` condition and the else path for `nBeadsLeft` are more concise.

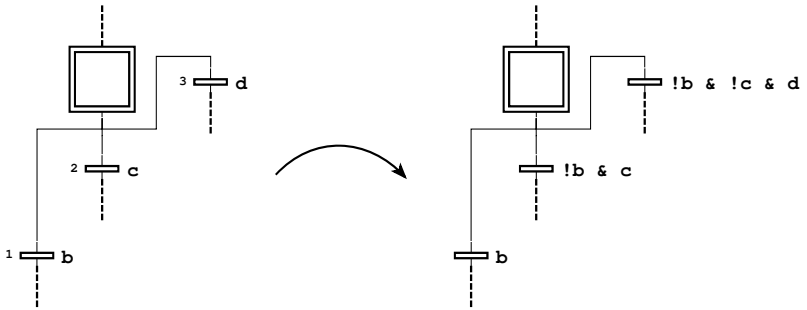


Figure 5.4 A Grafchart application with priorities can be transformed to an application that does not use transition priorities if the guard condition expressions are deterministic.

sions. In the general case, with side effects or non-deterministic functions, the application behavior cannot be perfectly preserved. One example is when a high priority transition has a guard condition which calls a non-deterministic function with the current value of an input as an argument.

Macro Step Resume Mode

If the execution of a Macro Steps is aborted by an exception transition, its internal state is stored and can later be resumed, see Figure 4.6. Resume is recursive which means that Macro Steps within Macro Steps will also be resumed. This corresponds to a deep history pseudostate in statecharts, see Section 3.4. Another way to abort a Macro Step is with an abortive Step Fusion Set. Recall that if one step in an abortive Step Fusion Set is deactivated, all other steps in that Step Fusion Set are aborted.

A Macro Step activated through a Step Fusion Set is a normal activation, that is, the same as an activation through the Macro Step's in port. In [40] the desired behavior was to have a resume activation for abortive Step Fusion Sets, that is, the same as activation through the Macro Step's history port, see Figure 4.6. To be able to choose whether or not a Macro Step resume should be recursive, similar to the shallow history state in statecharts, was also needed. A resume mode setting for each Macro Step was thus added.

Three possible resume modes have been defined, namely *default*, *always*, and *never*. A Macro Step with the *default* resume mode behaves the same as before the resume mode setting was introduced, that is, the Macro Step is resumed recursively on resume activation and activation through an abortive Step Fusion Set is a normal activation. A Macro Step with the *always* resume mode is resumed recursively on resume activation and activation through an abortive Step Fusion Set is a resume activation. A Macro Step with the *never* resume mode is never resumed, that is, a resume activation of the Macro Step will instead trigger a normal activation. This is particularly useful to break a resume recursion and can, for example, be used to implement shallow resume.

The Macro Step resume mode makes it possible to customize how and when a Macro Step should be resumed. Without the setting, similar behavior could often be obtained but would require much effort. To get a shallow resume for example one way was to store which step was active when the Macro Step was aborted and manually go back there on "resume" activation, see Figure 5.5.

5.2 Compiler Techniques

As mentioned in Section 4.6 there are two versions of G2Grafchart, namely the Basic Version (BV) and the High-Level Version (HLV). JGrafchart corresponds to BV and extending it with HLV would enable more flexible ways to write applications. It is also desirable to retain a pure BV and have both versions available. Since HLV is a superset of BV it is desirable to reuse the BV implementation as base for the

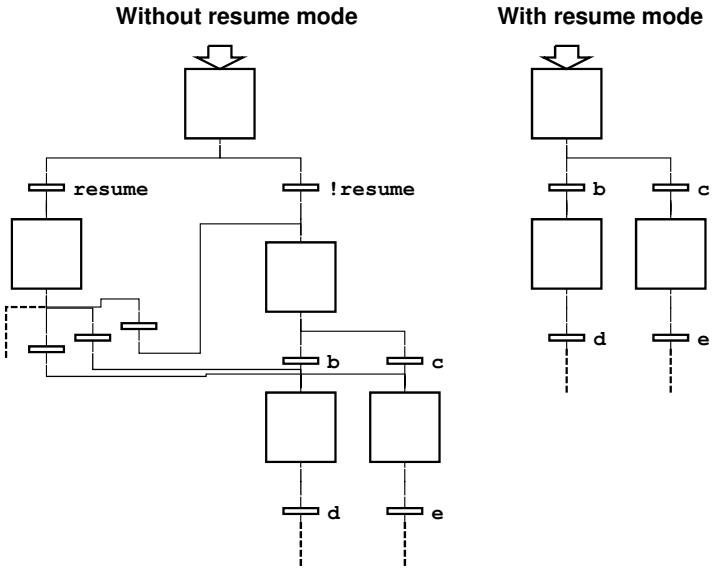


Figure 5.5 The internal FC of a Macro Step that should have a shallow resume. Without resume mode the aborted state can be stored and resumed manually.

HLV implementation. Consequently, redundant code is avoided and maintainability is improved. This imposes an extensibility requirement on both the editor and the compilers.

The work presented in this section introduces modern compiler techniques to an automation language with the goal to make it extensible. Thus a more flexible language implementation is obtained. To use modern compiler techniques for JGrafchart is also a natural first step toward real-time execution as they make it easier to extend the implementation with code generation. This section concerns the compilers for the textual languages of Grafchart [46].

Background

Programming Languages A programming language can be described by its *syntax*, *semantics*, and *pragmatics* [47]. The *syntax* describes the allowed structure of the language, the *semantics* describes the meaning of the syntactic elements, and the *pragmatics* describes what the constructs in the language can be useful for. Take assignments for example. The *syntax* describes how assignments are written, for example a valid assignment in Grafchart is `temp = 12 * y;`. The *semantics* describes that an assignment means to evaluate an expression and assign the result to a variable. In this case the expression `12 * y` will be evaluated and assigned to the variable `temp`. The *pragmatics* describe what an assignment can be useful for, in this case to initialize a temporary variable.

Compilers The task of a compiler is to transform written applications into something executable. A typical compiler works in a sequence of phases, see Figure 5.6. The input to the compiler is the application source code. The *scanner* splits the source code into a sequence of classified tokens. The *parser* uses these tokens to build an Abstract Syntax Tree (AST). The AST contains all information needed to execute the application. Executable code can be generated from the AST or an interpreter can execute the AST directly.

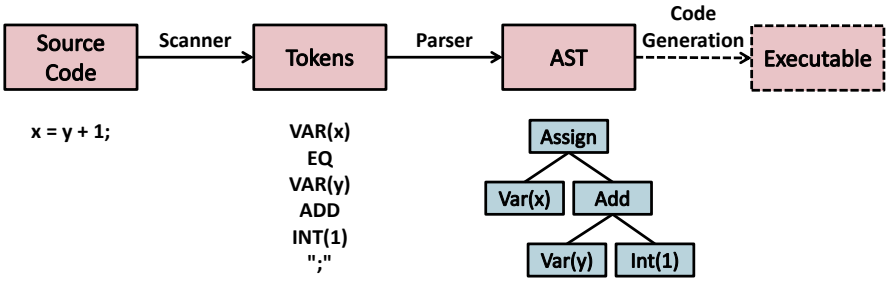
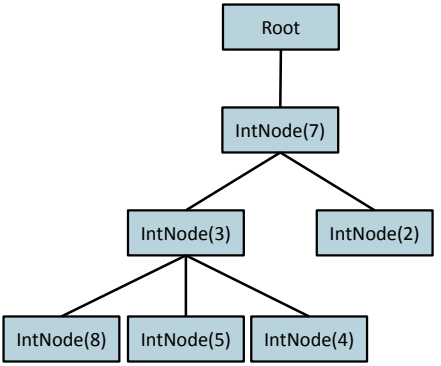


Figure 5.6 The overview of a compiler. The *scanner* transforms the source code `x = y + 1` into a sequence of classified tokens. For example, `x` is classified as the token *variable x* and `1` is classified as the token *integer 1*. The *parser* uses these tokens to build an AST representation of the application.

The *scanner* and *parser* check that the application is syntactically correct, a prerequisite to build an AST. An AST must also be analyzed to check if the application is semantically correct. In Figure 5.6 the application is always syntactically correct but semantically correct only if the variables `x` and `y` exist.

Rewritable Reference Attribute Grammars Attribute grammars with *synthesized* and *inherited* attributes were introduced by Donald Knuth [48]. The main difference between attribute grammars and traditional compiler techniques is that attribute grammars are declarative while traditional compilers are imperative. Instead of explicit AST traversal, the semantics are specified with equations. The main advantage of declarative programming is that the evaluation order is determined automatically and does not have to be considered. The focus shifts from designing the evaluation order to just adding equations for the desired attributes. Rewritable Reference Attribute Grammars (ReRAGs) adds several new concepts such as *reference attributes*, *collection attributes*, *parametrized attributes*, *circular attributes*, and *node rewrites*.

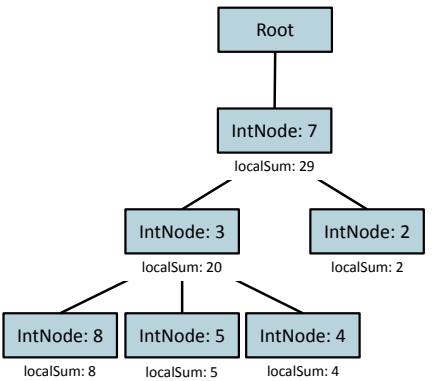
Consider the toy AST with integer nodes in Figure 5.7. It consists of a `Root` node with one `IntNode` child node and each `IntNode` has an integer value and zero or more `IntNode` child nodes.




```
Root ::= IntNode;
IntNode ::= <Value:int> IntNode*;
```

Figure 5.7 A toy AST of integer nodes. Each `IntNode` has a value and zero or more child nodes. For example, the `Root` node's child node has the value 7.

Synthesized attributes are assigned locally and depend on the local subtree. In Figure 5.8 the *synthesized attribute* `localSum` has been added for the local sum of a subtree.




```
syn int IntNode.localSum() {
  int localSum = getValue();
  for (IntNode child : getIntNodes()) {
    localSum += child.localSum();
  }
  return localSum;
}
```

Figure 5.8 The AST from Figure 5.7 attributed with the local subtree sum for each `IntNode`. This is done by adding a *synthesized attribute* called `localSum` of type `int`, which is calculated as the node's value plus the local sums of its child nodes.

Inherited attributes are assigned by an ancestor. In Figure 5.9 the *inherited attribute* `globalSum` has been added for the sum of the whole tree.

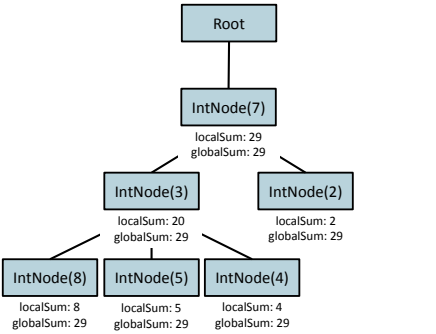


Figure 5.9 The AST from Figure 5.8 attributed with the sum of the whole tree for each `IntNode`. On the first line the *inherited attribute* `globalSum` is declared for all `IntNodes`. For each `IntNode` there must be an equation in an ancestor node that defines the value of this attribute. The second line specifies an equation that applies recursively to all nodes, starting from `Root.getIntNode()`, that is, the `IntNode` directly below a `Root` node. The equation's value is `localSum` from the `IntNode` directly below the `Root` node, which is the sum of the whole tree.

Parametrized attributes are attributes which may depend on parameters. In Figure 5.10 a *parametrized attribute* has been added, which checks if the local sum is greater than the supplied argument.

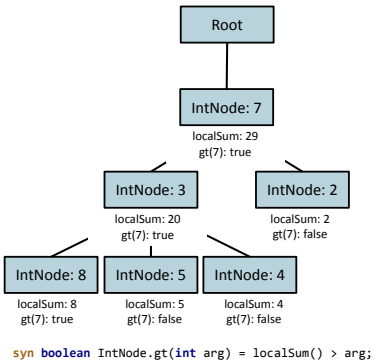


Figure 5.10 The AST from Figure 5.8 attributed with a *parametrized attribute* called `gt` for each `IntNode`. It is used to check if the local sum is greater than the argument, here `gt(7)` is shown.

Reference attributes are attributes which refer to other nodes in the AST. This was not allowed for the original attribute grammars as it may cause circular evaluation dependencies. However, this can be detected by the evaluation framework. In Figure 5.11 a *reference attribute* to the root node has been added for all nodes.

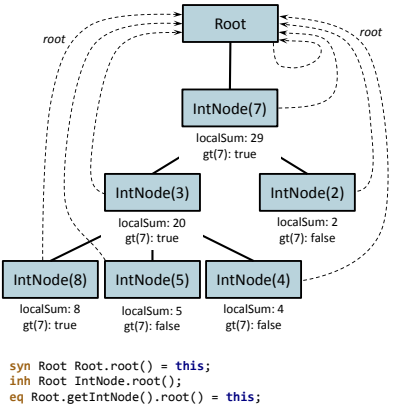


Figure 5.11 The AST from Figure 5.10 attributed with a *reference attribute* to the root node for all nodes. The `Root` node specifies itself as its own `root` with a *synthesized attribute* and as the `root` of all other nodes with an *inherited attribute*.

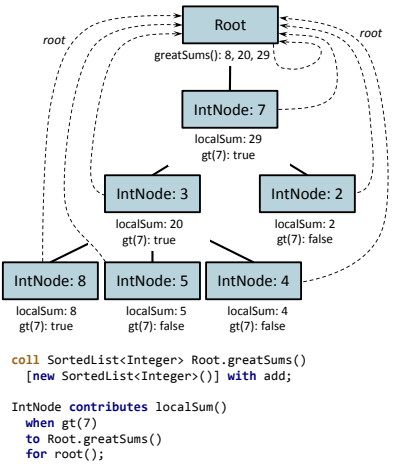


Figure 5.12 The AST from Figure 5.11 attributed with a *collection attribute* which contains all local sums greater than 7. It is defined for all `Root` nodes and the `IntNodes` with a local sum greater than 7 contribute their sum to the collection of their `root` reference.

A *collection attributes* is a node local collection, to which any node in the tree can contribute. In Figure 5.12 the *collection attribute* `greatSums` which contains all local sums greater than 7 has been added to the `Root` node.

A *circular attribute* is an attribute which is allowed to depend on itself. *Circular attributes* are evaluated iteratively until the value has converged. Finally, *node rewrites* are used to replace nodes in the original AST before attributes are evaluated.

JastAdd ReRAGs are implemented in the open source compiler compiler system JastAdd [49]. It has been used to successfully implement extensible compilers for a wide variety of purposes, for example the JastAdd Java Compiler (JastAddJ) that is written as a Java 1.4 compiler with Java 1.5 [50] and Java 7 [51] extensions, the Control Module Language with object-oriented extensions [52], and the Optimica extension to Modelica [53]. It was thus an attractive candidate to make the JGrafchart compilers extensible. Other alternatives considered include Eli [54], Synthesizer Generator [55], Silver [56], and Kiama [57].

In JastAdd, imperative compiler code can be mixed with attributes, which makes it possible to convert a traditional compiler piece by piece. The semantics specification can also be split up in modules (aspects) based on for example functionality.

JGrafchart Compilers

As mentioned in Section 4.3 Grafchart consists of three languages: the FC language, the action language, and the condition language. In JGrafchart there is a separate compiler for each language. The AST consists of an FC AST and ASTs built by the action and condition compilers. The FC AST contains for example steps, transitions, variables, I/O, and graphical elements. Below each step in the FC AST there is an action language AST. Below each transition in the FC AST there is a condition language AST. The action and condition ASTs depend on the FC AST, for example variables are declared in the FC AST and used in actions and conditions. Thus the complete AST is needed for semantics analysis.

The application in Figure 5.13 implements a controller for a batch tank that is filled until `full`, then emptied until `empty`. The sequence is repeated indefinitely, and each time the filling is initiated the `cycles` counter is incremented. In the current execution state the fourth filling has just been initiated. The AST for the application is shown in Figure 5.14.

The compilers for the action and condition languages were previously written with traditional compiler construction techniques and tools (JGrafchart version 1.5.3.4). The scanners and parsers were generated with JavaCC [58]. Semantic checks were then added by inserting handwritten code into the generated files. Interpreter code for execution was also added to the same files. This is a common way to implement compilers. It is similar to ordinary programming and most developers are familiar with the techniques. It is also easier for developers without specific compiler technique knowledge to work on these compilers. However, it has the

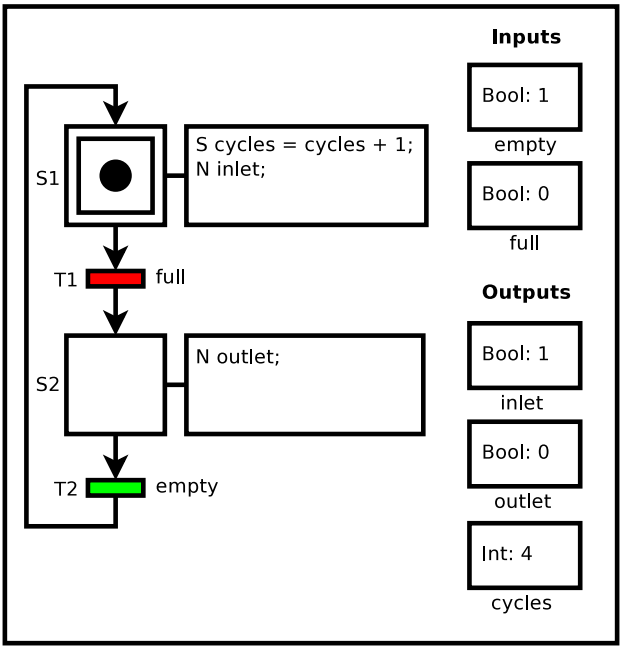


Figure 5.13 A control application for a batch tank that is filled until `full` and then emptied until `empty`.

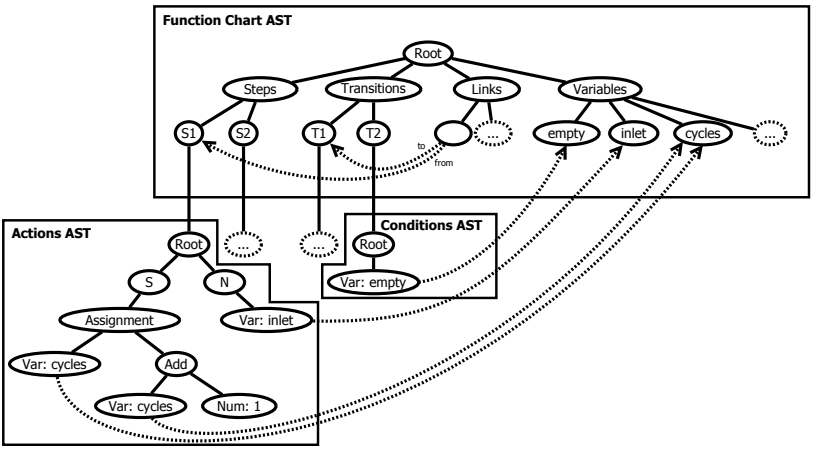


Figure 5.14 The AST for the application in Figure 5.13. The steps, transitions, and variables are used to build an FC AST by the FC compiler. For each step the actions source code is read by the action compiler to build an action AST. The condition text of each transition is used by the condition compiler to build a condition AST.

following drawbacks and trying to create an extension under these circumstances would be error-prone:

- The semantics is written with imperative code which is inherently hard to extend.
- The semantics code, the interpreter code, and the generated code are intermixed.
- The functionalities are hard to overview since they are split up in all contributing Java classes.

Rewriting the Compilers

The following strategy to rewrite the compilers was used, see Figure 5.15:

1. Separate handwritten code from generated code.
2. Split the handwritten code into logical modules based on functionality.
3. Simplify the semantics analysis with ReRAGs.

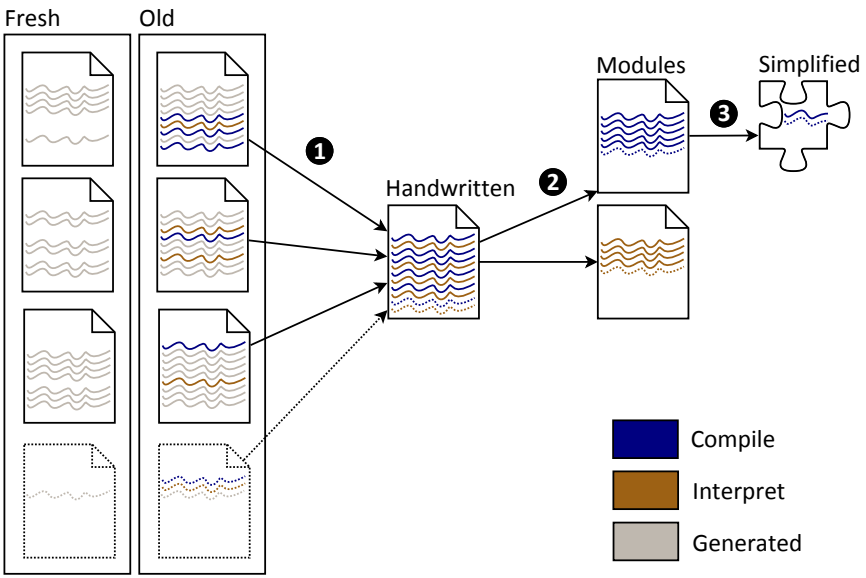


Figure 5.15 How the JGrafchart compilers were rewritten. First the handwritten code was extracted, then it was split up into modules, and finally the compile module was simplified.

These steps were applied to both the action and the condition language implementation in turn. For convenience the condition language was considered first as it is roughly a subset of the action language.

Step 1: Separation To separate generated from handwritten code was straightforward. The scanner and parser were generated from specification into an empty directory and then compared to the current code. Code only present in the current code was considered handwritten and moved into a single JastAdd module.

JastAdd must be told which node types are available. As a starting point a plain list of all node types was used. Later it was rewritten to make the AST structure explicit (like the toy AST specification in Figure 5.7).

Step 2: Split Into Modules The modules chosen for both language implementations were Compile, Interpret, and Utilities. The Compile module handles the semantic analysis of the AST, the Interpret module handles interpreted execution, and the Utilities module contains various helper functions.

Built-in functions and methods do not belong in either module since information about them is required during compilation and their implementation is required during execution. Previously, they were implemented separately for the action language and the condition language. Since most of them are available in both languages it is better to only have them implemented once. Therefore they were extracted to a separate package which is used by both language implementations during both compilation and execution.

Step 3: Simplification The Compile modules were transformed piece by piece to ReRAG equations. In parallel, automatic tests were created to verify that nothing was broken. For this purpose the Java unit testing framework JUnit [59] was used.

The old implementation performed a one-pass traversal of the entire AST and compilation messages were sent directly to the editor during traversal. To determine if the compilation was successful a separate Boolean variable was propagated upward in the tree and returned by the root node.

The new implementation uses a collection attribute in the root node for the compilation messages which the editor fetches. The root node determines if the compilation was successful by checking if the collection attribute contains any error messages. In fact, all the new compiler interface implementation does is to receive the compilation context and check the collection attribute, see Figure 5.16.

Since the interpreters specify what to do, rather than what to calculate, the Interpreter modules were kept as imperative code.

Evaluation

Confirming Extensibility In object-oriented programming a useful and common feature is method overriding. When using the hierarchical constructs of Grafchart for object-orientation it is useful to be able to override variables and procedures. However, in many cases the overriding implementation needs to use the overridden

```
public boolean Root.compile(Context c) {
  if (!isValidContext(c))
    return false;
  this.c = c;
  for (CompilationMessage msg : messages())
    if (msg.isError())
      return false;
  return true;
}
```

Figure 5.16 The new compiler interface implementation.

variable or procedure. Currently this is not possible in JGrafchart. To support this there must be a way to bypass the local context during lookup. The `sup` notation is proposed to bypass the local context, see Figure 5.17. To add `sup` as an extension to the new implementation was straightforward. This indicates that the new implementation is indeed extensible.

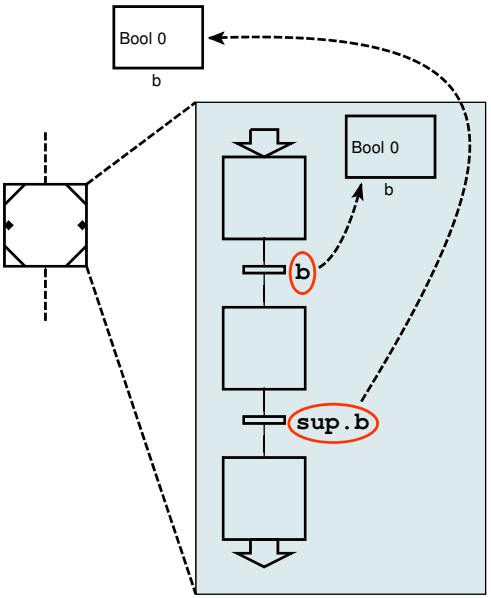


Figure 5.17 Variable bindings with and without `sup`. Without `sup` the local variable `b` is found. With `sup` the local context is skipped and the non-local variable `b` is found.

Code Size Comparison A common metric for the size of software programs is Source Lines Of Code (SLOC). It was chosen to show the difference in implementation size between the old and the new implementation, see Table 5.1. Care has been taken to make the comparison as fair as possible.

Table 5.1 SLOC comparison of the old and the new implementation. New_{eq} is equivalent to New, but written equally dense as Old. In Total_{excl} Built-ins are excluded. In Total_{fair} Separate Generated has also been excluded.

	Old	New	New_{eq}
Compile	1537	380	209
Interpret	1389	1089	983
Built-ins	3462	3514	-
Utilities	110	134	126
AST	2	114	52
Includes	230	-	-
Mixed Generated	1513	-	-
Separate Generated	6628	-	-
Dead	723	-	-
Total	15594	5231	-
Total_{excl}	12132	-	1370
Total_{fair}	5504	-	1370
sup	-	53	38

Initially, all lines were counted regardless of whether they were statements, comments, or empty lines. Then an equivalent new implementation was created by making it as dense as the old implementation, that is, with a similar amount of comments and empty lines. This is denoted New_{eq} in Table 5.1.

Since the functionalities in the old implementation were intermixed, both with each other and with the generated code, to determine the number of lines for each functionality had to be done manually by reviewing each line. To make the comparison as fair as possible, there are separate categories for dead code and files that only contain generated code.

Import statements in the old implementation have been counted separately while in the new implementation they have been counted with the corresponding modules.

In the old implementation the AST structure was implicitly determined by the parser specification together with the JJTree (part of JavaCC) stack implementation. The two lines counted as AST in the old implementation are manual AST structure modifications to the generated code.

The new Compile modules are 75% smaller than the old ones. Also, the new implementation is not as dense as the old one. Comparing the old implementation to the equally dense New_{eq}, the new implementation is 86% smaller. In addition, the new Compile modules have been enhanced with several new compiler checks and additional attributes have been added to simplify the interpreter. The simplified compilers use in total eleven different synthetic attributes, five different inherited attributes, and one collection attribute. There are also three different node rewrites.

One of the attributes is a reference attribute and one is a parameterized attribute.

The new Interpret modules are smaller, mainly because duplicate code has been removed. With all interpreter code gathered in one place it was easy to detect and eliminate the redundant code. The new implementation has also been enhanced, for example with support for multiple dereferences within an expression, like `b^^` in Figure 4.13.

Previously, built-in functions and methods (Built-ins) were implemented in both the action compiler and the condition compiler with anonymous classes. In the new implementation they are only implemented once and discrepancies between the implementations have been detected and resolved. The anonymous classes have been converted to public classes, which require more overhead but are easier to maintain. The overhead of the public classes is the reason why the new implementation is much larger than, as would be expected, half the size of the old implementation.

The Mixed Generated lines in the old implementation are roughly 20% of the lines in the manually maintained files. With the new implementation no generated files have to be maintained.

The most fair comparison is Total_{fair} where the implementations are equally dense and separate generated files and built-ins are excluded. The new implementation is then 75% smaller than the old one.

To implement the `sup` extension required only 53 lines of code, whereof 20 lines for the scanners and parsers and 33 lines for the semantics.

Performance Comparison A drawback with attribute grammars is longer compilation time. Here it is evaluated to check if this is an issue. The compilation code of the old and the new implementation of JGrafchart were instrumented manually. Compilation was performed 100 times in a burst and the best compilation time of these was considered. The Online Tutorial application in JGrafchart 2.1.0 was used since it exercises most features.

The compilation time was 17.3 ms for the old and 39.3 ms for the new implementation. The new implementation performs more checks and has also been rewritten to use a more extensible and maintainable, but slower name lookup. The rewritten lookup alone added 7 ms. The rest of the new implementation thus takes about twice as long as the old implementation.

Interpreted performance has also been analyzed since it is currently the only way to execute JGrafchart applications. The interpreters were also profiled on the Online Tutorial with the scan cycle time reduced to 10 ms. The execution code was instrumented manually and the execution time was accumulated during approximately 5.7 million scan cycles. The average execution time per scan cycle was 0.204 ms for the new implementation and 0.212 ms for the old implementation. The execution performance is practically the same with the new and the old implementation. Better handling of dots and references weigh up the performance loss due to larger overhead and the new lookup. Lookup is involved since dereferencing performs dynamic name lookup during execution.

Conclusions

To extend JGrafchart with HLV would enable more flexible ways to write applications. It is desirable to also keep a pure BV. Since HLV is a superset of BV it is desirable to reuse the BV implementation as a base, meaning the BV implementation needs to be extensible.

ReRAGs and JastAdd are used to make the JGrafchart implementation of the action and condition languages extensible. To confirm extensibility, the `sup` notation was added as an extension to the rewritten implementation of BV. The `sup` extension was straightforward to add and required only a few lines of code.

In summary the pros and cons of the new implementation are:

- The new compiler is extensible.
- Improved maintainability: No files with generated code have to be maintained.
- The different functionalities are defined in separate modules which make them easier to overview.
- Considerably fewer lines of code.
- Improved compilation checks that fetch application errors at compile time.
- Increased robustness: Automatic tests have been created. They can be used to verify that the compiler has not been broken when new functionality is added.
- The compiler takes longer to execute.
- JGrafchart developers must understand attribute grammars to be able to maintain the compilers.

Improved maintainability and increased robustness lead to fewer bugs and better quality. The increased time to compile an application is acceptable and typically not noticeable for the user. On the other hand fewer compiler bugs increase the tool's reliability and added compilation checks and improved features are also of great value. Improved compilation checks mean that errors are found at compile time and consequently less time is needed for debugging.

In conclusion, the new JGrafchart implementation of the textual languages should be ready for implementing HLV as an extension. However, first the compiler of the FC language must also be made extensible.

5.3 Toward Real-time Execution

JGrafchart is an Integrated Development Environment (IDE) with interpreted execution. To execute an application it must first be compiled. The compiler checks if

the application is valid and prepares it for execution by attaching additional data. Applications are executed directly from the IDE in an interpreted manner using the same Java instances as the editor. Thus the editor is reused as a visualizer. To implement visualization is then easier since the editor, visualizer, and interpreter are interlinked. There are, however, several drawbacks which are typically not acceptable in an industrial context:

- The execution has to be performed on hardware with graphics support since the applications can only be executed in the IDE.
- It is not possible to make changes to a running application since the execution must be stopped to go to editing mode.
- The visualization is tied to the computer that executes the application.
- Visualization can only be made on one computer.

Another approach would be to let the compiler prepare a separate representation which contains all data required for execution and to execute the applications separately. With this approach none of the mentioned drawbacks are inherent but the cost is explicit connections between editor, interpreter, and visualizers. To separate the execution engine from the editor also makes it easier to enable real-time execution, something that is difficult to attain when execution is interconnected with visual updates.

In most IDEs the editor, compiler, and runtime are separate. This is a great structure which would bring several advantages and opportunities for JGrafchart. In future versions of JGrafchart (3.0.0 and later) the editor, compiler, and execution engine are separated and the editor still has visualization capabilities. The main goal for the work presented in this section is to make it possible to execute Grafchart application in real-time.

Background

JGrafchart has been developed with the focus to add new features quickly. Many shortcuts have been taken, a strategy that pays off in the short run. However, accumulation of less appropriate dependencies and shortcuts makes it more time consuming to add new features, something that is often referred to as technical debt [60]. For new developers it is also harder to understand how things are connected and why. To split JGrafchart into smaller self-contained parts enforces removal of several less appropriate dependencies and shortcuts.

To split the implementation into stand-alone editor, compiler, and execution engine requires deep understanding of all parts of the JGrafchart implementation. This was obtained by fixing bugs, adding smaller features, and refactoring the code. All existing features were retained but the focus was now instead to create a clear and robust structure where it is easier and less error-prone to add new features. After

rewriting the compilers for the textual languages (Section 5.2) and refactoring the code, the earlier 85,000 lines of code (JGrafchart version 1.5.3.4) were reduced to 45,000 lines of code.

In Section 5.2 a more clean interface between the textual compilers and the editor was created, which was a first step toward a stand-alone compiler. The textual compilers depend on the FC AST which was still interconnected with the editor. To create a stand-alone compiler the FC compiler must also be separated. To make real-time execution possible, execution related code must be stand-alone. This ensures that there are no dependencies on graphical painting operations or other user interactions.

Stand-alone Editor

Each line of code in the whole implementation of JGrafchart was analyzed. Editor, compiler, and execution engine code were moved to separate packages. FC compiler related code was put in a single JastAdd module (see Section 5.2). After this operation, only the editor worked. A new FC AST for the compiler and the execution engine, without the graphics library dependency, had to be created. New explicit interfaces were also needed to get the parts to work together again.

Explicit Interface Design

To design the explicit interfaces between the parts was an iterative process. Several designs have been discarded as trying to reconnect parts gave new insights and ideas on how the design could be improved. Figure 5.18 shows an overview of the final design. The new IDE contains both the editor, compiler, and an internal execution engine. Hence the previous workflow can still be used. It is also possible to use an external Execution component which consists of the execution engine and the compiler. The new execution engine is still an interpreter but both the execution engine and the interpreted AST are different. Another new feature is that it is possible to run the compiler and the execution engine from a command line.

The application XML is used in most interfaces and is the same as when applications are saved to file. An advantage of this is that no additional exchange format is needed. The editor sends the application XML to the compiler and shows compilation messages to the user. To start execution the application XML is sent to the Execution component. The compiler is then used to build the AST, verify that the application is valid, and prepare the execution data. The compiler is also used during execution for dynamic features such as reference lookups and procedure calls. Visualization is driven by both the editor and the execution engine. Operator commands are passed to the execution engine and the execution engine sends information about visual updates to the editor.

With well defined interfaces the parts are interchangeable. It is for example possible to attach a completely different visualizer, which for example just prints out which steps are activated/deactivated or visualizes interesting variable statistics.

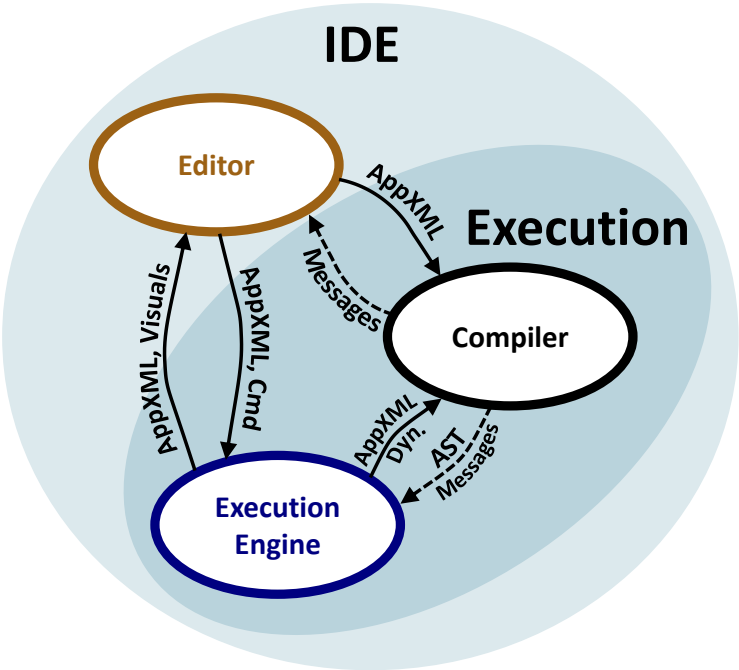


Figure 5.18 This figure shows an overview of the interface design between the stand-alone editor, compiler, and execution engine.

Stand-alone Compiler

Previously, the FC compiler code had been added to the existing editor classes with many interdependencies. For example, the FC compiler directly manipulated the text color of transition conditions for error highlighting. The FC AST was implicitly given by the used graphics library and the AST structure was adapted to fit the graphics library.

Most compiler code had to be rewritten when the editor dependencies were removed. The new compiler was written with ReRAGs to make it extensible. JUnit tests were added to verify the new implementation.

The AST structure was changed to better fit the language. This was also an iterative process, the AST structure was refined as additional pieces of compiler code were added. The new AST structure does not require as many Java interfaces since, for several cases, inheritance could be used instead. For example, the previous implementation required an interface for elements with a name since they extended different classes in the graphics library. Consequently there were 35 implementations of each method in that interface. In the new implementation a super class could be used and thus only a single implementation of each method was required.

One of the methods could be implemented with a Java 8 default method, which was useful for the editor where this Java interface was still also required.

A new XML parser was implemented to build the new AST from the application XML. This was straightforward and required less than 200 lines of code.

Some compiler checks were previously implied by editor code. For example there were no checks for connections between elements as it was assumed that only valid connections could exist in the editor. This had been ensured for most cases but with for example Connection Posts, which can be connected to any element, could be used to create invalid connections. Such checks have been added to the new compiler.

Some less appropriate shortcuts were found when rewriting the compiler code. Some of these would have to be implemented in both the editor and the compiler. Where possible, the code was instead redesigned. Several changes were also made to the application XML format to simplify the compiler implementation and make it more robust. For example, the way to save connections between elements was changed. Previously, a link identified the connected elements by their identifiers only. For many cases it can be inferred which element ports the link should be connected to but in some cases additional information is needed. For a link from a Transition to a Macro Step for example it might either be connected to the in port or to the history port, see Figure 4.6. For such cases, a port identifier was added as a suffix to the identifier. In the new implementation both an element and a port identifier are always used. Less reasoning is then needed to find the right ports. Also, with the old implementation there was a risk that a suffixed identifier would be identical to another, unsuffixed, identifier. This cannot happen with the new implementation.

Several changes were made regarding Connection Posts. Previously, the connection was specified at both ends which meant that there was a 1 to 1 connection and a risk for inconsistencies. Now the connection is only specified at the Connection Post In which means that several Connection Post Ins can be connected to the same Connection Post Out and that there cannot be any inconsistencies. To simplify compilation, Connection Post connections are replaced by links in the compiler before the connection check.

Stand-alone Execution Engine

Application Execution At a first glance, step execution and transition evaluation and firing might seem straightforward to implement. However, with all Grafchart constructs such as Connection Posts, Step Fusion Sets, and abort/resume of Macro Steps as well as combinations of these there are many cases to consider.

For the new execution engine a thorough analysis of all different cases and combinations was made. The old implementation contained several bugs and there were several combinations which did not work. These would be very hard to fix, partly due to some less appropriate shortcuts. The old implementation without the short-

cuts was used as a base for the new implementation where most combinations are supported.

The core idea in the new implementation is that Step activation and deactivation is always performed through one of the Step's ports. For an ordinary Step there is the normal activation through the Step's *in* port and the normal deactivation through the Step's *out* port. There are also many other special cases where activation and deactivation should be handled differently, for example when stopping the application, activation or deactivation through a Step Fusion Set, and abort/resume due to an enclosing Macro Step being aborted/resumed. For the special cases there are internal ports such as *stop* which means deactivation due to stopping the application, *fusionSet* which means activation/deactivation through a Step Fusion Set, *abort/resume* which mean deactivation/activation as a result of a Macro Step abort/resume, and *self* which is used for deactivation/activation of a Macro Step itself only. This core idea led to a more concise implementation.

As an example, consider the application in Figure 5.19 which has a Step Fusion Set for `S1` and `S2`. In the old implementation this is a combination of Step Fusion Sets and Macro Steps which does not work. In the new implementation, when the `b` transition fires, `S1` is activated through its *in* port, just as if it were not part of a Step Fusion Set. Then `S2` is activated through its *fusionSet* port, which means that first `M1` is activated through its *self* port. Similarly, when the `c` transition fires, `M1` is activated through its *in* port and `S2` is activated through its *self* port, just as if it were not part of a Step Fusion Set. Then `S1` is activated through its *fusionSet* port.

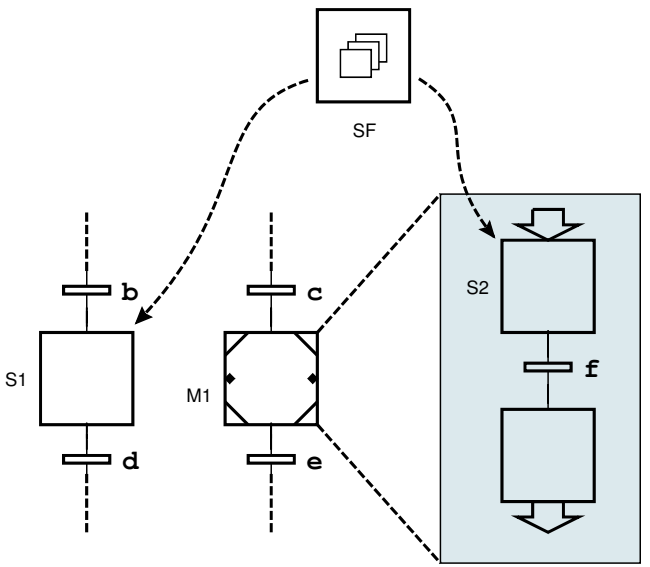


Figure 5.19 An application which did not work with the old implementation.

Flank Detection Conceptually there is a flank if the value of an expression has changed since the previous scan cycle. The old implementation was limited to Boolean variables and the previous and current variable values were considered. For example, if a value was false in the previous scan cycle and assigned true twice it would not count as a positive flank. Similarly, if the last two values were false and true it would count as a positive flank during each scan cycle until a new value was assigned.

In the new implementation, each application element receives a notification after the completion of each scan cycle and expressions use this for proper flank detection.

Concurrency An execution model is the foundation for deterministic execution. For an implementation, concurrency also has to be considered to ensure determinism. Without proper concurrency handling it cannot be guaranteed that the application is executed with consistent data and then the execution not deterministic. Concurrency issues are hard to spot and cause intricate and confusing behavior.

Consider the application in Figure 5.20. It may look harmless but since `b` is a socket input a message with a new value may arrive at any time, for example in between the evaluation of the guard conditions, which might cause a transition conflict unless concurrency has been considered.

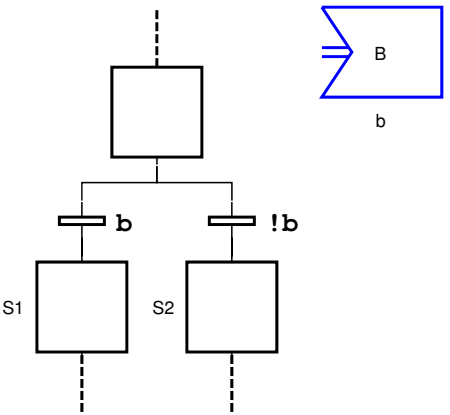


Figure 5.20 If the socket input is updated in between the evaluation of the guard conditions might cause a transition conflict.

If `b` would be a Boolean variable, a transition conflict could still occur since the user may change the value of variables during execution, again for example in between the evaluation of the guard conditions. Transition priorities could be used to resolve the conflict but that only helps for the particular case of conflicting transitions. Another case which has no workaround is consecutive actions which are executed on inconsistent data.

Based on these examples it should be clear that Grafchart needs a concurrency model.

The easiest solution to ensure thread safe interaction between Grafchart applications is serialized execution, that is, only one Grafchart application at a time may execute its scan cycle. Also, to update asynchronous inputs or apply changes made by the user must only be allowed when no Grafchart application is executing. A sensible approach is to make the changes visible during the read input phase of the execution model, as this phase is already dedicated to let externals affect the application. This is the strategy that was used in the new execution engine.

Performance An early version of the new execution engine was profiled to get an initial estimate of its expected performance. A small application that only uses constructs which were available in that version is shown in Figure 5.21.

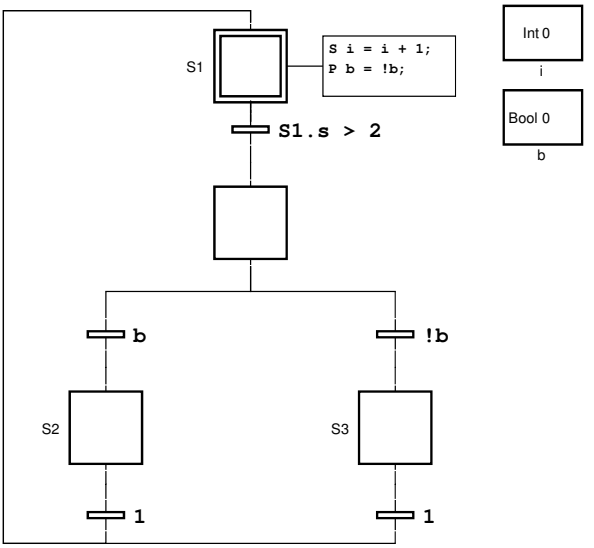


Figure 5.21 The profiled application. Its sole purpose is to exercise the basic constructs.

The interpreted execution was profiled by manually instrumenting the execution code and removing the sleep between the scan cycles. The execution time was then accumulated over 100 million scan cycles. The average execution time per scan cycle was 500 ns for the new implementation and 4,000 ns for the old implementation. The execution code was barely changed and the main difference is that graphical painting had been removed. If painting is removed in the old implementation the average execution time per scan cycle was also 500 ns. Provided that nothing else is changed, the performance should thus be an order of magnitude faster. Note, however, that it is only faster, there are still no timing guarantees.

Editor/Compiler Interaction

In graphical programming languages compilation messages cannot refer to problems by file and line numbers. In JGrafchart, compilation issues are highlighted in the editor. For example, the text color becomes red for incorrect transition conditions.

In the previous implementation the error highlight feature was limited to a few elements and only worked under certain conditions. To retain and enhance this feature was desirable. To use direct references is not possible with a stand-alone editor and compiler. Instead unique identifiers for the elements are included in each compilation message. Graphically connectable elements already had unique identifiers which were used to store the graphical connections. All other elements were extended with similar unique identifiers to make it possible to refer to them as well.

Figure 5.22 and Figure 5.23 show the error highlighting of an application in the previous and the new implementation respectively. Compiling the application results in a warning about an uncorrected transition and an error in the actions of the step.

Note that the previous implementation changed the name of the unnamed step to `#0` to be able to refer to the unnamed step. It also gives an additional compilation message to specify that the other compilation message is related to this step. If the actions for the step were shown they would be colored red. The warning can only state that there is an unconnected transitions, not which one, since transitions do not have names.

In the new implementation no modifications are made to the application. The step that causes the error and the transition that causes the warning are clearly highlighted. Another new feature is that clicking on a compilation message now selects the corresponding element in the editor. This is particularly useful for unnamed elements, like transitions, and for elements which are currently not visible in the editor. Clicking on a compilation message for an element in another workspace for example opens that workspace and selects the element.

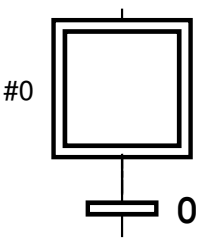


Figure 5.22 Error highlighting with the previous implementation. Warnings are not highlighted at all and the error in the step actions is not highlighted since the actions for the step are hidden.

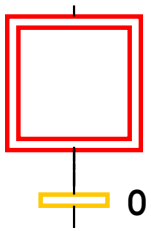


Figure 5.23 Highlighting with the new implementation. The step with an error is colored red and the unconnected transition is colored orange.

Compiler/Execution Engine Interaction

To verify that the application does not contain any errors and to obtain and prepare the AST to interpret, the execution engine first sends the application XML to the compiler.

A JastAdd module for FC language execution has been created. The module basically creates execution nodes and tie these to the FC AST, see Figure 5.24. The interpreted execution of the FC AST is implemented in the execution tree. The main reason to have it in a separate tree is to make the execution engine easier to debug. To create execution engine extensions, the execution module can be refined to create other execution nodes. The interpreted execution of the action and condition ASTs are still implemented as interpreter modules as described in Section 5.2.

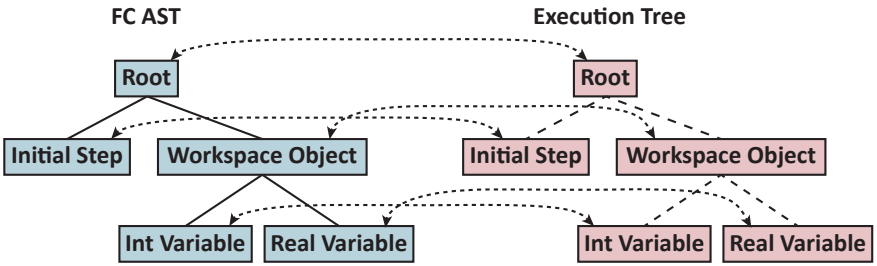


Figure 5.24 The compiler FC AST and the parallel execution tree. The execution tree nodes do not reference each other, instead the execution tree is traversed via the FC AST.

Reference evaluation and procedure calls make use of the compiler at runtime as well. This is a dependency that was kept from the previous implementation and must be removed for code generation for hard real-time control to be possible. Most use cases can be replaced by lookup calls but for the `spawn` method for example this is not possible. `spawn` is a method that spawns a call to a Procedure. The argument to the method is a string with the call parameters. The string must be compiled during execution which requires a lightweight compiler.

Execution Engine/Visualizer Interaction

Visualizers register themselves with the execution engine to receive visual updates, such as when a step is activated, when a variable or I/O gets a new value, when a rectangle is moved, or when the application execution is stopped. The unique identifiers that were added for compiler feedback are also used as references for visual updates. However, as these identifiers are only unique within an application a unique execution instance id is also supplied.

At the end of the execution of each scan cycle a tick message is sent. It informs the visualizer that one scan cycle has passed and that the variable values are consistent, which can for example be used to create a visualizer that logs execution data for application debugging.

Commands from the visualizer, for example to change a variable value, are sent in a similar manner to the execution engine. Elements are identified with the same identifiers as for visual updates. As discussed in the concurrency section, commands are only processed in between the execution of scan cycles.

A bonus feature with a stand-alone execution engine is that it is possible for JGrafchart developers to use the editor while debugging the execution and to debug the visualization without interrupting the execution.

In the IDE the calls between the visualizer and the execution engine are Java method calls. Only primitive types and strings are allowed as parameters to the methods to simplify use of remote execution engines as well as support language independence. It is possible to connect to a remote execution engine with LabComm [61] over a TCP socket. LabComm was chosen since it has minimum communication overhead and is supported in several languages. LabComm is described further in Section 6.5. Each update and command are implemented as a separate LabComm sample. For example, notification about when a step is activated or deactivated has the sample specification shown in Figure 5.25.

```
sample struct {
  int appId;
  string nodeId;
  boolean active;
} setStepActive;
```

Figure 5.25 The LabComm sample specification to notify about when a step is activated or deactivated.

Evaluation

To reuse the application XML for some of the interactions is sufficient since it contains all information about the application and means that no additional exchange formats are required. A running application also contains visualization data and can receive operator commands. These are sent separately.

With all compilers implemented with ReRAGs it is possible to add extensions, for example to generate code for hard real-time control. Application refactoring like renaming a JGrafchart variable should also be possible to implement by using the new AST. To be able to edit an application while a previous version of it is running is also a necessary first step toward making it possible to update a running application. With a stand-alone execution engine, distributed applications can also be implemented more easily.

Conclusions

There are several advantages and opportunities with the new structure:

- The execution performance is improved considerably.
- The interfaces between the parts are clear which means that they are interchangeable.
- It is easier to add new features and to do so without breaking anything as several less appropriate dependencies and shortcuts have been removed.
- The compiler is stand-alone and implemented with ReRAGs and should thus be extensible.
- Improved maintainability as the interaction between the different parts are explicit. It is thus easier to focus on one part at a time.

The performance measurements indicate that the new interpreter is an order of magnitude faster. The improved error highlighting is an example of where clear interfaces between the parts makes it easier to add new features. The whole JGrafchart compiler is now implemented with ReRAGs and should be ready for adding extensions, for example HLV or code generation for real-time execution. With separate parts it is easier to get an overview of each part which improves maintainability and robustness.

6

Grafchart for Factory Integration

This chapter addresses the challenge of integration and utilization of devices and software. A new way to do SOA service orchestration with DPWS and OPC UA as well as a new way to do SOA 2.0 service choreography are presented. The primary focus has been on manufacturing but the concepts are useful for process automation and robotics as well. Next, support for the high performance communication protocol LabComm [61] is added. Finally, it is investigated how Grafchart can be connected to FMI for Co-Simulation, a standardized co-simulation environment, to be able to co-simulate Grafchart applications with simulation or modeling tools. This addresses the shorter time to market trend by making it possible to perform simulations before commissioning. The high performance communication and simulation support are useful for both process automation, manufacturing, and robotics. High performance communication is, however, particularly useful for robotics where the execution rate is higher.

6.1 Service Technologies

Most SOA tools are tailored for business processes. In [62] several tools were evaluated and DPWS [63] was the tool deemed best suited for SOA-AT. DPWS is based on existing web service standards and defines a minimal set of implementation constraints to enable web services on resource constrained devices [64].

DPWS has been shown to be useful for industrial automation [65], but OPC UA [66], which is feature backward compatible with classic OPC, the current de facto standard for interoperability in the automation domain, is likely to spread faster [67].

As concluded in [65], both OPC UA and DPWS are required to to cope with all industrial requirements on device level.

6.2 Devices Profile for Web Services

An outcome of the SIRENA project [68] is an open source DPWS implementation targeted at embedded devices [69, 70], thus pushing the web services technology used at business level down to device level.

The work presented in this section enables automation to be built in a service oriented way by making it possible to use JGrafchart as a generic DPWS client [71, 72]. It has been carried out in collaboration with DFKI in Kaiserslautern, Germany.

Background

DPWS Plain web services are complicated to use since for each service the desired web service standard extensions may or may not be supported. DPWS defines a minimal set of mandatory extensions which are always available.

The hierarchy of DPWS is shown in Figure 6.1. At the root is a DPWS *device*. Below the *device* are *services*, *portTypes*, and *operations* which is the same structure as for ordinary web services. A *service* encapsulates a specific functionality. In SOA-AT this is the functionality provided by a field device, for example the features of a motor. A *device* hosts *services*, for example it could host *services* for a group of co-located motors. An *operation* corresponds to an action that a *service* is able to perform, for example a motor could have the *operation* `setTorque(t)`. The *portType* is used to group *operations*, for example a motor could have the *portTypes* `torqueControl` and `rotationSpeedControl`.

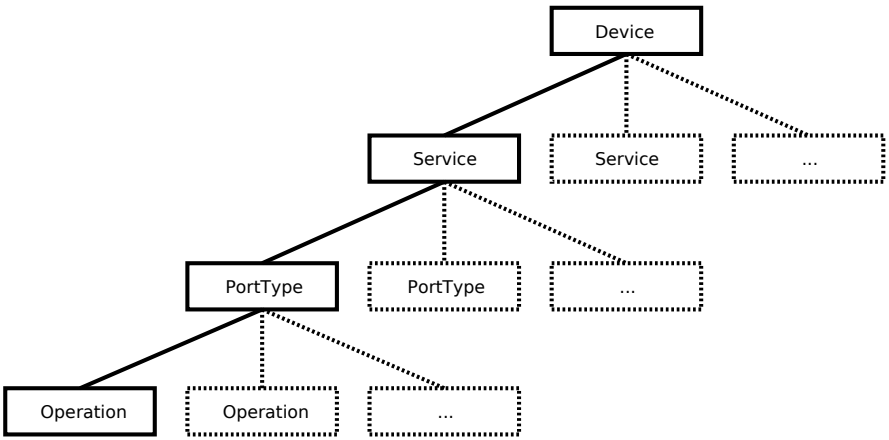


Figure 6.1 The DPWS hierarchy. A DPWS *device* hosts *services*, *services* contain *portTypes*, and *portTypes* contain *operations*.

Figure 6.2 shows an overview of the DPWS stack [73]. At the base level is the IP protocol for the actual communication. Both IPv4 and IPv6 may be used and IP

multicast is used for discovery. On top of this either HTTP/TCP or UDP are used. XML is used as the underlying message structure. On top of this SOAP-over-UDP and SOAP define the message exchange protocol and Web Services Description Language (WSDL) is used to define the messages. Finally, additional web service extensions like WS-Discovery, WS-Addressing, WS-MetadataExchange, and WS-Eventing are included and mandatory in DPWS.

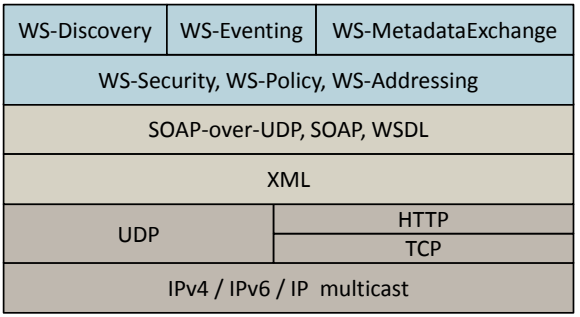


Figure 6.2 Overview of the DPWS stack.

WS-Discovery The WS-Discovery extension makes it possible to find devices dynamically on the local network, see Figure 6.3. To find devices a client multicasts a *Probe* message. Devices on the local network should receive this message and respond with a *Probe Match* message to the client. Devices also multicast *Hello* and *Bye* messages when they join and leave the network. Ideally, clients should only need to send a *Probe* message when they join the network.

With WS-Discovery it is possible to find and connect to a device without any prior knowledge about it. However, WS-Discovery only works on the local network and since multicast uses UDP it is inherently not reliable. For devices not found with WS-Discovery the connection can be set up manually.

WSDL Services are specified with the WSDL language. Each service is defined by its WSDL description, also referred to as its WSDL. It defines all portTypes and operations supported by the service. It also defines the message structure of all messages and specifies which messages are used for the operations. In other words, the WSDL contains all information required to interact with a connected service.

WSDL specifies four types of operations, namely *one-way*, *request-response*, *solicit-response*, and *notification*. *One-way* and *request-response* operations are invoked by the client by sending the corresponding message. For *request-response* operations the service also returns a corresponding response message. Symmetrically, *notification* and *solicit-response* operations are invoked by the service and for *solicit-response* operations the client returns a response message.

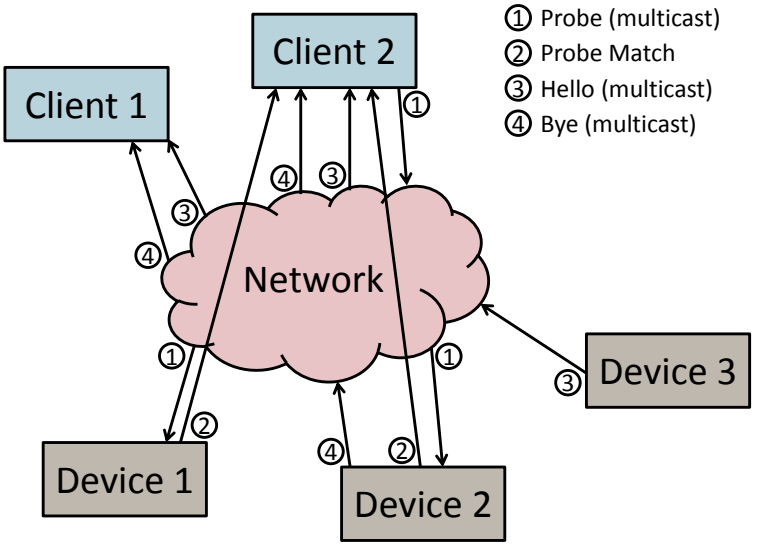


Figure 6.3 Initially Client 1, Device 1, and Device 2 are connected to the network. Client 2 joins and multicasts a *Probe* message. Device 1 and 2 receive this message and each respond with a *Probe Match* message to Client 2. Now Device 3 joins the network and multicasts a *Hello* message which is received by Client 1 and Client 2. Finally Device 2 is about to leave the network and multicasts a *Bye* message. It is received by Client 1 and Client 2.

WS-MetadataExchange DPWS devices must expose various metadata such as manufacturer and model name which is useful to retrieve additional details about a discovered device. The WS-MetadataExchange extension also requires services to expose their WSDL.

WS-Addressing For a client to ensure that the correct device is used, an unambiguous way to identify the device is required. The WS-Addressing extension requires that each device has a unique identifier. It is then possible to recognize a previously used device and be sure that this is the exact same device.

WS-Eventing Events are often preferred over polling. The WS-Eventing extension provides support for eventing.

Summary Figure 6.4 shows a sequence diagram where WS-Discovery is used to detect a device, that is, the client sends a Probe message and receives a Probe Match response from the device. WS-Addressing metadata is then fetched to ensure that this is the correct device. Then hosted services are listed and WS-MetadataExchange is used to fetch the desired service's WSDL. Then the one-way

operation `oneWayOp` is called, followed by a call to the request-response operation `reqRespOp`. Notice that the call to `oneWayOp` only consists of one message while `reqRespOp` consists of one message in each direction. After this a subscription is created and finally, when an event happens, a notification operation named `myEvent` is received from the service.

By relying on the DPWS mandatory extensions WS-Discovery, WS-Addressing, and WS-MetadataExchange it is possible to create a generic client that can be used for service orchestration, as shown in Figure 6.4. However, no such tool existed. JGrafchart was considered a suitable candidate for being extended with generic DPWS support. It is based on SFC which is widely used in industrial automation and, compared to business process tools, has a higher chance of being accepted by the automation community.

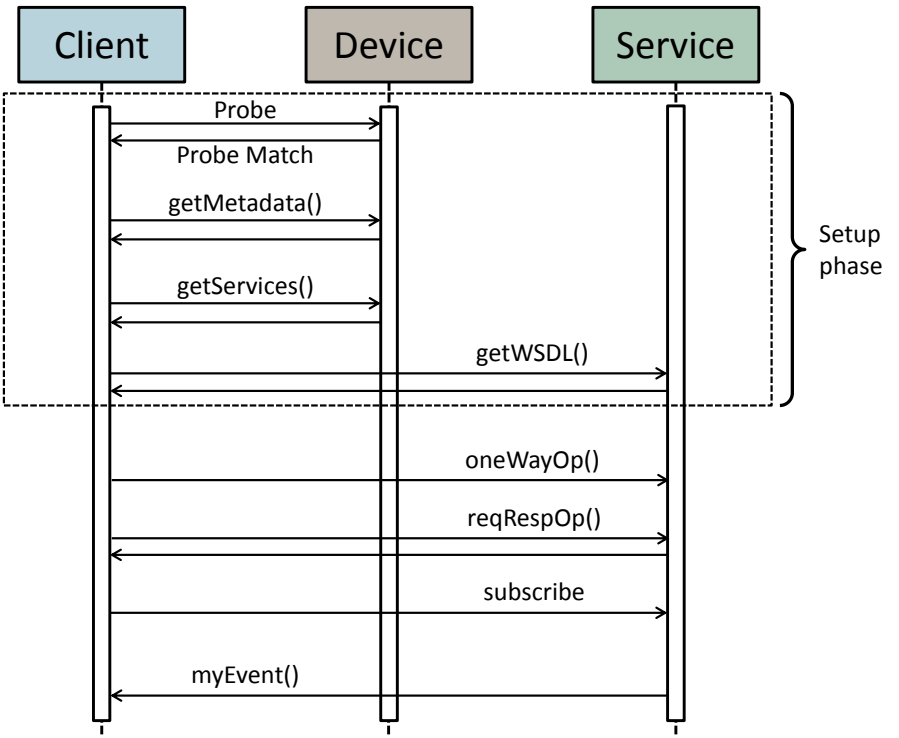


Figure 6.4 A sequence diagram for the interaction between a generic client, a device, and a service hosted by the device. First the device is discovered and the service's WSDL is fetched. Then operations in the service are called, a subscription is created, and finally a notification is received.

DPWS in JGrafchart

As shown in Figure 6.2, DPWS communication uses XML messages and thus requires to possibility to send and receive strings. The previously existing I/O mentioned in Section 4.4 are Digital/Analog In/Out and Socket I/O. Digital/Analog In/Out are insufficient as they only support Boolean and float values. Socket I/O on the other hand can send and receive any strings without newline characters. Since newline characters are considered whitespace in XML this is not a problem. The first prototype to integrate DPWS in JGrafchart was implemented with Socket I/O.

The Socket I/O Prototype A TCP/IP server was implemented to translate assignments to socket outputs in JGrafchart to DPWS operation call messages, as well as DPWS response messages to socket inputs in JGrafchart, see Figure 6.5. Each DPWS operation used requires its own translation code in the TCP/IP server. Some special socket inputs and outputs as well as some extra code to detect event arrival were also required for subscriptions and event notifications.

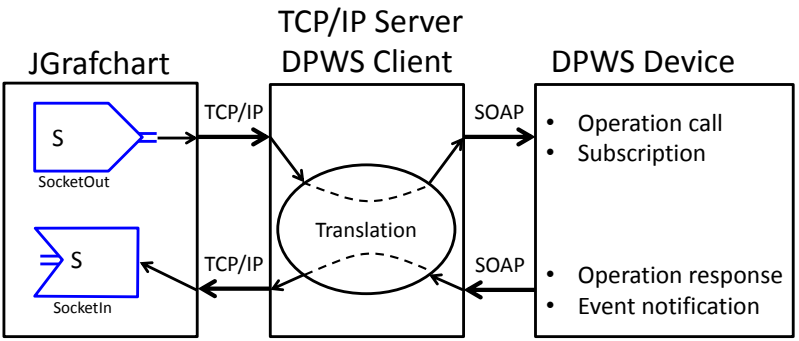


Figure 6.5 Overview of the Socket I/O prototype.

Previously, JGrafchart would only send a message to the server if the value of the assigned socket output changed. Since the assignments in the prototype correspond to DPWS operation calls this means that consecutive calls with identical arguments were lost. A send mode setting has been added to workspaces and individual socket outputs to control when messages are sent. With the default send mode *changed*, a message is only sent when the value changes, just like before. With the new send mode *assigned*, a message is sent each time the socket output is assigned. It is also possible to *inherit* the setting from the enclosing workspace. With the prototype the entire JGrafchart application should be configured to use socket send mode *assigned*.

A possible improvement to this prototype would be to implement a more generic translation in the server, thus reducing the amount of specific code for each operation. Another improvement would be a JGrafchart helper library for subscriptions and event notifications.

A major problem with the prototype is that it is hard to make calls to request-response operations synchronous. When JGrafchart has written to the socket buffer related to a request-response operation, it does not know that it should wait for the update of a specific socket input before resuming execution. A request-response call using the Socket I/O prototype is shown in Figure 6.6. To call the same operation from several parts of the application at the same time is also complicated. There are also the aesthetical issues that operation calls are represented by assignments and that returned values are fetched from separate socket inputs.

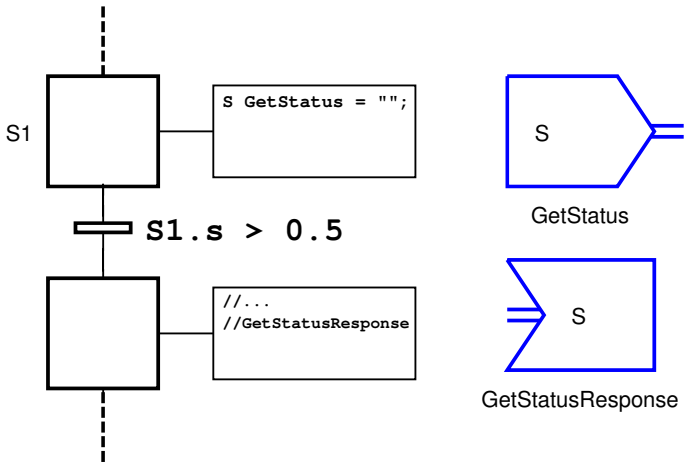


Figure 6.6 A request-response call with the Socket I/O prototype. The call is invoked by the assignment to `GetStatus`. The TCP/IP Server translates this into a DPWS call. When the TCP/IP Server receives the response message it forwards it to `GetStatusResponse`. In the application the response is assumed to be available after 0.5 seconds. It would be possible to use extra socket I/O to signal when the response is available but that would be even more complicated.

Integrated Generic DPWS A generic DPWS implementation has been integrated directly into JGrafchart using the DPWS4J toolkit [74]. With WS-Discovery, existing devices and device startups and shutdowns are automatically detected. With WS-MetadataExchange, each service's WSDL is obtained. It is possible to browse available devices, services, and operations in JGrafchart, see Figure 6.7. Device metadata and WSDL documentation are also displayed.

To call DPWS operations a new I/O element in JGrafchart called *DPWS Object* is bound to a portType, see Figure 6.8. The unique identifiers provided by WS-Addressing are stored to later restore the binding automatically, for example when a saved JGrafchart application is opened or when a device joins the network.

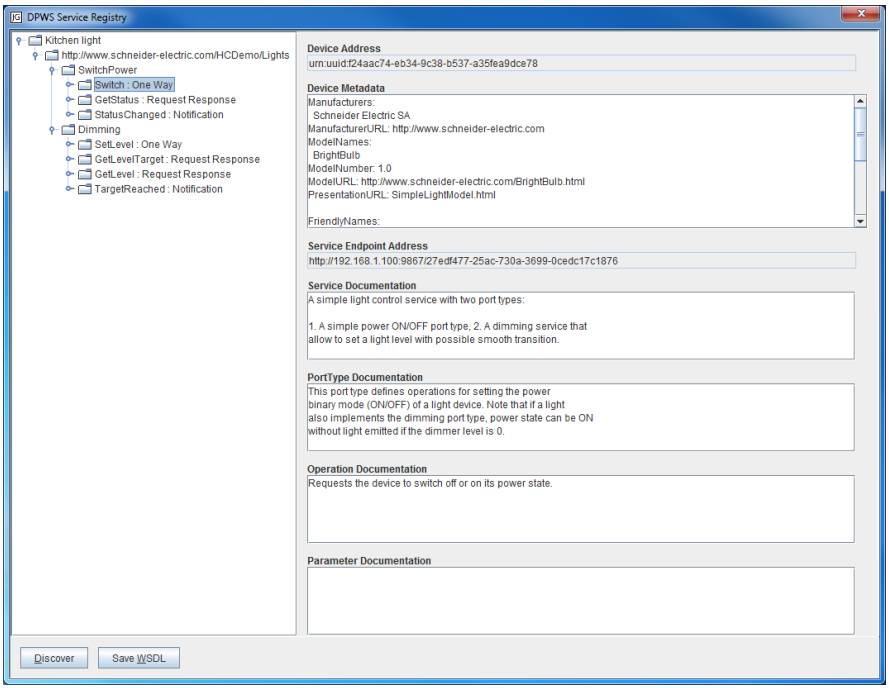


Figure 6.7 The service explorer in JGrafchart shows details about the available devices. The WSDL can also be saved, which is useful for example if the documentation is not sufficient.

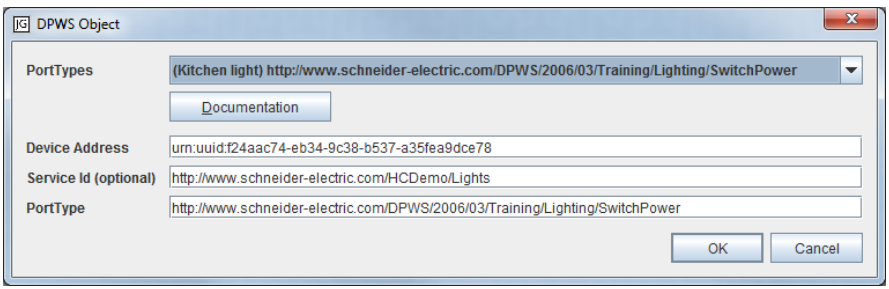


Figure 6.8 The configuration dialog for the new *DPWS Object*. It is bound to a specific portType of a specific service in a specific device.

To Call DPWS Operations JGrafchart supports all operation types except solicit-response. DPWS operation calls look like other method calls in JGrafchart, see Figure 6.9. Here the DPWS Object `myDPWSObj` is bound to the portType `SwitchPower`. In the Grafchart application, a 10 minute subscription is first created. Then the one-way operation `Switch` is called. The application then waits for a `StatusChanged` notification. Finally the request-response operation `GetStatus` is called and its response is stored in the variable `newStatus`. In this example the new built-in functions `dpwsSubscribe` and `dpwsHasEvent` are used for eventing. There are also new built-in functions for XML handling and fault detection.

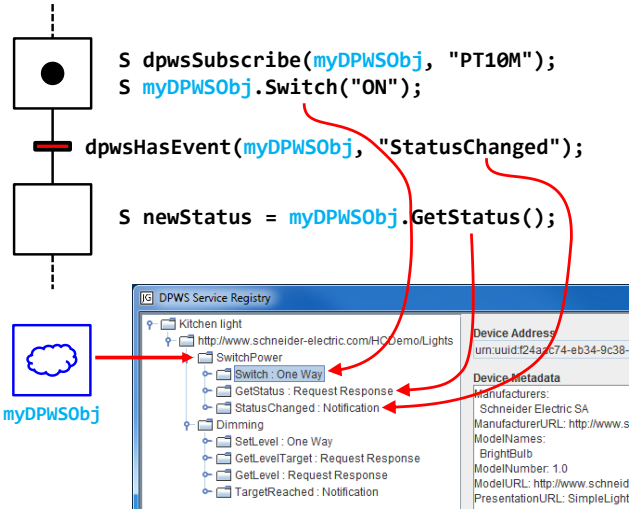


Figure 6.9 How to use the integrated DPWS feature in JGrafchart. The second argument to `dpwsSubscribe` is an ISO 8601 [75] duration string, in this case `"PT10M"` which means 10 minutes.

Calls to request-response operations are synchronous which means that when a request-response operation is called, execution pauses until the response message is received. The behavior is thus more deterministic and it is easier to reason about the execution. It also means that the execution is delayed if it an operation takes a long time to finish. Also, if either message is lost, the execution will freeze indefinitely. This should be improved in future versions of JGrafchart.

Compiler Aspects The WSDL contains all data needed to check if an operation call is valid. Some DPWS operations have content while others do not. For example, `GetStatus` in Figure 6.9 has no content while `Switch` has content that tells if it is a switch on or switch off request. In JGrafchart a call has 0 or 1 parameters corresponding to no content and content respectively and the compiler checks that a call has the correct number of parameters. The actual content is often built dynami-

cally and cannot be checked at compile time. For this kind of errors runtime SOAP fault handling is used. Applications can check for and handle these faults with the new built-in function `dpwsHasFault` and `dpwsGetFault`.

Devices might not be present during development or at compile time. The editor, compiler, and execution engine may be on separate local networks and the devices might then only be available to the execution engine. Devices could also be optional and only present at certain times. In the editor, bindings can be specified both with and without the device present. If a device is unavailable at compile time, the compiler will just warn about not being able to perform these compilation checks. If a device is unavailable during execution `dpwsHasFault` returns true and the message returned by `dpwsGetFault` states that the device is unavailable.

Evaluation

SmartFactory^{KL} is a manufacturer-independent research platform [4] where for example demonstrators are created to evaluate new ideas. The DPWS integration was evaluated on a SOA demonstrator at SmartFactory^{KL}.

The demonstrator consists of two stations, namely the filling station and the quality control station. At the filling station bins are filled with pills. The quality control station checks that bins contain the correct number of pills. The demonstrator uses real industrial field devices encapsulated as DPWS devices. It consists of a conveyor belt that transports carriers with bins to the stations. The bins have RFID tags where various information about the product is stored, for example the number of pills, if the bin has been filled, and the result of the quality control.

In the evaluation the quality control station was considered, see Figure 6.10. It consists of five field devices: a sensor that detects the arrival of carriers, a stopper that can stop the carriers, a sensor to check if there is a bin on the carrier, an RFID device to read/write from/to the RFID tag on the bin, and a camera to take a top view image of the contents of the bin to count the number of pills.

The coordination sequence for the station can be modeled as in Figure 6.11. Based on the model a JGrafchart application to coordinate the station was implemented, see Figure 6.12. As some states in the model have a straight flow they could be implemented in the same JGrafchart step. The step named `CheckBinRFID` and `QC` in JGrafchart correspond to model states (3)-(4) and (5)-(7) respectively. The application gives the desired behavior and is reliable.

The new built-in XML utility functions have been used to simplify the code. For example, `xmlFetch` is used to obtain a derived value from an XML string. The camera's `count` operation returns a sequence of value elements where each element describes the number of pills of a specific color. The total number of pills is fetched with `xmlFetch(resp, "value", "sum")` where `resp` is the returned string, `"value"` is an XPath [76] that selects all elements with the tag name `value`, and `"sum"` is a built-in handler that calculates the arithmetic sum of the selected elements' texts interpreted as numbers.

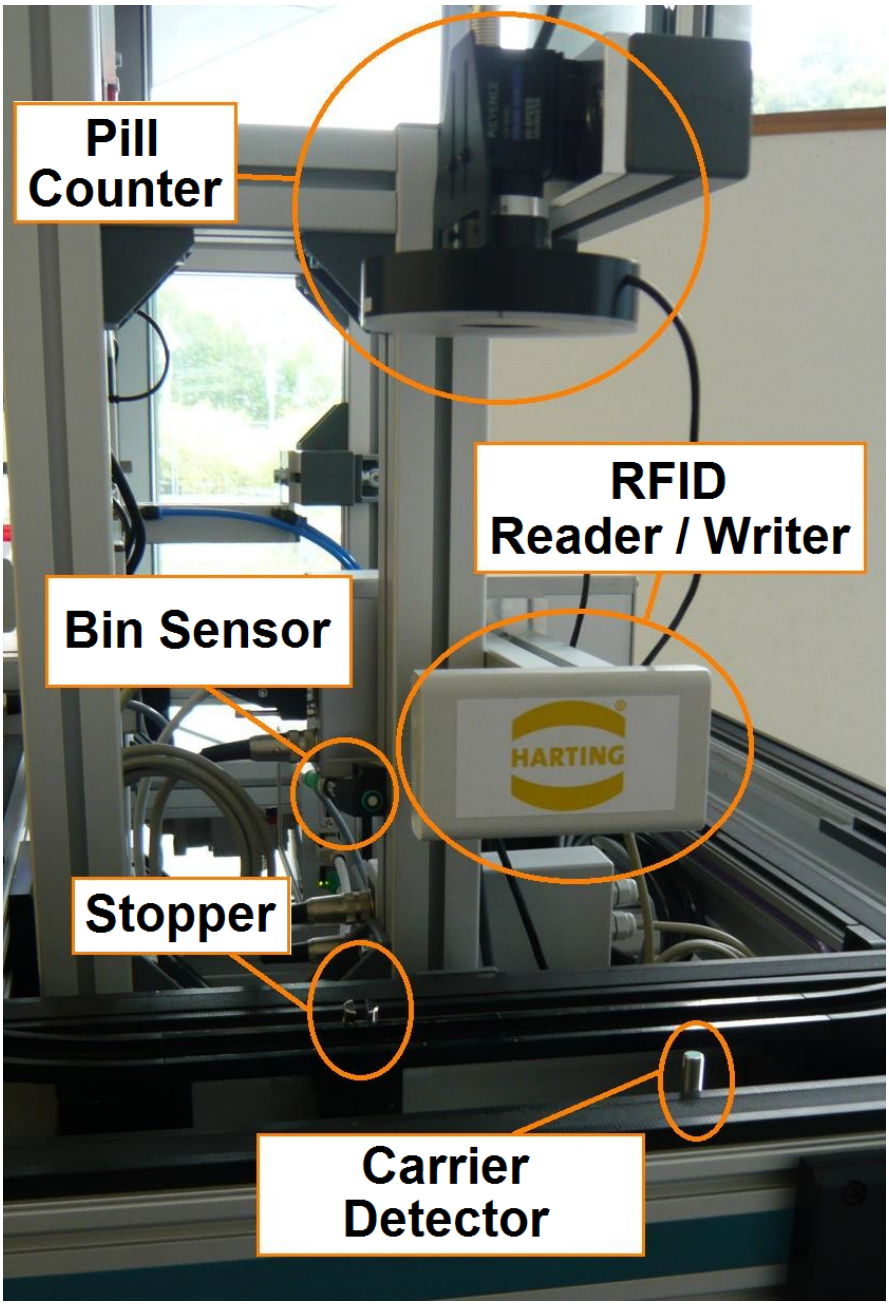


Figure 6.10 The quality control station of the SOA demonstrator.

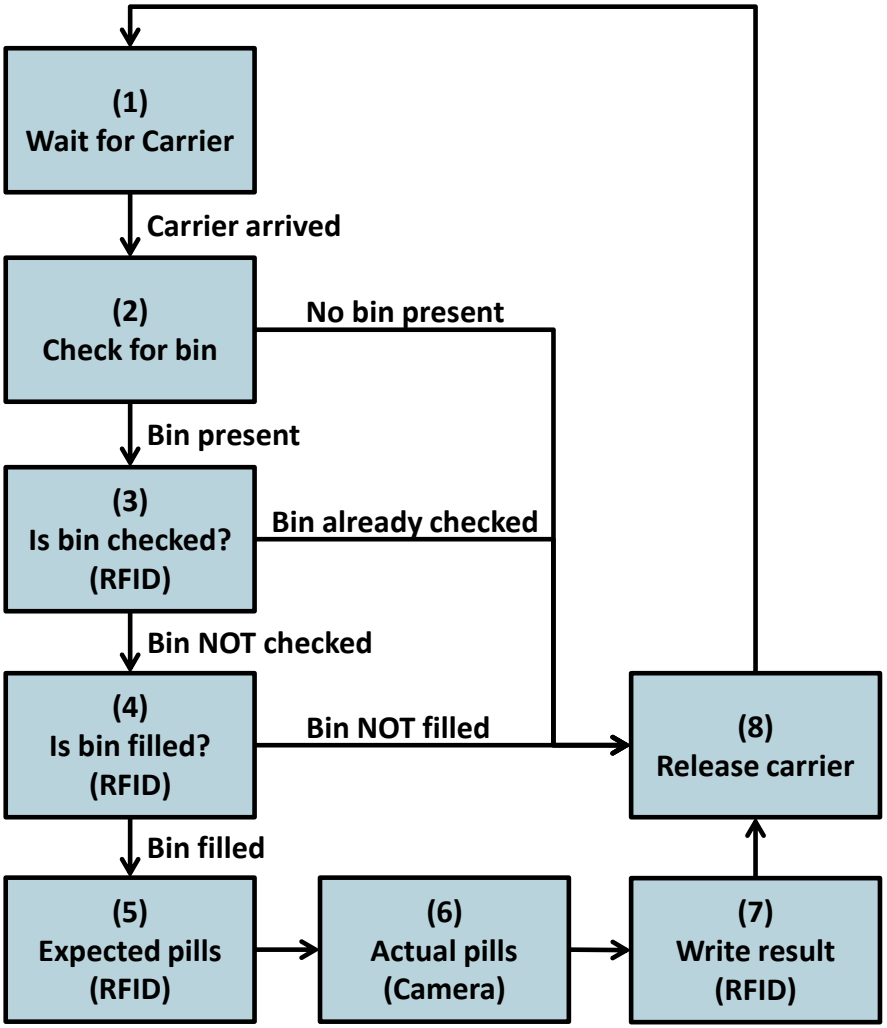


Figure 6.11 A conceptual coordination sequence for the demonstrator in Figure 6.10 where (1) is the initial state.

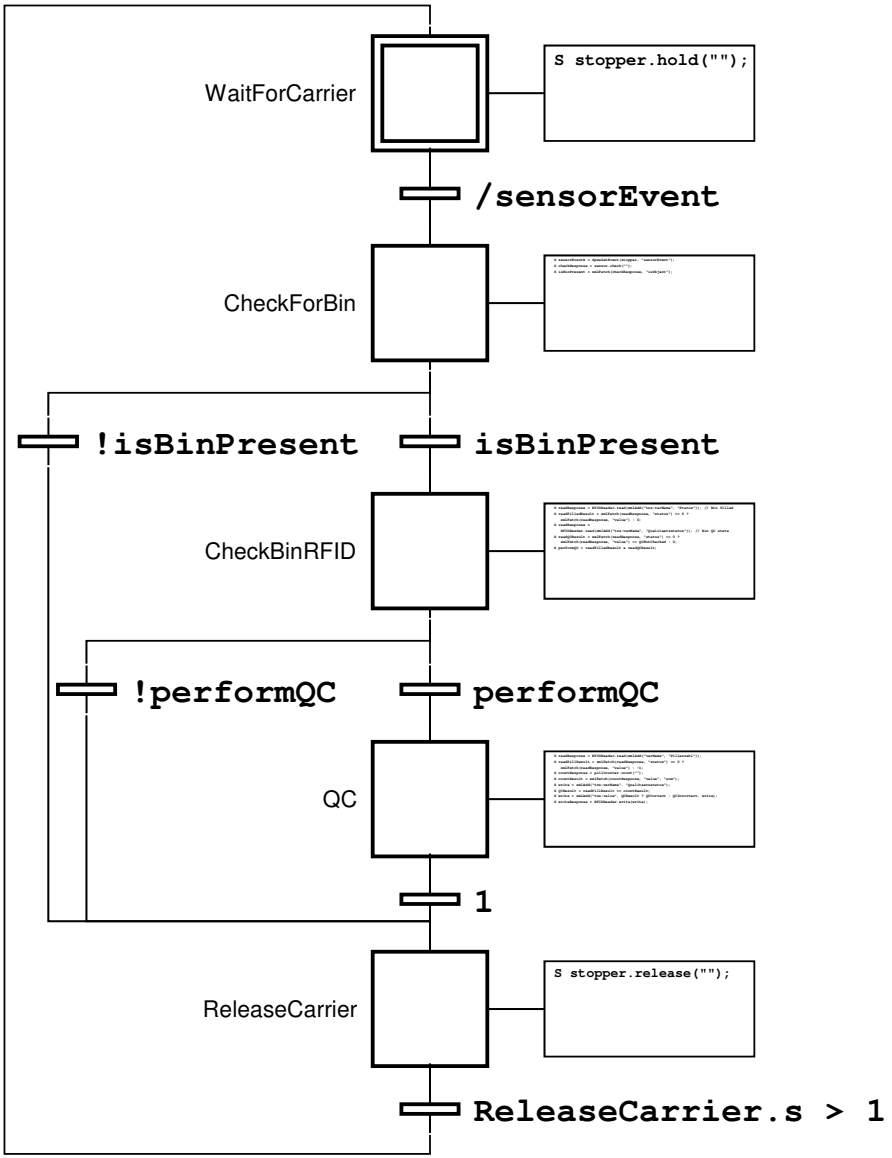


Figure 6.12 A service orchestration for the demonstrator in Figure 6.10 implemented with the integrated generic DPWS feature of JGrafchart. The actions are shown to give an idea of the amount of code required, they are not intended to be readable in the printed version.

Conclusions

SOA is a design methodology which can improve flexibility and reusability for industrial automation systems. With SOA, vertical integration is simplified since services can be used directly at any level in Figure 2.1.

It was shown that SOA works very well on a demonstrator with real industrial field devices. The implementation turned out to be practically identical to the conceptual coordination sequence. This means that it might as well have been modeled in JGrafchart and then implemented by adding the actions and transition conditions. Thanks to the successful use in the SOA demonstrator, the DPWS support has been used by SmartFactory^{KL} to implement more demonstrators, for example the automated assembly station of the keyfinder production line [77].

As a result of this work there is a generic tool for DPWS service orchestration. This enables anyone to try out SOA-AT and experience the advantages.

6.3 OPC Unified Architecture

In this section it is presented how SOA service orchestration for control of OPC UA services can be done with Grafchart. Generic support to use OPC UA servers has been integrated into JGrafchart [78] and as an example it has been used to control a physical process which has been modeled, encapsulated, and exposed as an OPC UA server by wrapping it with an Ethernet capable microcontroller. In other words the physical process has been retrofitted to become a service capable device.

OPC UA Standard

OPC UA is a recent standard [66] which incorporates all the features from the three classic OPC specifications OPC Data Access (OPC DA), OPC Alarm and Events (OPC AE), and OPC Historical Data Access (OPC HDA) which are all based on the Microsoft proprietary COM/DCOM technology. OPC UA on the other hand is platform and language independent and also incorporates SOA features [79]. OPC UA was developed with the goal to offer robust communication in distributed systems with high security, connecting everything from small embedded field devices to large enterprise systems.

Information Model OPC UA offers more flexible modeling possibilities than classic OPC in which only simple data could be represented [79]. Types and instances in a specific device are defined in its address space. OPC UA data modeling uses the concepts of nodes and references. Types and instances are represented by nodes and the relationship between two nodes are represented by references. Each node has a number of node attributes, some mandatory and some optional. The data type for all attributes except the value are defined by the OPC UA specification. The specification defines a fixed number of built-in data types such as the common Double, Int32, Boolean, and String as well as the OPC UA specific NodeId and

QualifiedName. Each node has a NodeId with a unique id number that is used for example to find a specific node in the address space.

Eight node classes are defined in the OPC UA specification and the three most important ones are variables, methods, and objects [79]. Variables represent values on the server. Methods are functions/operations which can be called and they may have multiple input and output parameters. Objects consist of a set of variables, methods, and objects and they and are typically used to represent real world objects. Each node also has a set of references to other nodes in the server's address space.

Unlike classic OPC where only data types information can be expressed, it is possible to express type information in OPC UA. There are two node classes for type definitions, namely object type and variable type [79]. It is possible to create extended (sub-)types based on another type. OPC UA thus allows type programming, that is, object or variable types can be defined and then instantiated to create concrete instances [79]. For example, a motor type can be created and there could then be five instances of the motor type which correspond to five physical motors. The motor instances have the exact same structure (variables, methods, and objects) as the motor type and can thus be used in the exact same way.

Discovery Unlike DPWS, OPC UA does not support discovery of services on the network without any prior knowledge. On the other hand, OPC UA discovery is reliable and works both across networks boundaries and over the Internet. A set of standard services for discovery is defined as the Discovery Service Set [79]. In the simplest setting, there is one discover server that all OPC UA servers and clients know about. Servers register themselves to the discovery server, which maintains a list of servers. The list can be fetched by OPC UA clients. OPC UA servers must register themselves periodically and notify when they intend to go offline so that the discovery server knows which OPC UA servers are currently available.

Client-Server Interaction The server decides the desired security level [79] and the client must meet the security requirements to be able to connect. All information modeling is done on the server side and connected clients can obtain this information by browsing the server's address space.

The client can access a variable value in the server in two ways: by subscribing to it (recommended) or by explicitly asking for it. The client can also specify how often values should be received in a subscription.

OPC UA has support for two transport protocols namely UA Binary and UA XML [80]. The binary TCP protocol offers best performance and interoperability whereas SOAP is for web service interoperability and is more widely supported by various tools [81]. Figure 6.13 shows an overview of the OPC UA stack. At the bottom is the transport protocol layer. Which transport protocol to use is determined by the URI scheme name (http, https, or opc.tcp). Above this is the security protocol layer which takes care of message signing and encryption. Next is the data encoding layer which determines the structure, encoding, and serialization of messages. In the application layer at the top are the user applications, implemented in the desired

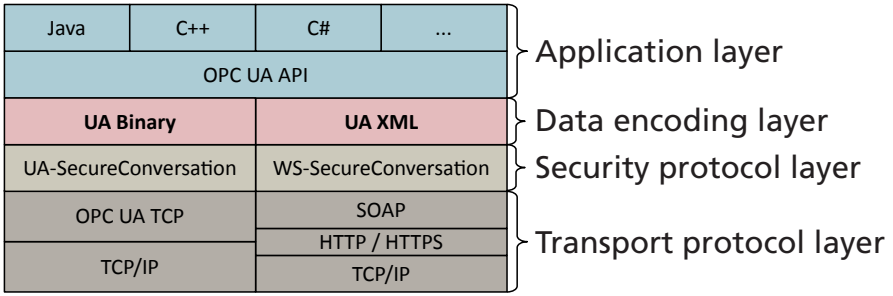


Figure 6.13 An overview of the OPC UA stack.

programming language, typically using an OPC UA toolkit for this language.

OPC UA supports server state monitoring which means that connected clients can choose to be notified about the current server state when the state changes. It is for example possible to detect when the server is shut down, if there is a communication problem, if the server is not functioning, and when the server is healthy and back online again. It can be used for example for automatic client error handling and reconnect.

Backward Compatibility One goal with the new OPC UA standard was to make it platform independent and thus it was necessary to remove dependencies to proprietary technologies such as DCOM which classic OPC is based on. Thus it is not possible to directly connect an OPC UA client to a classic OPC server or a classic OPC client to an OPC UA server. However, OPC UA is backward compatible feature-wise, which means that all features supported by classic OPC are also supported by OPC UA. It is thus possible to convert calls between OPC UA and classic OPC [82]. In this way it is possible to connect OPC UA clients to classic OPC servers and classic OPC clients to OPC UA servers.

OPC UA Integration in JGrafchart

OPC DA is currently used the most in industry. Among products that use classic OPC, 99% implement OPC DA while OPC AE and OPC HDA are mostly implemented in addition to OPC DA [79]. This work is thus mainly focused on support for OPC UA data access. OPC UA methods are also considered as this is a key part of web service interoperability.

JGrafchart has been extended with integrated support to connect to and interact with OPC UA servers over the UA Binary transport protocol. All OPC UA data types needed for controlling purposes are supported. A new I/O element called *OPC UA Object* has been added to JGrafchart to make it possible to connect JGrafchart applications to OPC UA servers. Each *OPC UA Object* is bound to a variable or object node in an OPC UA server's address space, see Figure 6.14. Multiple *OPC UA Objects* can be added to a JGrafchart application to set up connections to mul-

tiple nodes on one or more OPC UA servers. The binding for each *OPC UA Object* is configured by specifying the Server URI and the full BrowsePath to the desired node.

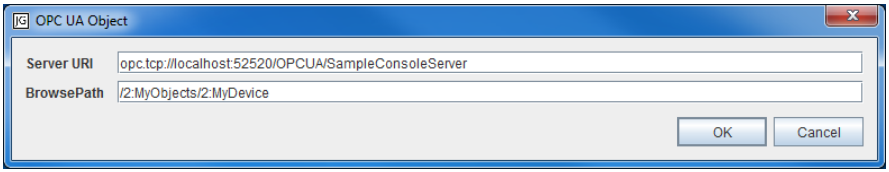


Figure 6.14 The OPC UA Object configuration dialog.

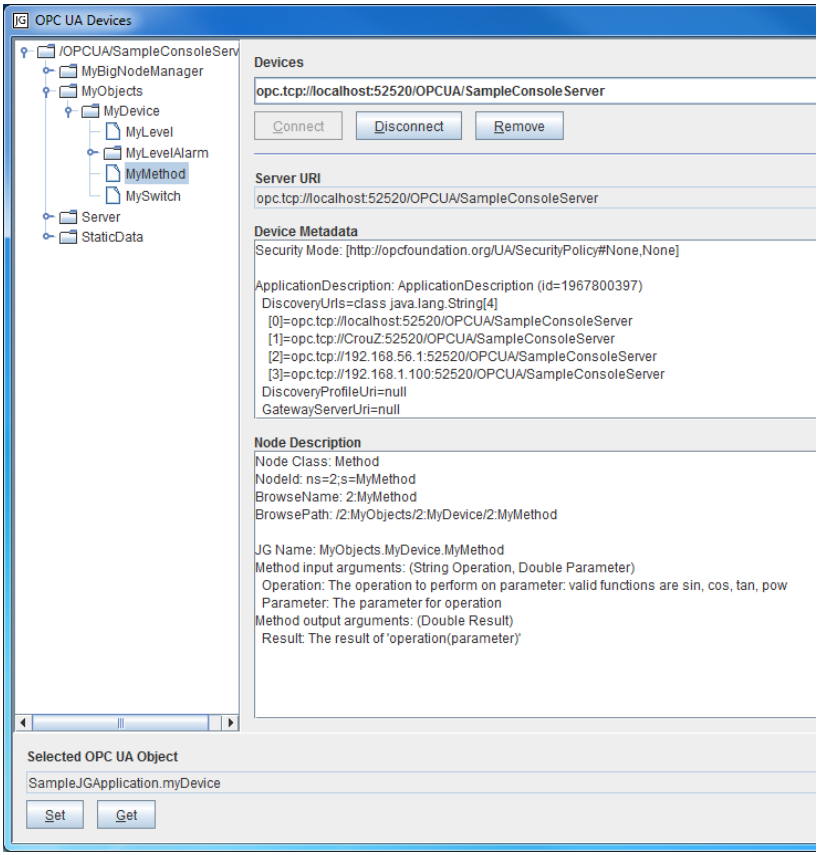


Figure 6.15 The dialog to explore the address space of OPC UA servers in JGrafchart.

A dialog to browse the address space of connected servers has been added to JGrafchart and is illustrated in Figure 6.15. The right side of the dialog shows the currently selected node's metadata, such as its BrowsePath, data types of variables, and input and output parameters for methods, as well as the node's device's metadata such as Server URI and Application Name. It is also possible to connect and disconnect OPC UA servers. At the bottom of the dialog is another way to configure *OPC UA Object* bindings.

When an application with configured bindings is opened, JGrafchart tries to connect to the bound OPC UA servers automatically. Support for OPC UA ServerState monitoring has also been implemented and there is an automatic reconnect when a previously connected server comes back online.

The integrated OPC UA support has been designed so that OPC UA variable accesses and OPC UA method calls look like other JGrafchart variable accesses and method calls. Variables and methods at any level in the bound node's subtree are accessed with dot notation. It is always possible to write to variables and call methods. To receive variable value updates requires subscriptions which are managed with the new JGrafchart functions `opcSubscribe` and `opcUnsubscribe`. Figure 6.16 illustrates how to use OPC UA variables and methods. On the first line a subscription for the variable `MyLevel`'s value is created. On the second line the Boolean variable `MySwitch`'s value is set to true (`1`). On the third line the method `MyMethod` is called with two input arguments and the call's output argument is stored in the JGrafchart variable `ret` (not shown). Finally, the variable `MyLevel`'s value is accessed in the transition condition.

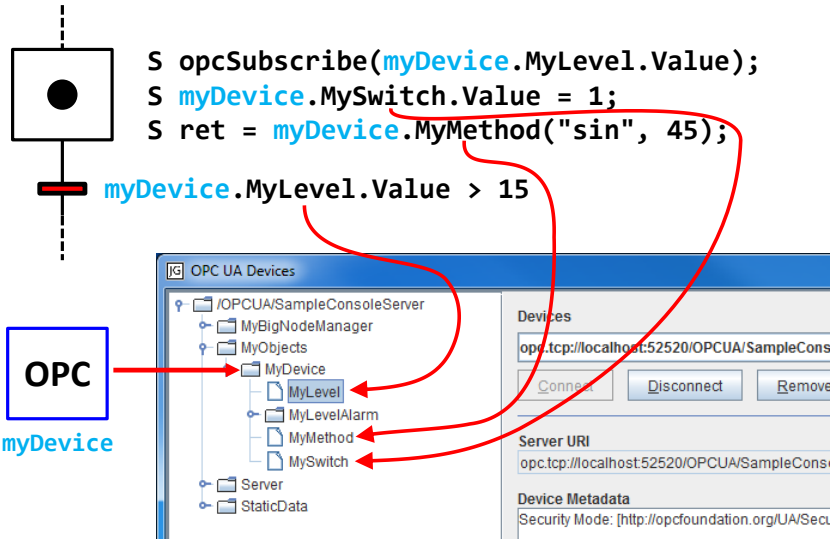


Figure 6.16 How to use OPC UA variables and methods in JGrafchart.

To choose which node in the address space to bind to is a trade off between the number of required *OPC UA Objects* and the number of levels to type for each variable and method access. The extremes are binding to the root node and binding to a variable node. Binding to the root node means that everything on the server is accessible through the same *OPC UA Object* but that each access requires typing the maximum number of levels. Binding to a variable node means that only that variable can be accessed through the *OPC UA Object* but each access only requires typing the minimum number of levels.

Limitations So far, only unsecure communication is supported. However, security handling is implemented in the OPC UA toolkits and it should thus not be a large effort to add support for secure communication.

OPC UA method calls from JGrafchart are currently limited to one output argument as support for several output arguments would require extensive modifications to the JGrafchart action language syntax. Hence, for methods with more than one output parameter only the first output parameter can be obtained. It would probably be a large effort to support multiple output parameters.

Method calls in OPC UA are asynchronous by design which is a new features compared to classic OPC. However, for convenience the toolkit also provides a synchronous version and this is used by JGrafchart. The server may also decide to apply a variable write asynchronously. If this happens a warning is printed in JGrafchart.

Example

A device from fischertechnik [83] has been used to evaluate the new integrated support for OPC UA in JGrafchart, see Figure 6.17. The device consists of a punching machine (1) which can punch goods, and a conveyor belt (2) which can transport goods to and from the punching machine. The device has four digital inputs and two DC motors. Two light barriers detect the position of the goods and two limit switches detect the vertical position of the punching machine. One motor drives the punching machine and the other motor drives the conveyor belt. The device is controlled by electrical signals (voltages) and to expose it as an OPC UA server it is connected to an Ethernet capable microcontroller (3), namely an Aria G25 from Acme Systems [84].

The task for the device is to process arriving goods (4). When goods arrive the conveyor should move it to the punching machine, punch it, and then move it back to the entry again.

Server Side Device Modeling The device is wrapped by its own OPC UA server that runs on the microcontroller, making the functionality of the device available for OPC UA clients to access and control. The conveyor belt is modeled as an object with two variables, `Forward` and `Backward`, which determine how to move the conveyor belt and two variables, `AtEntry` and `AtPunch`, which sense when the goods is at the entry or by the punching machine. Similarly, the punching machine

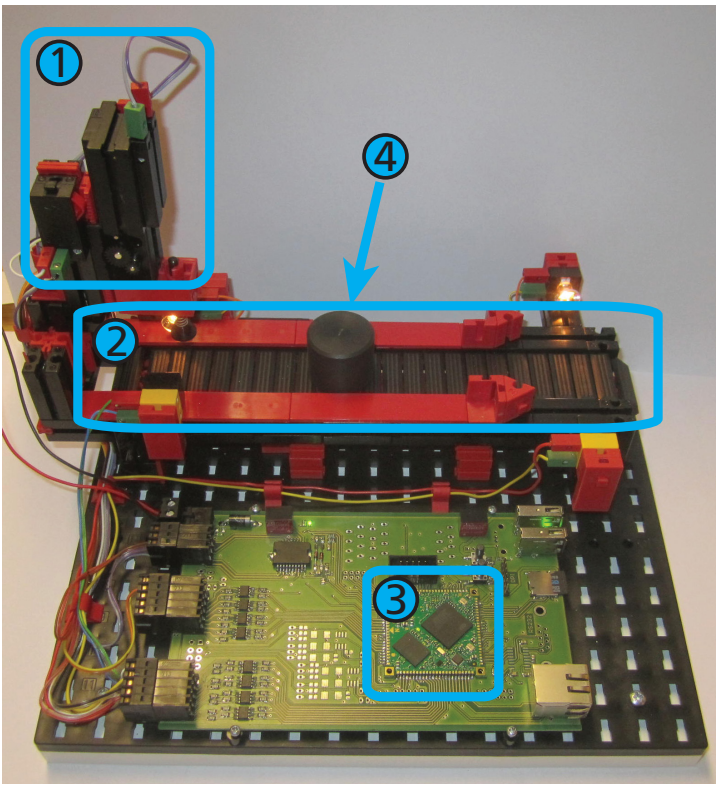


Figure 6.17 The fischertechnik device and the microcontroller circuit board.

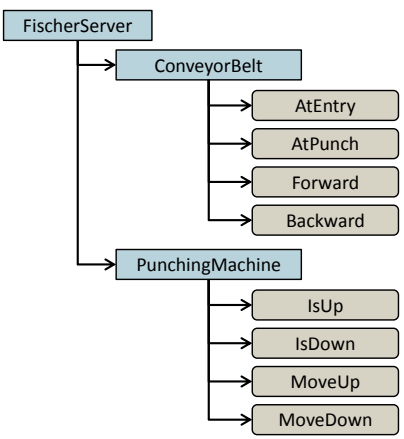


Figure 6.18 The OPC UA server's address space.

is modeled as an object with the variables `MoveUp`, `MoveDown`, `IsUp`, and `IsDown`, Figure 6.18.

To prevent networking issues or a poorly written client application from damaging the device, the server should contain interlocks to avoid situations which might harm the device, for example prevent the punching machine from moving further up when `IsUp` is set.

The OPC UA server was implemented with the C++ OPC UA toolkit from Softing [85] cross compiled to the Aria G25 which runs Debian on an ARM9 processor. To implement the server was straightforward.

Device Control The device is controlled from a JGrafchart application which connects to the OPC UA server, see Figure 6.19. In the application `cb` (conveyor belt) and `pm` (punching machine) are *OPC UA Objects* bound to the corresponding object nodes on the OPC UA server that runs on the microcontroller connected to the fischertechnik device. In addition to what is shown in the figure there is also a separate initial step which sets up subscriptions to variables and forward their values to internal variables used in the transitions.

In the initial step the application waits until goods arrives at the entry of the conveyor belt (transition `GoodsAtEntry`) and it then starts to move the goods toward the punching machine. When the goods arrives at the punching machine (transition `GoodsAtPunch`) the conveyor belt is stopped and the punching machine starts to move down. When the punching machine is in the bottom position (transition `IsPunchDown`) it starts to move up again until it reaches the top position (transition `IsPunchUp`). Then the conveyor belt starts to move the goods back to the entry position and when it arrives (transition `GoodsAtEntry`) the conveyor belt is stopped and the application waits for the current goods to be removed (transition `!GoodsAtEntry`) before going back to the initial state to wait for the next goods to arrive.

Modeling Alternatives The main reason to implement everything as variables is that they are easier to implement on the server side than methods. An alternative would be to have a `move` method for each motor which accepts an argument with the desired direction, for example an integer where positive numbers mean forward, 0 means stop, and negative numbers mean backward. It would also be possible to have both the variables and the methods so that the client can choose which to use.

Another alternative that makes use of OPC UA's modeling capabilities is to model the sensors and motors as types. The sensors and motors can be modeled as two types and then four sensor instances and two motor instances can be created to model the device. The benefit of using types is that each instance of a type will have the same structure and functionality. More alternative ways to model and design OPC UA devices are discussed in [86].

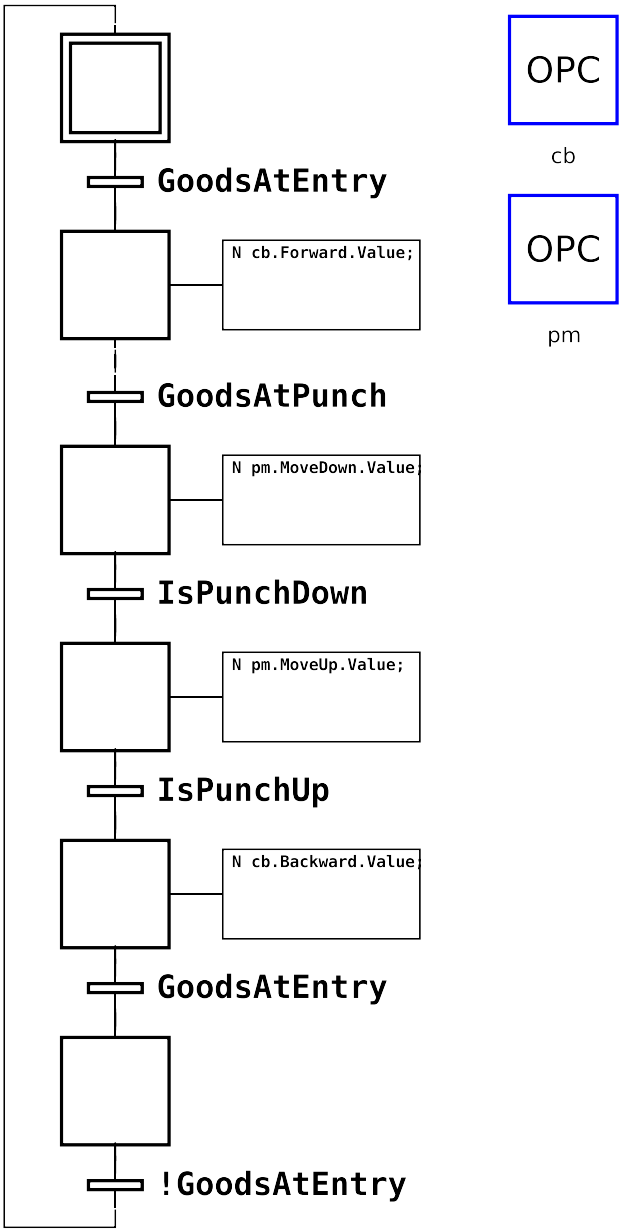


Figure 6.19 Control application for the device's task implemented in JGrafchart with the new OPC UA capability.

Conclusions

A way to implement service orchestration with OPC UA has been presented. The support for OPC UA in JGrafchart is well integrated. OPC UA variables are used like other JGrafchart variables, OPC UA method are used like other JGrafchart method calls, and OPC UA objects are traversed with dot notation like for hierarchical structures in JGrafchart. Automatic connect and reconnect to servers is also supported during both development and execution. The new OPC UA support was successfully used for control of an embedded device which had been made self-contained by wrapping it with an Ethernet capable microcontroller and exposing it as an OPC UA server, a service capable device. JGrafchart can be used to control any device that is accessible through an OPC UA server. Note that with an adapter for classic OPC this includes the field devices used by industrial control systems that support classic OPC.

As concluded in [65], both OPC UA and DPWS are required to to cope with all industrial requirements on device level. As JGrafchart now supports both integrated DPWS and OPC UA it can be used regardless of which is more suitable in a specific scenario.

Data access is by far the most commonly used part of the OPC UA standard. The focus has so far been to integrate OPC UA data access and methods in JGrafchart. Alarm and events and historical data access are part of future work. Method support has also been integrated as this is a key part of web service interoperability. However, unfortunately this interoperability has not yet been possible to test since the OPC Foundation OPC UA Java stack currently only supports the OPC UA TCP transport protocol.

6.4 Service Oriented Architecture 2.0

The key to achieve integration of the smart factory is loose coupling and flexibility of the stand-alone devices. As most factories are modified rather than rebuilt, ease of retrofitting legacy devices to expose their data to the higher levels is also an important aspect.

There is no vendor independent integration architecture for such information management and many companies use their own solutions. Due to increasing demand on customizable production they require flexible and scalable information systems. In this section a new information system architecture that enables flexibility and scalability is presented [87]. It was developed with discrete manufacturing in mind and is called Line Information System Architecture (LISA). The architecture is event-based and uses a prototype-based information model. LISA is able to handle layout and structural changes. As event data is immutable, event sourcing can be utilized to add new aggregated values, not only for new data but also retroactively for historical data. This also makes it possible to modify formulas and re-evaluate them for the historical data.

LISA is currently being installed by a company that has been involved in this research. Previously, at the company, a workstation did only send out predefined KPIs regarding each work cycle. With LISA, all communication is event-based and devices like PLCs, robots, product carriers, and operators only send and receive events. LISA has also been evaluated on historical data from another automotive manufacturer. The data did not at all conform with the LISA structure but due to the flexible nature of LISA, events could be identified and generated from the data.

LISA

A common approach for information systems is an object-oriented structure for event types and events. LISA on the other hand uses a prototype-based approach. Prototypal inheritance, unlike object-oriented inheritance, is achieved by cloning and refining an object, here an event. This makes the event creation, identification, and filtering less rigid since the strict hierarchical relation enforced by a class structure is removed.

The core components of LISA are the message bus, communication endpoints, and the LISA message format. Creation and transformation of raw event data into valuable information is done in a modular and loosely coupled way. An overview of the LISA communication architecture is shown in Figure 6.20.

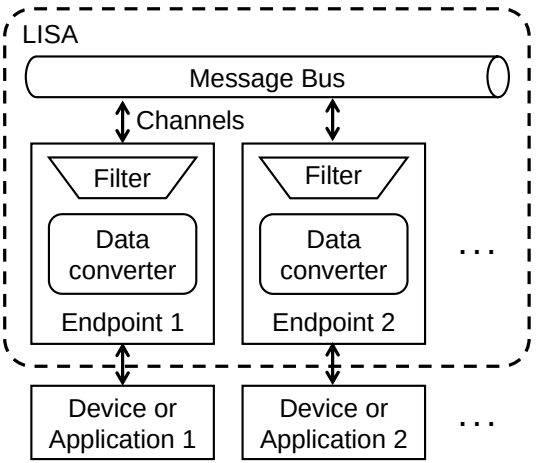


Figure 6.20 An overview of the LISA architecture.

Message Bus

LISA uses an Enterprise Service Bus (ESB) for messaging that takes care of the routing of data/information to distributed applications. To avoid PtP connections,

the ESB should support the following patterns defined by the Enterprise Integration Patterns (EIP) [88, 89]:

- *Message*: The data or information are packaged into a message that the messaging system can transmit through a message bus.
- *Messaging*: Packets of data are transferred frequently, immediately, reliably, and asynchronously using customizable formats. Messaging is an event-based process: as soon as a new message is ready it is sent to the message bus.
- *Publish-subscribe channel*: When a message is sent on a publish-subscribe channel, a copy of this particular *message* is delivered to each channel subscriber.
- *Message filter*: If the content of an incoming message does not match the criteria specified by the message filter the message is discarded. This pattern allows each application that subscribes to a publish-subscribe channel to further filter incoming messages.

In the LISA prototype Apache ActiveMQ [90] has been used, but it could be replaced by any ESB that supports these patterns.

Communication Endpoints

The connection between applications (communication devices, services, external applications) over the ESB is done through communication endpoints. Many devices have limited capabilities and limited knowledge and they communicate with different device specific protocols and interfaces, for example OPC or RS-232. To exchange all production equipment to new devices which all support the same specific protocol and interface is unfeasible. Instead, the diversity of devices has been embraced and in LISA a communication endpoint is the way to integrate a device.

The communication endpoints are adapters between the messages on the ESB and the device. Device event data are converted to the LISA message format and are published on ESB channels. Similarly, the communication endpoint filters events and converts and communicates event data to the device. The communication endpoint responsibilities are thus:

1. Communicate with the device.
2. Convert between the device's data format and the LISA message format.
3. Publish LISA messages on ESB channels.
4. Filter incoming LISA messages from the ESB.

If an application is modified (for example due to hardware replacement, variables renaming, or new sensors), only the corresponding communication endpoint needs to be updated.

LISA Messages

The LISA message format is designed to be simple and enforce as little structure as possible. It consists of a header and a body. The header has information related to message sending and routing. The body is an ordered key-value map between attributes (the keys) and their values. Values are usually of primitive data types like strings or integers but can also be lists or maps. Hence arbitrary hierarchical structures can be built and sent in a LISA message. Two attributes are mandatory in the body, namely an event *id* and an event *timestamp*, otherwise there are no constraints. This makes LISA flexible as it is easy to update, change, and add new attributes. On the other hand, LISA users need to decide the structure themselves.

LISA messages sent on the ESB are immutable. To refine a LISA message, a new LISA message is created and sent to the ESB. The new message will have the same *id* and *timestamp* but the content of the message may otherwise change arbitrarily.

Persistence

When an application failure occurs, for example a random application crash, the application should be able to recover gracefully and it should behave the same as if it had not failed. Ideally, other applications should not be able to tell if it has crashed or not. Of course, the timeliness will be affected but the events generated by the crashed application should not be affected.

If all events are persisted they can be replayed in the restarted application to make it reach the same state as when it crashed. Then the application can proceed from there and produce the same events as if it had not crashed. Replay performance can be improved by occasionally persisting a state snapshot to limit how much history has to be replayed. For practical reasons there should be logic to avoid event duplicates during replay.

To store the whole history of events with the purpose to persist an application's state is called event sourcing [24]. Compared to persisting the state itself there are some notable differences. With event sourcing the exact same application behavior can be replayed and analyzed in detail. For example, if an application is found to be in an incorrect state it is possible to step through the replay of events to find out which event processing introduced the error. It might even be possible to go back and correct some such errors retroactively.

Another advantage of event sourcing is that it is possible to apply the event history to new applications. For example, if an application that calculates a new KPI is added it might be possible to calculate that KPI retroactively for the whole history. Since LISA is based on event sourcing and store only the events in a journal, it is possible to change and add services and then replay the events to update the information. The disadvantage is that it uses more storage space, but since storage is cheap and there are many potential benefits it is often well worth it [91].

Event-Based Control

Events originating from different levels can be used to perform high-level coordination and control. The production could for example be initiated by business level order event and carried out by interaction between production machines, coordination software, and field devices. As there is no central coordinator, this is classified as service choreography rather than service orchestration.

Manufacturing Views Two manufacturing views are machine-centered and order-centered. A machine-centered view focuses on the flow through each machine while an order-centered view focuses on flow of the product being produced for each order. On the lower level, a machine-centered view makes more sense since it is easier to get the control right for one stand-alone machine than for a group of coupled machines. On the higher level, an order-centered view is more relevant as produced products are parts of orders which should be possible to trace and visualize. For the coordination in between, either view may be chosen and both have their advantages. Either you get a good overview of the machines or of the order. With the right events both views can be constructed.

Control Unlike the upper levels which should not depend on details about the lower levels, control and coordination require some knowledge about the lower levels. A coordination application should also select a manufacturing view. With a machine-centered view, each machine selects which product to process next and gets information about how to process the product. With an order-centered view, each product selects where it should be processed and tells the machine how it should be processed.

With three similar workstations a machine-centered application could look like in Figure 6.21 and an order-centered application could look like in Figure 6.22. Here reusable procedures are particularly useful for workstation and order control.

A difference between event-driven control and service orchestration with DPWS or OPC UA is that these have built in error handling to make it possible to detect if an operation call fails. With event-driven control, a request is sent as an event and to tell if the operation was successful an acknowledgment event is required.

JGrafchart to LISA Connection A communication endpoint for JGrafchart's Socket I/O was created. There is a mismatch between LISA which is event-driven and JGrafchart which is executed periodically. If events are allowed to arrive at any rate to JGrafchart, pulse events might be invisible to the JGrafchart application. To avoid this kind of issues the communication endpoint throttles the message delivery rate to JGrafchart according to the application's scan cycle time.

Demonstrator Control with LISA was evaluated against a system consisting of a real PLC connected to a physical system, a simulated CNC machine, and a simulated order system. The CNC and order system consist of communication endpoints only and the PLC system is connected via an OPC communication endpoint. In the

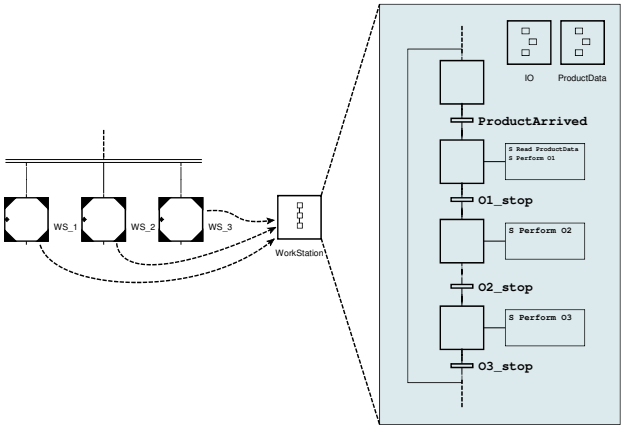


Figure 6.21 A pseudo chart for machine-centered control. The workstation has been extracted as a procedure which is parameterized by its I/O and processing information for each product.

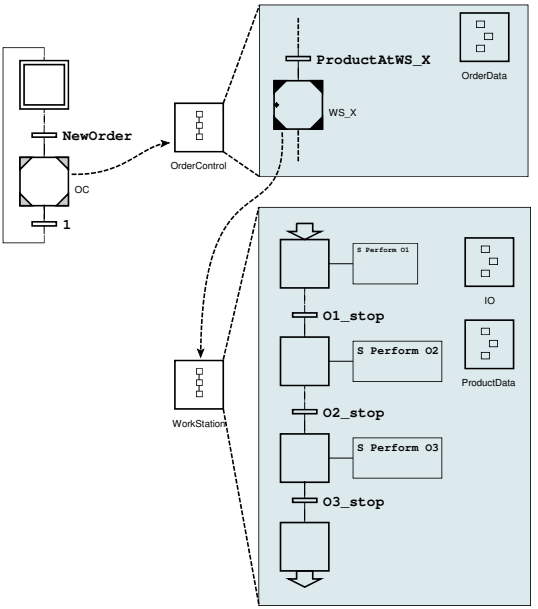


Figure 6.22 A pseudo chart for order-centered control. The control for each product is executed in a separate procedure call.

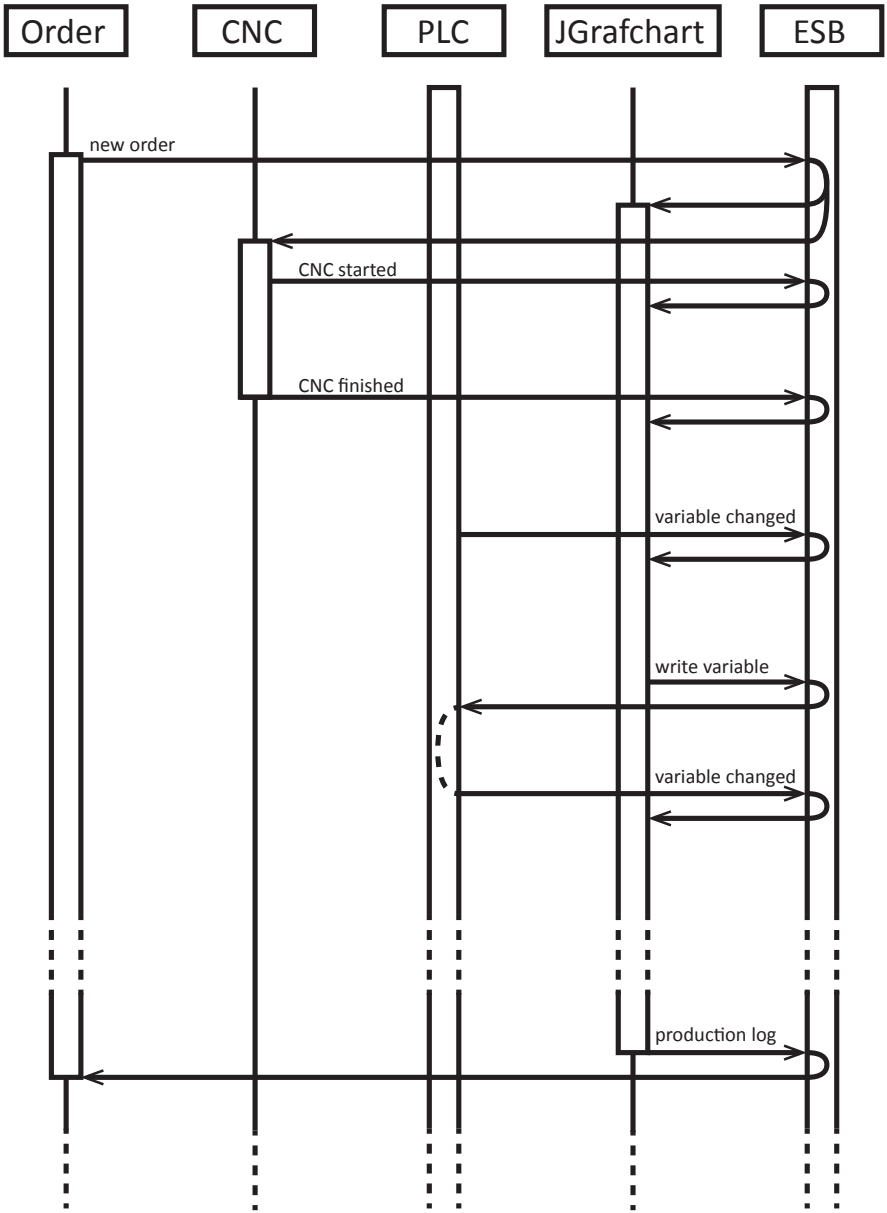


Figure 6.23 An overview of the production of an order in the LISA demonstrator.

OPC communication endpoint, all writable variables generate an event when they change which ensures acknowledgments for write requests.

An overview of the production of an order is shown in Figure 6.23. For the demonstrator an order-centered view was selected and when a new product is requested by the order system its control in JGrafchart is executed in a separate procedure call. The request for a new product also triggers the CNC machine to start producing the product. When the CNCing is completed the product enters the physical system controlled by the PLC and JGrafchart begins to send requests to the PLC which handles the real-time control. When the production is completed, an event containing the production log is sent. The resulting control program in JGrafchart is similar to that of Figure 6.22.

Conclusions

Application integration in LISA was straightforward for the demonstrator. The advantages of the extreme loose coupling of EDA were also experienced. In particular, applications could be developed and tested in isolation as the other application were easily replaced by mockups which produce events without their respective physical device.

The presented architecture has been developed with discrete manufacturing in mind, where processes are running asynchronously. One core feature of LISA is that it should be usable for any device and application. As a result the LISA message format is very flexible.

Involvement of several industrial partners provided valuable feedback on the applicability of the research. Off-the-shelf solutions, for example ActiveMQ, permits reuse of well-proven technology. The industrial involvement also permitted evaluation of the architecture.

As a result, LISA is an event-based service oriented architecture which offers flexibility and scalability both for control of low-level applications and aggregation of higher level information, such as KPIs.

6.5 LabComm

LabComm [61] is a binary communication protocol for one-way communication of structured data samples. When a connection is made the signature of all possible samples are sent. The protocol is thus self-describing and compatibility can be ensured at runtime by comparing received and expected signatures. When the signatures have been sent, samples are sent with minimum communication overhead. LabComm is thus suitable for performance critical communication. Primitive data types such as boolean, int, double, and string are supported as well as structs, fixed and variable size arrays, multidimensional arrays, and arrays of arrays. A LabComm sample is described by a specification which can be used to generate code for various programming languages such as C, C#, Java, and Python.

There exists two versions of LabComm, namely the 2006 and the 2013 version. The sample specification language is the same but the binary format has been changed and thus the versions are runtime incompatible.

Motivation

With real-time execution of JGrafchart applications, high performance interaction with the applications should also be possible. Integrated LabComm support has been added to support high performance communication of structured data.

LabComm is and has been used in several robotics research projects at the department, for example ROSETTA [39, 38] and PRACE [92] where it is used for sequence control with JGrafchart. However, as there was no integrated support for LabComm in JGrafchart a LabComm *Custom I/O* based on an internally modified *Custom I/O* interface was used. This had several drawbacks such as a hard coded LabComm specification, only support for the 2006 version, hard coded TCP connection, and the lack of *Custom I/O* array support. The hard coded LabComm specification consisted of a few LabComm samples where each sample was a large array. Instead of a few arrays in JGrafchart, there were nearly 1,000 *Custom I/O* elements.

In the robotics environment the LabComm connection is made through an Orca [93] server which requires some additional special handling.

Normal LabComm Use

The LabComm implementation primarily consists of two parts, the compiler and the runtime library. The compiler parses and checks the LabComm specification and produces generated code for the target language. The semantics and code generation are implemented with JastAdd. The runtime library provides communication classes and implements the low-level communication details.

The typical workflow is to, see Figure 6.24:

1. Write a LabComm specification
2. Generate code to handle the LabComm samples in the specification in the target language
3. Integrate the generated code

Since it should be possible to write JGrafchart applications with custom samples this workflow cannot be used.

LabComm Integration in JGrafchart

To use LabComm in JGrafchart the new I/O element *LabComm Object* has been added. Each *LabComm Object* has its own LabComm sample specification and settings for the connection, LabComm version, and channel direction, see Figure 6.25.

The workflow to use LabComm with the integrated support in JGrafchart is to add a *LabComm Object* to the application, configure it, and write the application to

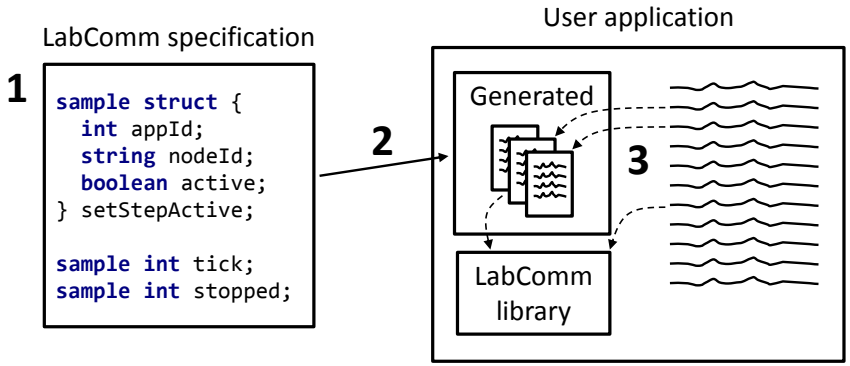


Figure 6.24 The typical workflow to write an application that supports LabComm. 1) write the LabComm specification, 2) generate code for the target language, and 3) integrate the generated code.

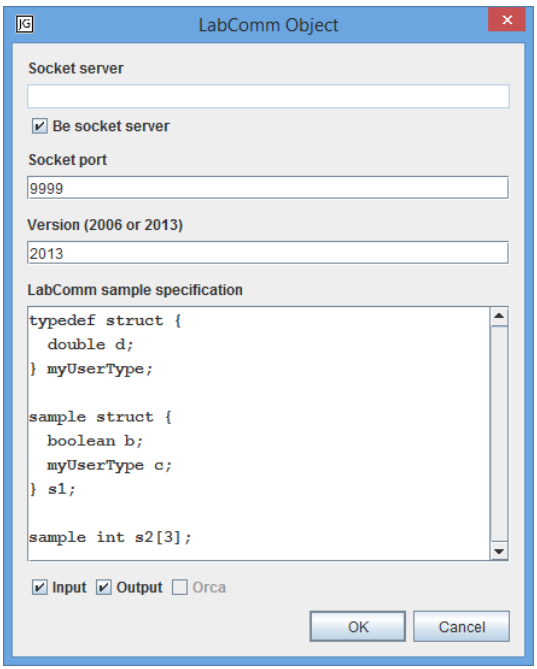


Figure 6.25 Configuration dialog for a *LabComm Object*. This particular *LabComm* object is configured to run a socket server on port 9999, to communicate using the binary format of LabComm version 2013, and to be allowed to both send (output) and receive (input) samples. In the LabComm specification, a custom type called `myUserType` and the two samples `s1` and `s2` are declared.

send and receive samples and use the sample data. Compared to the normal workflow, the steps to generate and integrate code are automatic and the user does not have to know the internals of the LabComm implementation.

The structured data in the LabComm specification are mapped to invisible internal JGrafchart elements. Primitive LabComm types are mapped to the corresponding JGrafchart variable types, for example a LabComm double is mapped to a JGrafchart Real variable. LabComm structs are mapped to Workspace Objects. One-dimensional LabComm arrays of primitive types are mapped to JGrafchart lists. Other arrays are not supported. The internals of the *LabComm Object* are accessed with dot notation. The samples are accessed by their respective sample name and are placed immediately below the *LabComm Object*. Figure 6.26 shows the internals of the *LabComm Object* in Figure 6.25 and how to use it. As the sample `s2` has a fixed length, the internal JGrafchart list is initialized with that number of elements with its data type's default value.

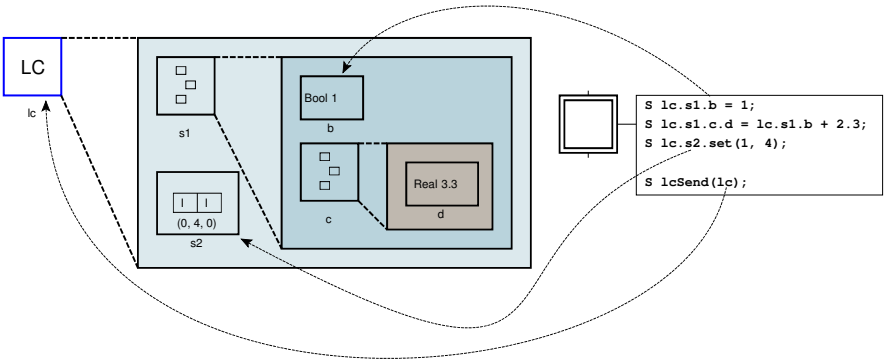


Figure 6.26 How to use the data in the *LabComm Object* from Figure 6.25, here named `lc`. The internal structure of `lc` is shown to the left. `s1` is a struct and has thus be mapped to a Workspace Object and `s2` is an int array and has thus been mapped to an Integer List. In the first action the boolean `b` of sample `s1` is assigned the value `1`. In the third action the second element of the int array is assigned the value 4. Finally, all samples in `lc` are sent, that is, both `s1` and `s2`.

LabComm samples are sent with the new built-in function `lcSend()` which takes either one or two parameters. The first parameter to `lcSend()` is the *LabComm Object* and the second parameter is the name of the sample to send. If the second parameter is omitted, all samples in the *LabComm Object* are sent.

When a LabComm sample is received the data in the *LabComm Object* is updated accordingly. The new built-in function `lcReceived()` can be used to check if a LabComm sample has been received since the previous scan cycle.

Implementation

The LabComm compiler code was added to JGrafchart together with two new JastAdd extensions. One extension adds interpreted execution of the LabComm sample specification AST. It does what the generated classes typically do, that is, registers, encodes, and decodes samples. The other extension adds JGrafchart specific features. It performs JGrafchart specific checks, such as checking that only supported arrays are used. It is also the glue between the LabComm samples and the *LabComm Object*. It creates the *LabComm Object*'s internal structure and integrates it with sending and receiving of samples.

The LabComm runtime library code for Java for both version 2006 and 2013 were also added to JGrafchart. They had to be modified slightly since it assumed that code generation would be used.

Special code to connect through an Orca server for LabComm version 2006 was also added.

Conclusions

The LabComm integration in JGrafchart has been tested and works both for normal use and through an Orca server. Less knowledge about LabComm is needed to use LabComm in JGrafchart than in for example Java since more is taken care of automatically. The next step is to replace the special *Custom I/O* implementation which should be straightforward.

To add code generation for the integrated LabComm support in JGrafchart should also be straightforward as this is closer to the normal LabComm use, see Figure 6.24. The LabComm specification is given by the application and code generation for LabComm as well as the LabComm library are already available for several languages. What remains to be added is code generation for the language specific glue code.

6.6 Functional Mock-up Interface

FMI [94] is a recent standard which aims to combine dynamic system models developed in various tools. Modelica [95], the state of the art language to express dynamic behavior of technical systems, promotes this standard and the number of tools that support FMI is increasing rapidly.

A tool can export a model as a Functional Mock-up Unit (FMU) which can then be combined with other FMUs to compose the whole system. The FMI standard consists of two parts, namely *FMI for Model Exchange* and *FMI for Co-Simulation*. The difference is that an *FMI for Model Exchange* FMU only provides a model while an *FMI for Co-Simulation* FMU provides both a model and an individual solver to simulate the system's behavior. For *FMI for Model Exchange* all models can be combined and simulated by the same solver. For *FMI for Co-Simulation* all individual solvers are instead coordinated.

In this section it is investigated if and how *FMI for Co-Simulation* support can be added to JGrafchart [96]. First *FMI for Co-Simulation* is described in more detail. Then the need to connect JGrafchart to *FMI for Co-Simulation* is motivated and possible ways to implement it is discussed.

FMI for Co-Simulation

FMI for Co-Simulation is a standard which enables simulation of coupled technical systems with focus on time-dependent problems. It is designed for both stand-alone FMUs and FMUs which are wrappers for simulation tools.

A co-simulation is executed from a given starting time to a stop time which is not necessarily pre-specified, see Figure 6.27. There is an FMI *master* which coordinates the co-simulation and there are FMU *slaves* which correspond to models or subsystems. Each *slave* has a pre-specified set of inputs and outputs which are known by the *master*. The *master* is responsible to initialize the *slaves* and handle the coupling between them by getting and setting their inputs and outputs.

The co-simulation is executed for one time interval at a time, known as a *communication step*, during which each *slave* executes independently. Between the *communication steps* are the *communication points* where the *master* communicates the inputs and outputs between the *slaves*. *Slaves* can specify their desired *communication step* size and the *communication step* size may also vary during the co-simulation provided that all *slaves* support this. A *communication step* may also fail. Then a new *communication step* of different size may be attempted if all *slaves* support this. It is the *master* who decides the *communication step* size and what to do when one fails.

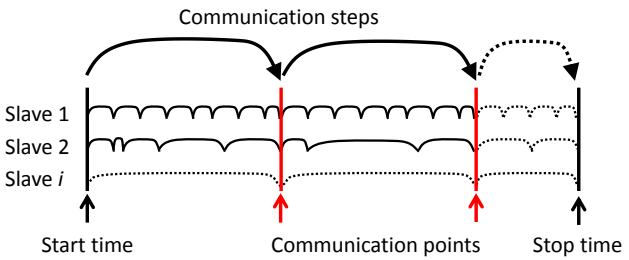


Figure 6.27 Overview of an FMI co-simulation.

The standard does not define an FMI *master* algorithm, and the level of sophistication is depends on the *master* implementation. What the standard does define is the Application Programming Interface (API), a set of *slave* capabilities, and rules for how these may be used.

An FMU is described by an XML metadata file which primarily contains the inputs and outputs. It also contains co-simulation capabilities such as support for variable *communication step* size and *communication step* redo.

Motivation

In a laboratory exercise at the department, JGrafchart is used to control both a simulated and a real batch tank. The simulated process is implemented as a simplified model in Java. It is also possible to implement simulations of simple processes directly in JGrafchart. However, there is much potential for improvement in terms of effort to specify the simulated model, quality of the models, support to inspect simulation results, and time required to simulate, especially for more complicated physical systems. To extend JGrafchart with support for *FMI for Co-Simulation* gives more and better opportunities to connect JGrafchart to other tools.

There is also a need to efficiently develop and test JGrafchart control applications before they are used to control the real system. This may save a lot of time as many industrial systems have slow dynamics and to run simple test on the real system could take days. With a good model of the system the development time could be reduced considerably, and the quality of the control application will be higher. Industrial systems are often hazardous and to run a proper simulation first could be essential for safety reasons. For the batch tank in the laboratory exercise, the simulated process is 10 times as fast as the real process. Special code is required to add support for this which both makes it fragile and susceptible to errors as it is possible to run the control application in the wrong mode, for example in simulation mode against the real system. With a simulation environment that does not run according to wall clock time, it can be run faster and without the special code.

It is also important to verify that the system behaves properly when controlled by a JGrafchart application. JGrafchart executes periodically and only sees the sampled behavior. When controlling a continuous system, the behavior between the sampling points may also be of interest. Also, JGrafchart applications can currently only be executed according to wall clock time. With a large or complex simulated system the JGrafchart application might execute faster than system can be simulated.

Support for state machines were introduced in Modelica 3.3 [97], providing a way to implement hierarchical state machines directly in Modelica. On one hand, JGrafchart does not provide the mutual hierarchical structuring property with data flow that Modelica state machines do [97]. On the other hand JGrafchart supports high-level features such as object-orientation, hierarchical structuring, code reuse, and exception handling and is based on an industrial automation language.

JGrafchart with FMI Support

The JGrafchart data types Real, Integer, Boolean, and String correspond to the FMI data types fmiReal, fmiBoolean, fmiInteger, and fmiString. Both variables, lists (arrays), and I/O in JGrafchart use only these data types. As the I/O in the JGrafchart application are the means to connect it to external components, these would ideally be the FMU inputs and outputs. The mapping for *Custom I/O* and *Socket I/O* is straightforward. DPWS on the other hand is based on method calls instead of data and need some configuration to be able to expose the methods as data instead. The

same applies to OPC UA methods. Thus DPWS and OPC UA methods should be excluded initially. Something similar to the socket I/O prototype for DPWS could be added later. The number of FMU inputs and output is fixed. The only case where this is an issue is for variable size arrays in LabComm samples. A workaround for this could be to specify an upper bound for the array size.

Regarding *communication step* redo, the state of a JGrafchart application is described by variable, list, and I/O values as well as which steps are active, how long they have been active, and current procedure calls. The number of simultaneous procedure calls and the list sizes are unbounded. However, as these are internal states this should not be an issue.

JGrafchart applications are executed periodically, one scan cycle at a time, as described in Chapter 4. The execution can be modeled as discrete events at the beginning of each scan cycle. During the rest of the scan cycle nothing happens. Ideally, there would be *communication points* just before and just after the beginning of each scan cycle. With a sufficiently small *communication step* size this should be sufficient for all JGrafchart applications, regardless of scan cycle time. This could be requested by setting the `stepSize` attribute of the `DefaultExperiment` element in the FMU XML.

JGrafchart applications are currently always executed according to wall clock time and it is not possible to get and set the execution state as there has been no need for this before. However, it should be possible to extend JGrafchart with the possibility to get and set the current execution state to support *communication step* redo.

Next, ways to connect a JGrafchart application to an FMI *master* are discussed.

Hardware-in-the-loop The simplest way is to consider the JGrafchart application as a hardware-in-the-loop, see Figure 6.28. Then the JGrafchart application executes as usual, with the FMI *master* getting and setting its I/O. The co-simulation must then be able to keep up with and synchronize with JGrafchart's wall clock time execution. The main advantage of this approach is that no modifications to JGrafchart are necessary, it would be sufficient to create an FMU compatible *Custom I/O* or TCP server for *Socket I/O*. This is a suitable approach for FMU integration prototyping but it does not improve matters for systems with slow dynamics.

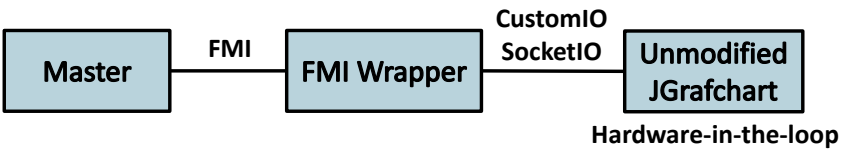


Figure 6.28 Overview of connecting JGrafchart as a hardware-in-the-loop. The FMI Wrapper has been written for a specific JGrafchart application.

Generic FMI Wrapper Another approach is to implement a generic FMI wrapper for JGrafchart and extend JGrafchart with support for external clocks, see Figure 6.29. It is a small effort to add this feature. The same FMI wrapper would be possible to use with any JGrafchart application and the wrapper would expose different inputs and outputs to the FMI *master* depending on the JGrafchart application that it currently wraps. This approach only requires slightly more effort than the hardware-in-the-loop approach and gives more benefits as the co-simulation is no longer executed according to wall clock time. For this approach it is suitable to also add support for playback and to be able to inspect individual scan cycles of the JGrafchart application during the co-simulation. To add these features should only be a moderate effort, it could be as simple as trace printouts or as advanced as interactive scan cycle stepping. Compared to the hardware-in-the-loop case, the main drawback is that modifications to JGrafchart are required. However, these additions are not solely useful for *FMI for Co-Simulation*. They open up possibilities for integration with other tools and improves JGrafchart's debugging capabilities.

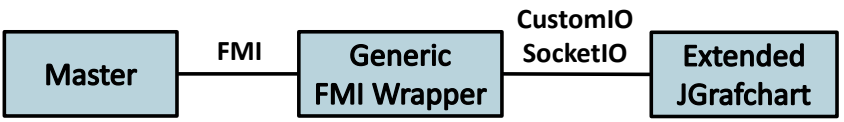


Figure 6.29 Overview of connecting JGrafchart with a generic FMI wrapper. The FMI Wrapper is written so that it can be used for any JGrafchart application. Here, JGrafchart has also been extended to make it possible to run application according to an external clock.

Stand-alone FMU Yet another approach is to generate a stand-alone FMU for a JGrafchart application, see Figure 6.30. The FMU is then self-contained and does not rely on JGrafchart running in parallel. This is a clean and portable approach but requires the most effort and might make it harder to inspect the co-simulation results.

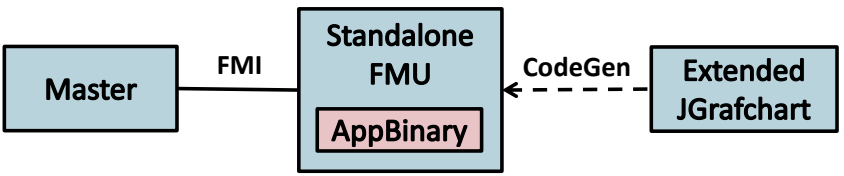


Figure 6.30 Overview of using code generation to create a stand-alone FMU for a JGrafchart application. The FMU is executed without JGrafchart.

A hybrid approach is to use a generic FMI wrapper with both JGrafchart and the JGrafchart application embedded as additional FMU *resources*, see Figure 6.31.

Then the FMU is stand-alone but no code generation is required. A drawback with this approach is that the FMUs would be roughly 20 MB larger, this is, the size of JGrafchart.

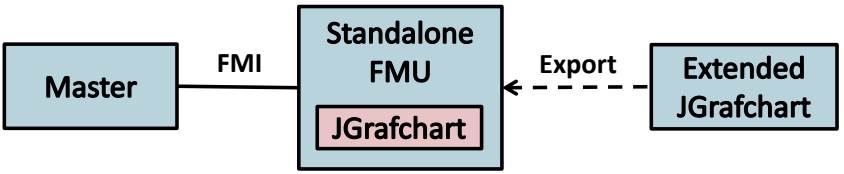


Figure 6.31 Overview of exporting a stand-alone FMU with JGrafchart and the JGrafchart application embedded.

Implementation

The FMI API is defined for C and FMUs are distributed with C source code and/or binary executables for supported platforms. JGrafchart is written in Java and is platform independent. However, the FMU itself can be implemented in any language which is able to interact with C code, that is, practically any language. There are language specific wrappers for FMI, for example PyFMI for Python [98] and JFMI for Java [99] which uses JNA [100] to interface with native code. Which language that is used is less important and up to the one who implements the FMU. Note, however, that parts of the *Custom I/O* implementation must be written in Java.

Conclusions

It has been investigated if and how support for *FMI for Co-Simulation* can be added to JGrafchart. It appears to be possible and there are several alternative ways that it could be realized.

The next step is to implement a prototype to verify that it actually works. A suitable first attempt would be the hardware-in-the-loop approach with an unmodified JGrafchart implementation and the use of *Custom I/O* and/or *Socket I/O*. A desirable future solution would be either Generic FMI Wrapper or Stand-alone FMU.

7

Grafchart for PID Control

In this chapter, work related to PID control is presented [101] [102]. As control loops typically need to be executed in real-time, this work is related to the real-time execution described in previous chapters.

First, PID control is introduced. Next, a PID controller that has been implemented as a reusable library component in JGrafchart is described. It makes it easy to use Grafchart for PID control in general and for education in particular. Finally, a previously unknown problem with the PID controller algorithm is presented and a solution to the problem is proposed. Both the PID component and the improvements to the PID algorithm are immediately usable wherever PIDs are used. This includes for example both process automation, manufacturing, and robotics.

7.1 PID Control

The PID controller is used in more than 95% of all control loops [13] and is by far the most commonly used controller in industry. A basic PID controller in continuous time is described by

$$u(t) = K\left(e(t) + \frac{1}{T_i}\int_0^t e(\tau)d\tau - T_d\frac{dy_f(t)}{dt}\right) \tag{7.1}$$

where $u(t)$ is the control signal, $e(t)$ is the control error (2.1), $y_f(t)$ is the filtered process value, K is the controller gain, T_i is the integral time, and T_d is the derivative time.

PID Features

A PID implementation must consider many aspects to ensure good behavior under all circumstances. In particular for the work described later, physical limits of signals need to be considered. If the physical limits for the control signal are not considered, there is integrator windup [103] when the control signal saturates, see Figure 7.1. Here, the integrator, and thus the desired control signal, continues to

grow even though the real control signal is saturated and cannot be increased further. When the set-point is reached and the integral terms starts to decrease, it takes a long time before the desired control signal is in the allowed range again. This causes a large process value overshoot which is not desirable. The solution to integrator windup is known as anti-windup and involves adjusting the integral part according to the actually actuated control signal. This means that the control signal limitation is considered and that this knowledge is used to make sure that the control signal does not grow outside the control signal range. Hence, there is no overshoot, see Figure 7.2.

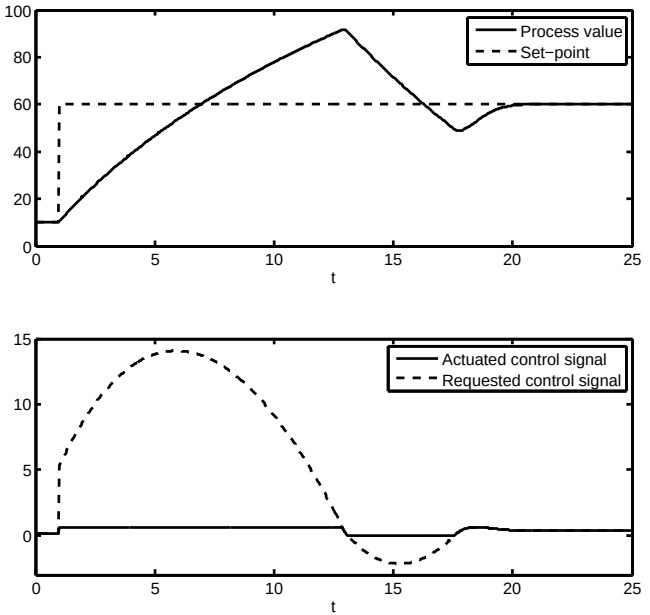


Figure 7.1 An example that illustrates integrator windup. The upper plot shows the process value (solid) and set-point (dashed). The lower plot shows the actually actuated control signal (solid) and the control signal requested by the PID (dashed).

In (7.1) the controller parameters are constant, which is not true for practical applications as the operator may change them at any time. A change in controller parameters should be bumpless, that is, the change should not cause a sudden control signal change. Many PID implementation also support manual mode, that is, the PID is temporarily turned off and the control signal is chosen manually. To turn manual mode on and off should also be bumpless.

Tracking is a feature similar to anti-windup which applies to all situations where the actually actuated control signal is not the same as the PID's requested control signal. One example is selectors, where several controllers are run in parallel and,

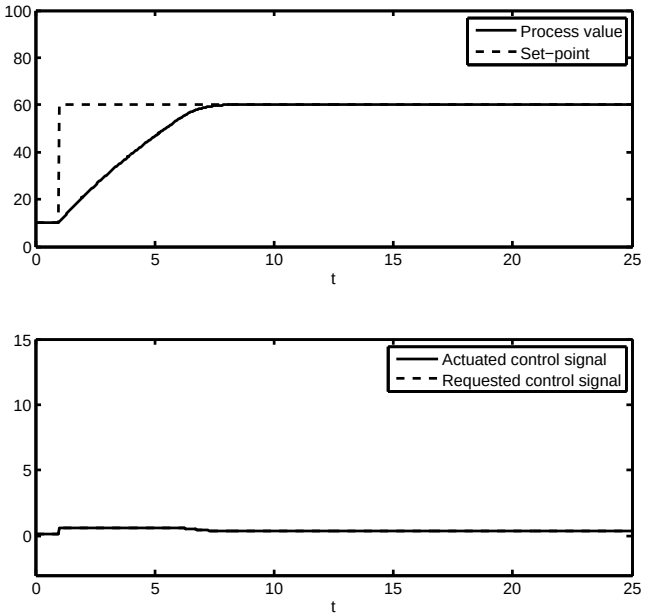


Figure 7.2 The same experiment as in Figure 7.1 but with anti-windup included in the controller. The requested and actuated control signals are almost the same.

depending on the circumstances, the control signal from one of the controllers is selected and actually actuated. All controllers then need to track the actuated control signal to make sure that switching between the controllers is bumpless.

PID Algorithms

Most PID controllers are implemented on positional form, which means that the control signal is calculated directly. Each sample, a new control signal is calculated based on the new and/or previous process and set-point values as well as on the internal state of the PID controller. A drawback with positional form is that the internal states, primarily the integrator state, are sensitive to mode and parameter changes and many special cases are require to get proper behavior.

An alternative to positional form is velocity form, which instead calculates an increment to the control signal each sample. An advantage with velocity form is that it has fewer internal states. Most notably, the integral term is not an internal state of the controller. Instead it is included in the control signal itself, which means that several controller features like anti-windup, tracking, and bumpless mode changes practically come for free. In conclusion, the velocity form implementation does not require as many special cases which makes it more compact and easier to implement correctly.

A discrete PID algorithm on velocity form [103] is described by

$$\begin{aligned}
\Delta u(t_k) &= u(t_k) - u(t_{k-1}) = && (7.2)\\
&= \Delta P(t_k) + \Delta I(t_k) + \Delta D(t_k) + \Delta u_{ff}(t_k) && (7.3)\\
\Delta P(t_k) &= P(t_k) - P(t_{k-1}) = \\
&= K(by_{sp}(t_k) - y(t_k)) - K(by_{sp}(t_{k-1}) - y(t_{k-1})) && (7.4)\\
\Delta I(t_k) &= I(t_k) - I(t_{k-1}) = b_{i1}e(t_k) + b_{i2}e(t_{k-1}) && (7.5)\\
\Delta D(t_k) &= D(t_k) - D(t_{k-1}) = \\
&= a_d \Delta D(t_{k-1}) - b_d(y(t_k) - 2y(t_{k-1}) + y(t_{k-2})) && (7.6)\\
\Delta u_{ff}(t_k) &= u_{ff}(t_k) - u_{ff}(t_{k-1}) && (7.7)
\end{aligned}$$

where y is the process value, y_{sp} is the reference value, Δu is the control signal increment, and ΔP, ΔI, and ΔD are the increments for the proportional, integral, and derivative part respectively. K is the controller gain and b is the scalar set-point weight for the proportional part. b_{i1} and b_{i2} are constants which depend on how the integral term is discretized. Similarly a_d and b_d are constants which depend on how the derivative term is discretized. Most derivative part discretizations except for backward difference either have issues with overflow or instability. For backward difference the constants are $a_d = \frac{T_d}{T_d+Nh}$ and $b_d = \frac{KT_dN}{T_d+Nh}$, where T_d is the derivative time and N is the maximum derivative gain. Finally, u_{ff} is the feedforward signal.

Without integral action the velocity form has arbitrary stationary error. The reason is that the proportional part is assumed to be included in the control signal and only needs to be adjusted when the measurement value changes. This assumption does not hold, for example, when tracking or when the control signal saturates. To avoid this issue, the proportional part is then instead

$$\begin{aligned}
\Delta P(t_k) &= P(t_k) + u_b(t_k) - u(t_{k-1}) = \\
&= K(y_{sp}(t_k) - y(t_k)) + u_b(t_k) - u(t_{k-1}) && (7.8)
\end{aligned}$$

where u_b is the bias term which will be set to u_{ff} to support feedforward for controllers without integral action. Note also that set-point weighting ha removed for this case.

7.2 Grafchart for PID Control and Education

The PID controller concept is taught in most introductory automatic control courses and given its wide use it is important that the students get a deep understanding for PID control. The availability of interactive learning tools in these courses play an important role. Examples of interactive learning tools in the field of automatic

control are ICtools [104] and Interactive Learning Modules for PID [105]. ICtools is an interactive MATLAB based tool for learning the basics of automatic control whereas Interactive Learning Modules for PID is a learning tool which covers many aspects of PID controller design and tuning in an intuitive and interactive way. Students should also be familiarized with PID controllers in a realistic setting, that is, controlling a physical process. If a physical process is not available, although this is not ideal, an animated simulation of the process can be used instead. The idea is that the students control the process, and by doing so develop an understanding of the basic ideas of automatic control and get an intuitive feel for how the PID controller and its parameters work.

A reusable PID implementation and an interactive PID learning module for educational purposes have been implemented in JGrafchart. The learning module is designed for users without any knowledge about JGrafchart. Unlike MATLAB, JGrafchart is free and based on an industrial control language. Unlike many other learning tools, JGrafchart can be connected to physical processes and can be used in industrial environments.

PID Implementation

A full-fledged PID controller has been implemented as a reusable Grafchart Procedure which means that it can be used for any number of control loops by adding a Procedure Step or Process Step for each control loop. The Procedure is called `PID` and is shown in Figure 7.3. The Procedure parameters are explained in Table 7.1.

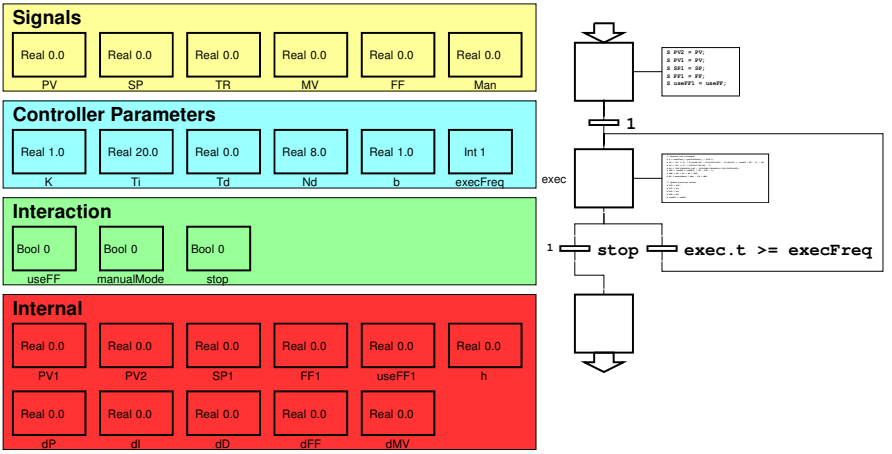


Figure 7.3 The new `PID` Procedure. The parameter values here are the default values. See Table 7.1 for parameter descriptions. The step actions are not intended to be readable in the printed version.

Table 7.1 Parameter descriptions for the parameters of the `PID` Procedure in Figure 7.3.

Parameter	Description
PV	Process Value, the measurement value for the process.
SP	Set-Point, the reference value.
TR	TRacking signal, the actually actuated control signal in the previous scan cycle.
MV	Manipulated Variable, the control signal.
FF	FeedForward, an external signal added to the control signal.
Man	Manual, the control signal when in manual mode.
K	Controller gain.
Ti	Integral time.
Td	Derivative time.
Nd	Maximum derivative gain, usually called N but that is a reserved word in JGrafchart.
b	Set-point weight for the proportional part.
execFreq	Execution Frequency, the PID controller sample time is `execFreq` times the calling Grafchart application's scan cycle time.
useFF	Signal to turn feedforward on/off.
manualMode	Signal to turn manual mode on/off.
stop	Signal to terminate the Procedure call and thus stop executing the PID controller.

A Procedure executes at the same rate as the application it is called from. The PID sample time is thus limited to a multiple of the application's scan cycle time. The PID algorithm is implemented in a single step and can thus execute at the same rate as the caller.

In Procedure calls, each Procedure parameters can be set as either call-by-reference (`R`), call-by-value (`V`), or default (omitted). For call-by-value and default the parameter gets its value when the call is made. For call-by-reference the parameter will be a reference to a variable or I/O in the calling context. To be useful, `PV`, `SP`, `TR`, and `MV` should all be call-by-reference and the caller should set and update all parameters except `MV` which is the `PID` Procedure's sole output.

A call to the `PID` Procedure is shown in Figure 7.4. Here, the parameters for feedforward, manual mode, set-point weighting, and the D-part are omitted and will thus get their default values, which means that these features are not used. Call-by-

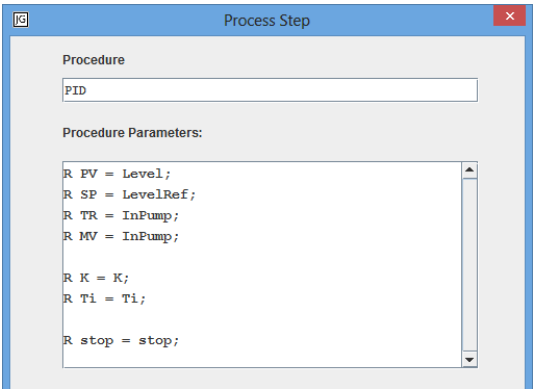


Figure 7.4 A Process Step that calls the `PID` Procedure.

reference is used for all parameters. For example, the parameter `PV` in the call will be a reference to the variable `Level` in the calling context. Changing the value of the variable also updates the parameter and vice versa.

Code The code for the main step (`exec`) of Figure 7.3 is shown in Figure 7.5. It is a straightforward implementation of the discrete equations in Section 7.1. The velocity form was chosen as it requires fewer special cases which means less code, specifically considerably less special code which is rarely executed and thus more likely to contain errors. For no particular reason, backward difference was selected for the integral part which gives $b_{i1} = \frac{Kh}{Ti}$ and $b_{i2} = 0$.

```
// Execute one increment
S h = execFreq * getTickTime() / 1000.0;
S dP = (Ti != 0) ?
         K*(b*SP-PV) - K*(b*SP1-PV1) :
         K*(SP-PV) + (useFF ? FF : 0) - TR;
S dI = (Ti != 0) ? K*h/Ti*(SP-PV) : 0;
S dD = (Td/(Td+Nd*h))*dD -
         (K*Td*Nd/(Td+Nd*h))*(PV-2*PV1+PV2);
S dFF = (useFF & useFF1) ? FF - FF1 : 0;
S dMV = dP + dI + dD + dFF;
S MV = manualMode ? Man : TR + dMV;

// Update previous values
S PV2 = PV1;
S PV1 = PV;
S SP1 = SP;
S FF1 = FF;
S useFF1 = useFF;
```

Figure 7.5 The code for the main step (`exec`) of the `PID` Procedure in Figure 7.3.

In Figure 7.5, first the current PID controller sample time is calculated. Then, the P, I, D, and feedforward increments are calculated. `dP` and `dI` have special handling if the integrator is turned off as discussed in Section 7.1. The feedforward calculation has special handling to avoid bumps when feedforward is enabled at the

same time as the feedforward value is changed. After this, the whole control signal increment dMV is calculated and then the control signal is selected depending on the current mode, that is, manual or automatic. Note that TR is typically the limited MV from the previous scan cycle and that the caller is responsible to update TR. Finally, values for use in following samples are stored in internal variables. The suffix tells how many scan cycles old the value is, for example PV1 is PV from the previous scan cycle.

Execution Order Using the PID Procedure properly requires that the proper execution order is ensured, that is:

1. Execute the PID controller.
2. Limit the control signal.
3. Update the simulated process.

The PID Procedure uses S actions to make it execute as early as possible (step 4 in the execution model). Hence, to implement a process simulation or to model control signal limitations, P actions can be used (step 6 in the execution model). To ensure that an update of a simulated process uses the limited control signal, the limiting should be a preceding P action in the same step as the simulation update.

Educational Aspects

For educational tools it is paramount that they are freely available and easy for anyone to run. The interactive PID learning module presented only requires the freely available and platform independent software JGrafchart which has been verified to run on both Windows, Linux, and Mac. The PID Procedure and the interactive module are both included in JGrafchart. The learning module can be started directly with a separate shortcut or through the menu in JGrafchart.

So far, the learning module only consists of one exercise which contains a simulated tank process similar to the upper tank of the tank process used in the introductory automatic control course at the department. The process consists of a water tank and the objective is to control the tank's water level. The inflow to the tank is controlled with a pump and there is an outflow through a hole in the bottom of the tank. Figure 7.6 shows a screenshot of the exercise. There is a live animation showing the current state of the (simulated) process and both set-point and basic control parameters can be changed while executing. This is a suitable exercise for beginners.

It is possible to create and use simulated processes for hands on or laboratory exercises. With JGrafchart's capability to connect to external environments it is also possible to use the same implementation to control the actual process. One possibility is to control a physical process in a laboratory exercise and provide a simulation of the physical process for preparatory or followup work. Another possible use is live demonstrations in lectures.

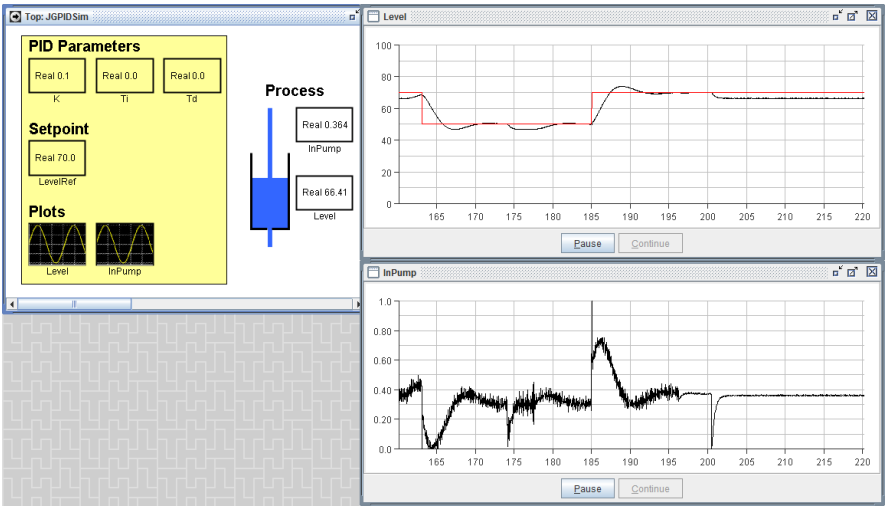


Figure 7.6 In the left part, the set-point and controller parameters can be changed and an animation of the simulated tank process is shown. The upper plot shows the measurement value (black) and the set-point (red). The lower plot shows the control signal. In this screenshot, the set-point and controller parameters were recently changed several times.

Conclusions

The `PID` Procedure has been added to a control library that is included with JGrafchart to make it easy to use Grafchart for PID control. It is a full-fledged PID which supports most of the common PID features such as anti-windup, tracking, feedforward, set-point weighting, automatic/manual mode, and bumpless parameter and mode changes. Possible extensions to the current `PID` Procedure are, for example, an optional process value filter or more advanced features, such as an auto-tuner.

The PID controller is taught in most introductory automatic control courses. To aid learning of new concepts, interactive learning tools play an important role. The interactive PID module presented aims to aid obtaining both an understanding of the PID controller concept and an intuitive feel for PID parameter tuning in an industrial setting. Unlike many other learning tools, JGrafchart is free, based on an industrial control language, and can be used in industrial environments. The learning module is freely available and has been designed for users without any knowledge about JGrafchart. Future work includes evaluating the learning module by including it in the education as well as extending the learning module with more exercises to include more of the features already implemented in the `PID` Procedure.

7.3 A PID Algorithm Improvement

In this section, an issue analogous to integrator windup is presented. The issue affects the derivative part when the process value saturates and has been named derivative backoff. First, derivative backoff will be explained in detail, followed by a discussion about various attempts to solve the issue. Finally, a solution is proposed which only requires that the controller also has an integral part and that it knows the process value limits.

Derivative Backoff

Knowledge about future process behavior can be used to get better control performance. The derivative part of the PID controller predicts the process value T_d seconds in the future by considering the current process value derivative. For example, if the process value is increasing the process value derivative is positive and thus the derivative action will be negative, see (7.1). In essence the derivative part counteracts fast process movements, which can be used to, for example, avoid overshoots.

The derivative part depends on accurate process value measurements. Noisy measurements lead to noisy derivative action. To suppress the effect of measurement noise the process value is usually filtered. A saturated process value means that there is no information about the current process value derivative and thus the derivative part breaks down. At saturation, the derivative of the measured process value is 0, which means that there is no derivative action. This makes sense as there is no information about future process values. However, the behavior when the process value saturates makes less sense as can be seen in Figure 7.7. Here the process is controlled close to the process value limit (100) when a large load disturbance occurs. Just before the process value saturates the process value derivative is positive which means that the derivative action is negative. Shortly after the process value has saturated the derivative part is 0, that is, there is no derivative action. In other words, when the process value reaches its maximum the derivative part backs off and thus the control signal is increased. This is not a desirable behavior.

The control signal change due to derivative backoff is as large as the derivative part was just before the saturation. The change will be close to a step if the derivative part filter has high bandwidth, while a low bandwidth filter gives a smoother change.

In the example in Figure 7.7 the process $P = \frac{1}{(s+1)^4}$ and its corresponding MIGO design method PID parameters with $M = 1.4$ from [103] have been used. The process has been discretized with zero-order hold with a sample time of $h = 0.04$ s. The controller parameters are $K = 1.19$, $T_i = 2.22$, and $T_d = 1.21$. A first order filter for the derivative part has been used ($\frac{sT_d}{1+sT_d/N}$) with maximum derivative gain $N = 10$.

Derivative backoff occurs whenever the process value saturates, for example, due to a load disturbance as seen in the previous example or due to a set-point change. It is easy to find examples where the effects of derivative backoff are much

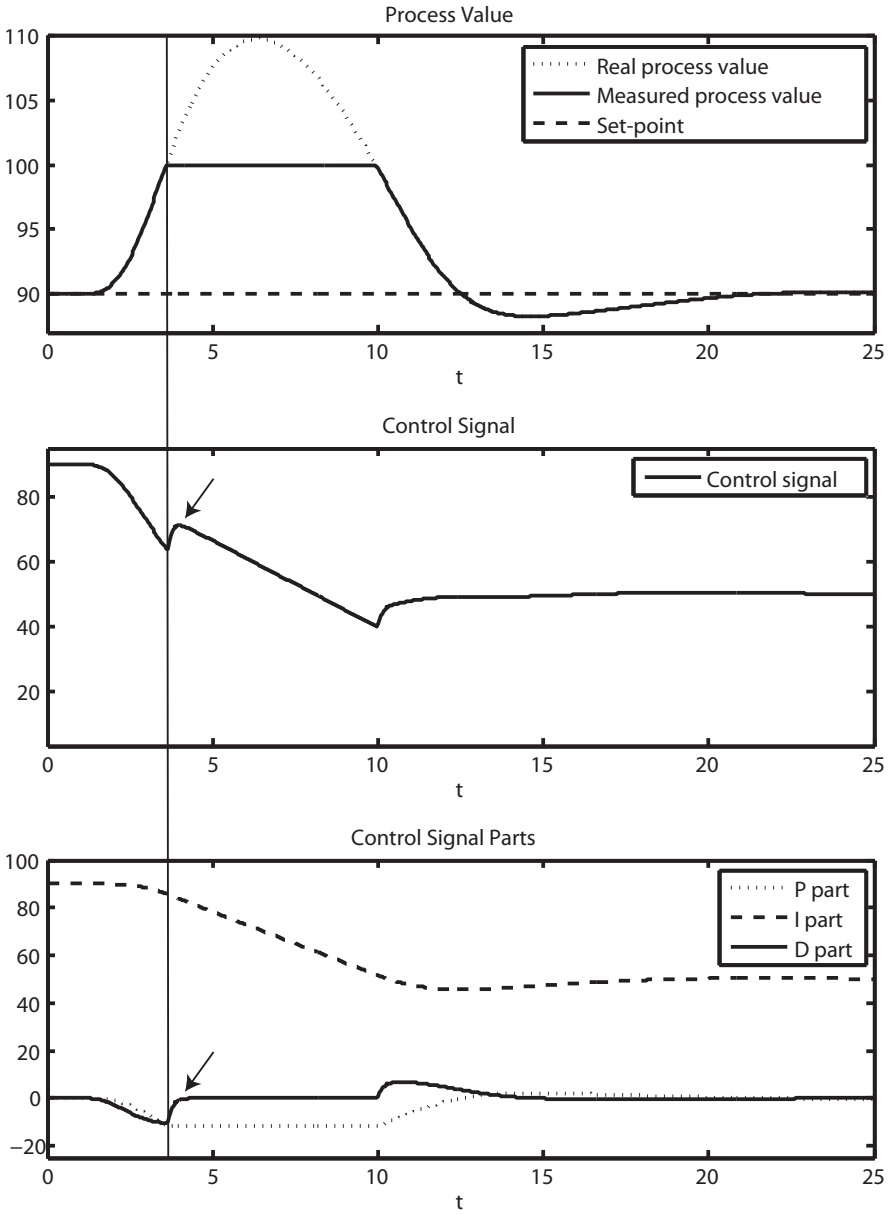


Figure 7.7 An example where a load disturbance causes derivative backoff to occur. When the process value saturates, the derivative part backs off and thus the control signal is increased. The process value is thus pushed further away from the measurement range.

more severe by selecting a favorable process and hand tuning the controllers. However, this example is meant to demonstrate what the issue looks like and when it occurs, as well as show that it is an issue, not only for a hand constructed example, but for relevant processes with well tuned PID controllers as well. Note that the controller parameters from the MIGO design method are optimized to handle the kind of disturbance that was applied.

Process Value Saturation Handling

Theoretically, the best way to avoid derivative backoff is to base the derivative action on an accurate process model which does not saturate, see Figure 7.8. However, in the typical industrial setting, very limited or no models are available and this approach must thus be discarded. Also note that it is time consuming to build an accurate model.

To avoid derivative backoff without a model, the derivative part should be kept when the process value saturates. A naive approach would be to simply not update the derivative part during the saturation. This has the disadvantage that when the process value is no longer saturated, there is initially derivative action in the wrong direction. As the sign of the process value derivative has then changed, the old derivative action will vanish quickly. This is similar to derivative backoff, but occurs when the process value stops being saturated instead of when it becomes saturated, see Figure 7.9.

It is desirable to keep the derivative part while still starting over from no derivative action when the process value is no longer saturated. To simply reset the derivative part when the process value is no longer saturated would cause derivative backoff then instead. As the whole backoff will be applied as a step this is even worse than keeping the derivative part constant.

Anti-backoff The proposed solution to derivative backoff is to disable and immediately re-enable the derivative part bumplessly, see Figure 7.10. Practically this means that the derivative part is moved to the integral part, pseudocode is shown in Figure 7.11. Note that the derivative part then starts over from 0 and that it will remain 0 as long as the process value is saturated. An advantage with this approach is that no new controller parameters are needed. Note also that this solution only works for controllers with an integral part and that the controller must know the physical process value limits. This holds for most PID controllers with a derivative part.

Controllers Without Integral Action For controllers without an integral part the proposed solution cannot be used as it is not possible to disable the derivative part bumplessly. The closest thing to the integral part for a PD controller is the bias term, basically a manually selected constant term, u_b, that replaces the integral term in (7.1), see (7.8). However, to automatically modify the bias term would mean that the bias is wrong when stationarity is reached again.

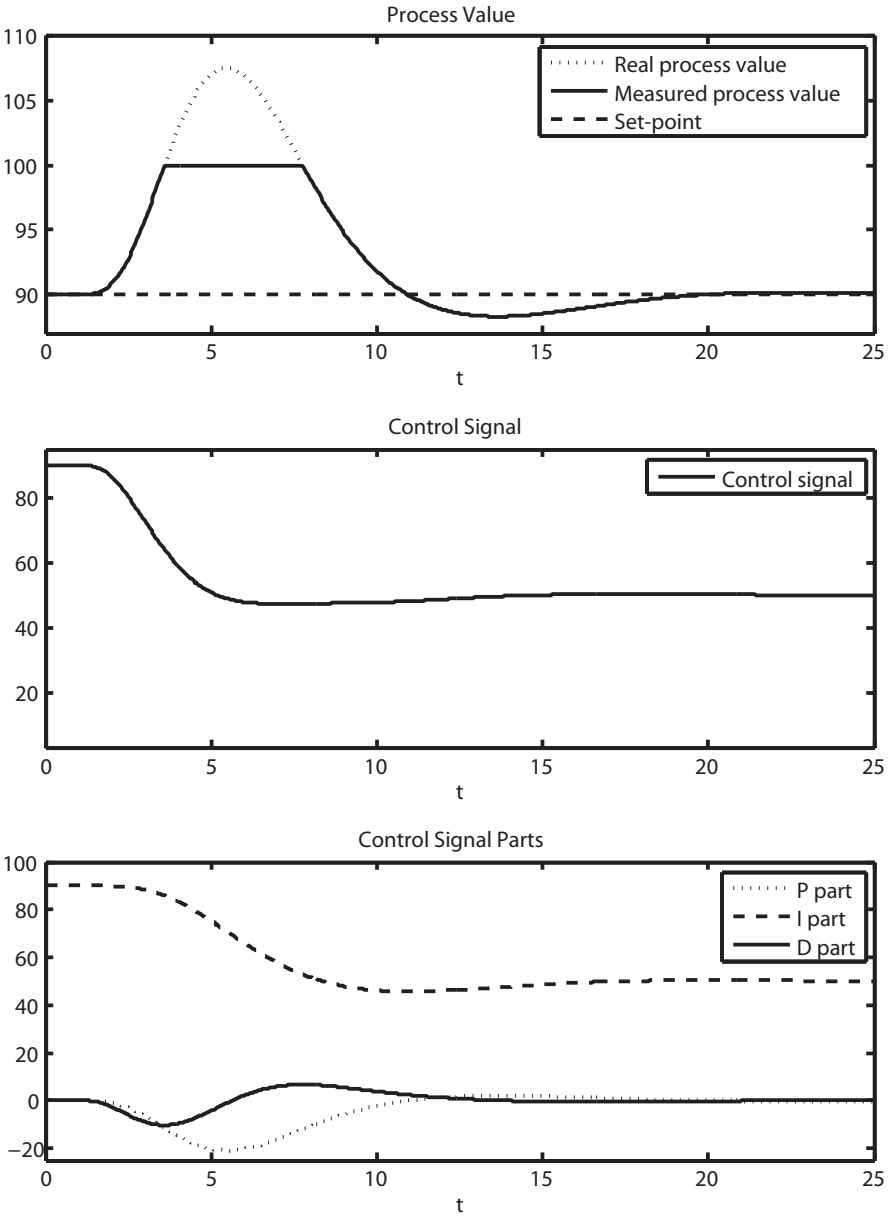


Figure 7.8 The same experiment as in Figure 7.7 using a perfect process model with an unbounded range, that is, the PID controller uses the real process value (dotted) instead of the measured process value (solid).

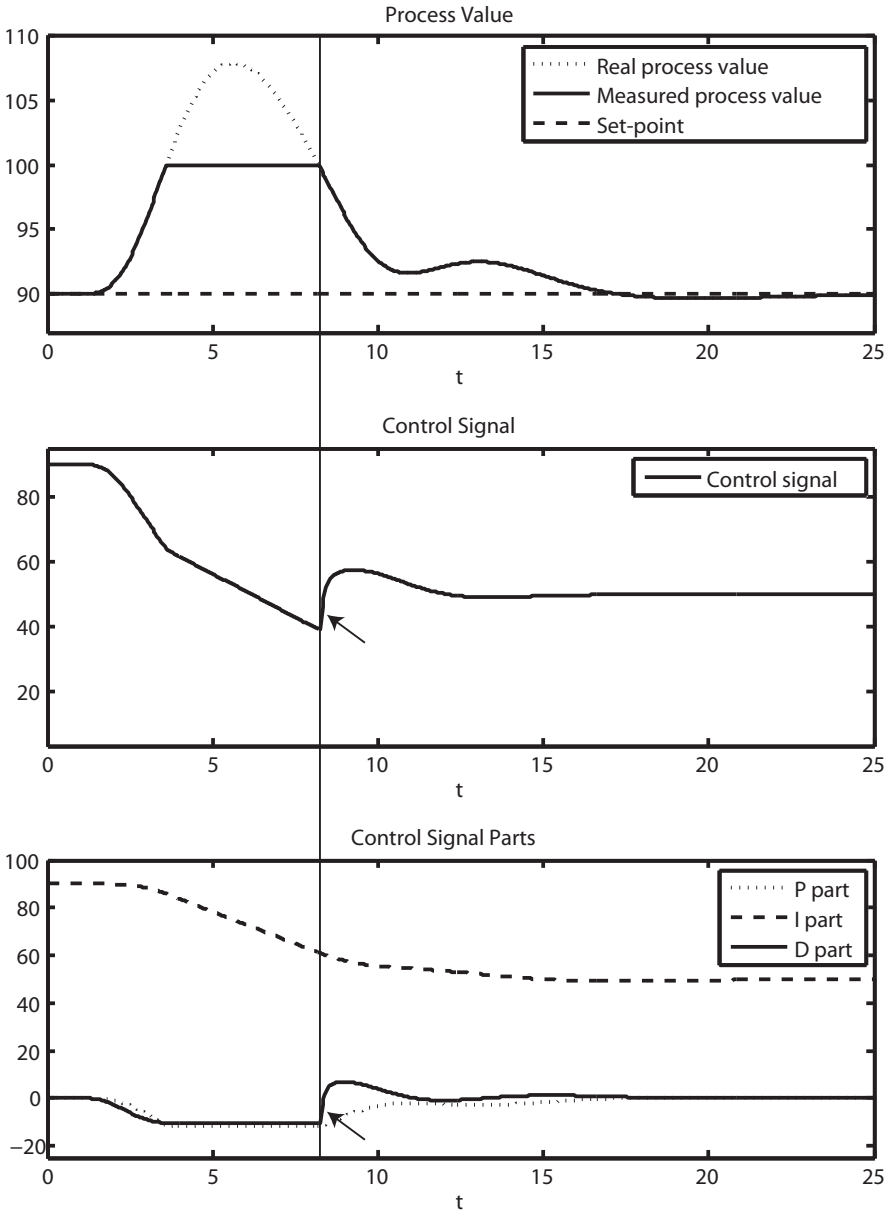


Figure 7.9 The same experiment as in Figure 7.7 but keeping the derivative part constant during the saturation. Now the derivative part backs off when the process value is no longer saturated instead.

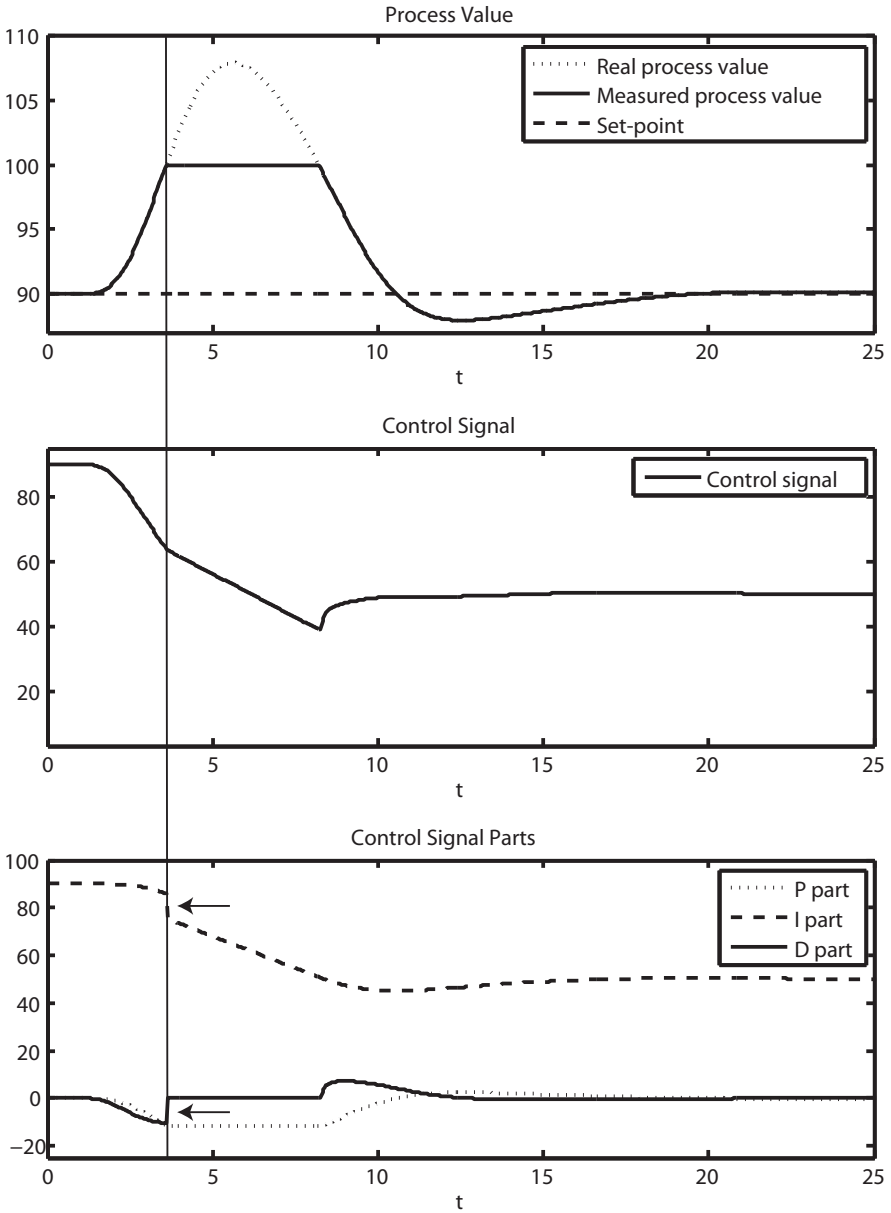


Figure 7.10 The same experiment as in Figure 7.7 where the derivative part is bumplessly disabled and immediately re-enabled when the process value saturates. Practically, the derivative part is moved to the integral part.

```
// Anti-backoff
if PV = PVMax or PV = PVMin:
    I := I + D
    D := 0

// Ordinary PID implementation
ad := Td/(Td + Nd·h)
bd := K·N·ad
e  := SP - PV
P  := K·e
I  := I + K·h/Ti·e
D  := ad·D + bd·(PV - PVOld)
u  := P + I + D
PVOld := PV
```

Figure 7.11 Pseudocode for implementing anti-backoff. The first lines are prepended to an ordinary PID controller implementation. `SP` is the set-point, `PV` is the process value, `PVMax` and `PVMin` are the process value limits, and `PVOld` is `PV` from the previous sample. The ordinary PID controller here is simple and does not even contain anti-windup. However, the modification is the same for a complete PID implementation.

Another alternative is to increase the time constant for the derivative part filter and thus have a slower backoff when the process value saturates. The derivative part will then still back off, but more smoothly. A disadvantage with this is that it must be decided how much slower it should back off, possibly with a new controller parameter. The parameter would in essence be a trade-off between normal derivative backoff, Figure 7.7, and keeping the derivative action constant, Figure 7.9. The new parameter is unlikely to be tuned and must therefore have a good default value. Note that for this approach the backoff is still exponential. Yet another alternative is to have a linear backoff instead and to either configure the backoff rate or the backoff time. This is more similar to how the integral part works in the PID case. The drawback is again that a new controller parameter is needed. However, this parameter has a more intuitive meaning and is thus easier to tune than a filter slowdown parameter.

Conclusions

For PID controllers with integral action, the control signal range must be considered to avoid anti-windup. In this section, it has been shown that for PID controllers with derivative action, the process value range must be considered to avoid derivative backoff. The proposed solution is to disable and immediately re-enable the derivative part bumplessly when the process value saturates. This requires that the PID controller has an integral part and that it knows the process value limits. This holds for most PID controllers with a derivative part. Controllers without integral action were discussed and several approaches were discarded. It remains to be investigated what a good solution is for controllers without integral action.

8

Summary

The starting point for this thesis was current market trends for industrial automation. Applications from both process automation, manufacturing, and robotics were considered. The trends are:

1. Customizable products.
2. Shorter time to market.
3. Powerful global competitive pressure.

For these trends the following challenges were identified:

1. Flexible production systems
2. Integration and utilization of devices and software

The challenge flexible production systems was addressed in Chapter 5 by considering flexible languages and tools, in particular Grafchart and JGrafchart.

The flexibility of the language was improved by adding transition priorities and macro step resume mode. Transition priorities make it easier to ensure that alternative transitions do not conflict. They can also be used to make the transition guard conditions more concise. The macro step resume mode makes it possible to choose how and when to resume an aborted macro steps. It can be used to get a shallow or deep resume or something in between. It is also possible to choose if activation through an abortive Step Fusion Set should trigger a resume.

The compilers for JGrafchart were rewritten using modern compiler techniques and tools, namely ReRAGs and JastAdd, to make the language semantics extensible. First the compilers for the textual action and condition languages were considered. They were previously implemented with traditional compiler techniques and tools. The compiler implementations were first moved to the JastAdd tool and then the semantics were simplified with ReRAGs to make it extensible. Extensibility was confirmed by adding a small extension. The new implementation was evaluated

with regards to code size and compilation time. The new implementation was 86% smaller and its compilation time was 130% longer.

To make it possible to use Grafchart for hard real-time control, real-time execution with JGrafchart was considered. As a first step the editor, compiler, and execution engine were separated. To make the compiler stand-alone, the compiler for the function chart language which was previously implemented with handwritten code was moved to JastAdd and simplified with ReRAGs. The execution engine is still an interpreter but it is now completely separated from the graphics. Also, concurrency for Grafchart application execution was investigated and several fundamental concurrency issues were identified. A conservative approach to avoid concurrency issues and thus achieve deterministic execution was presented.

With the JGrafchart compiler implemented with ReRAGs it is possible to create various extensions. Extensions of particular interest are High-Level Version and code generation for hard real-time control. Generation of SFC code which can be executed by industrial PLCs is also of interest.

Access to more data from industrial devices and software can be used to optimize production. The challenge integration and utilization of devices and software was addressed in Chapter 6 by considering architectures for factory integration as they are the foundation to connect all devices. The flexible software architecture SOA which is widely used for IT systems and business processes was considered. The two SOA service technologies DPWS and OPC UA are recognized to be required to cope with the industrial requirements on device level [65]. These are the service technologies that have been considered.

The mandatory web service extensions of DPWS are sufficient to create a generic client for service orchestration that can use any DPWS device. Prior to this work no such known tool existed for industrial automation. JGrafchart was considered a suitable candidate for this and has thus been extended with generic DPWS support. This makes it possible to use any DPWS capable device from JGrafchart and service orchestration can be implemented entirely in a JGrafchart application. The DPWS support is well integrated. DPWS operations are called in the same way as other JGrafchart element methods. The integrated DPWS support was evaluated on a demonstrator with real industrial field devices and has since been used for implementation of more smart factory demonstrators.

OPC UA is a recent interoperability standard for industrial automation. It incorporates the three classic OPC standards and adds SOA features. Generic support for OPC UA has been added to JGrafchart and thus a new way to implement service orchestration for OPC UA services has been created. OPC UA data access and methods are supported. The OPC UA support is well integrated in JGrafchart and feels like a natural part of the language. The new OPC UA support was used to successfully control an embedded device that had been retrofitted to become service capable. JGrafchart can be used to control any device that is accessible through an OPC UA server. With an adapter for classic OPC this includes the field devices used by industrial control systems that support classic OPC. JGrafchart now sup-

ports both integrated DPWS and OPC UA and can thus be used regardless of which is more suitable in a specific scenario.

SOA 2.0 which is event driven and features extremely loose coupling between applications has also been considered. The Line Information System Architecture, LISA, is based on SOA 2.0 and has been developed with focus on industrial manufacturing. The goals of LISA has been to make it easy to integrate any device or software, especially legacy devices which are common in factories. The core components of LISA are the message bus, communication endpoints, and the LISA message format. There are no direct connections between devices in the LISA architecture, all communication goes through the message bus. A devices is integrated through a communication endpoint, which is an adapter between the LISA message format and message bus and the device's message format and communication protocol. The LISA message format is designed to be simple and flexible. It also enforces as little structure as possible to ensure that it is usable for any device. LISA was evaluated on a system with various devices and with applications on different IEC 62264 levels. It was concluded that an event-based service oriented architecture such as LISA is useful for industrial automation. With access to raw event data, it is possible to implement data processing wherever it is most convenient. With the history of raw events it is also possible to add new calculations retroactively.

Another step toward real-time execution of Grafchart applications was to integrate support for the high performance communication protocol LabComm. Both versions of LabComm are supported as well as special connection through an Orca server. Less knowledge about LabComm is needed to use LabComm in JGrafchart than in for example Java as much is taken care of automatically. Most LabComm types are supported and LabComm sample data is accessed like other hierarchical structures in JGrafchart.

It has been investigated how JGrafchart can be connected to FMI for co-simulation to further address the shorter time to market trend by introducing simulation support. For a well integrated support, changes to JGrafchart are required to make it possible to redo communication steps, to support externally triggered execution, and to inspect simulation results. The DPWS, OPC UA, and LabComm I/O will also need some special treatment to fit into the FMI framework.

Work related to PID control was presented in Chapter 7. An implementation of the most common controller for industrial automation, the PID controller, has been added to a built-in Grafchart library. Also, a new PID algorithm issue for PID controllers with a derivative part, derivative backoff, was explained in detail. It occurs when the process value saturates and the cause is that the process value derivative then suddenly becomes 0. This means that the derivative part backs off which causes undesired changes to the control signal. It was shown that this is an issue for an industrially relevant process controlled by a well tuned PID controller. Different approaches to solve the issue were discussed. The proposed solution is to bumplessly disable and immediately re-enable the derivative part when the process value saturates. The solution requires that the controller has an integral part and that

it knows the process value limits. It remains to be investigated what a good solution is for controllers without integral action.

Bibliography

[1] Docutils. *Docutils: Documentation Utilities*. URL: http://docutils.sourceforge.net/ (visited on 2013-04-03).

[2] Automation Federation. *What is automation?* URL: http://www.automationfederation.org/Content/NavigationMenu/General_Information/Alliances_and_Associations/The_Automation_Federation/About1/What_is_Automation_/What_is_Automation_.htm (visited on 2014-09-13).

[3] Rucker, J.d. *Car buying tips: the very best way to buy a new car*. URL: http://www.streetdirectory.com/travel_guide/214317/car_buyer/car_buying_tips_the_very_best_way_to_buy_a_new_car.html (visited on 2013-01-25).

[4] SmartFactory^{KL}. *SmartFactory^{KL}*. URL: http://smartfactory.dfki.uni-kl.de/en (visited on 2013-03-29).

[5] P. Tabuada. "Cyber-physical systems: position paper". In: *NSF Workshop on Cyber-Physical Systems*. 2006.

[6] Wikipedia. *Internet of Things*. URL: http://en.wikipedia.org/wiki/Internet_of_Things (visited on 2014-10-05).

[7] ISA. *ANSI/ISA-95, Enterprise-Control System Integration*. Tech. rep. The International Society of Automation, 2010.

[8] IEC. *IEC 62264-1: Enterprise-Control System Integration – Part 1: Models and Terminology*. Tech. rep. International Electrotechnical Commission, 2013.

[9] IEC. *IEC 61131-3: Programmable controllers – Part 3: Programming Languages Ed2.0*. Tech. rep. International Electrotechnical Commission, 2003.

[10] Stora Enso. *Stora Enso plans profitability improvement actions across all business areas*. URL: http://www.storaenso.com/media-centre/press-releases/2012/10/Pages/stora-enso-plans-profitability-improvement.aspx (visited on 2013-01-28).

[11] ABB. *The product life cycle*. URL: http://www.abb.se/product/ap/seitp334/caab7ea34fea011ec1257919004976a9.aspx (visited on 2013-01-28).

[12] K. Soltesz. *On Automation of the PID Tuning Procedure*. Licentiate Thesis ISRN LUTFD2/TFRT--3254--SE. Department of Automatic Control, Lund University, Sweden, 2012.

[13] K. Åström and R. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, Princeton and Oxford, 2012. ISBN: 9780691135762. URL: http://www.cds.caltech.edu/~murray/amwiki.

[14] A. Boyd, D. Noller, P. Peters, D. Salkeld, T. Thomasma, C. Gifford, S. Pike, and A. Smith. "SOA in manufacturing guidebook". *MESA International, IBM Corporation and Capgemini Co-Branded White Paper* (2008).

[15] L. Ribeiro, J. Barata, and P. Mendes. "MAS and SOA: complementary automation paradigms". In: *Innovation in manufacturing networks*. Springer, 2008, pp. 259–268.

[16] D. Krafzig, K. Banke, and D. Slama. *Enterprise SOA: Service-Oriented Architecture Best Practices*. The Coad Series. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004. ISBN: 9780131465756.

[17] N. Bieberstein, S. Bose, M. Fiammante, K. Jones, and R. Shah. *Service-Oriented Architecture Compass: Business Value, Planning, and Enterprise Roadmap*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005. ISBN: 0131870025, 9780131870024.

[18] H. Mersch, M. Schlutter, and U. Epple. "Classifying services for the automation environment". In: *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*. 2010, pp. 1–7. DOI: 10.1109/ETFA.2010.5641170.

[19] L. M. S. De Souza, P. Spiess, D. Guinard, M. Köhler, S. Karnouskos, and D. Savio. "SOCRADES: a web service based shop floor integration infrastructure". *Networks* **4952** (2008). Ed. by C. Floerkemeier, M. Langheinrich, E. Fleisch, F. Mattern, and S. E. Sarma, pp. 50–67. URL: http://www.springerlink.com/index/05581001K35585K4.pdf.

[20] T. Kirkham, D. Savio, H. Smit, R. Harrison, R. Monfared, and P. Phaithoonbuathong. "SOA middleware and automation: services, applications and architectures". In: *Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on*. 2008, pp. 1419–1424. DOI: 10.1109/INDIN.2008.4618326.

[21] L. Ollinger, J. Schlick, and S. Hodek. "Leveraging the agility of manufacturing chains by combining process-oriented production planning and service-oriented manufacturing". In: *Proceedings of the 18th IFAC World Congress*. Elsevier Science Ltd., 2011.

[22] Y. V. Natis. *Service-Oriented Architecture Scenario*. Tech. rep. Gartner, Inc., 2003. URL: https://www.gartner.com/doc/391595/serviceoriented-architecture-scenario.

[23] B. M. Michelson. “Event-driven architecture overview”. *Patricia Seybold Group* **2** (2006).

[24] R. Kuhn and J. Allen. *Reactive Design Patterns*. MEAP 2. Manning Publications, 2014.

[25] N. Jazdi. “Cyber physical systems in the context of Industry 4.0”. In: *Automation, Quality and Testing, Robotics, 2014 IEEE International Conference on*. 2014. DOI: 10.1109/AQTR.2014.6857843.

[26] *Industrie 4.0 - Whitepaper FuE-Themen*. Tech. rep. Plattform Industrie 4.0, 2014.

[27] German Federal Ministry of Education and Research (BMBF). *Project of the Future: Industry 4.0*. URL: http://www.bmbf.de/en/19955.php (visited on 2014-10-03).

[28] *Recommendations for implementing the strategic initiative Industrie 4.0*. Tech. rep. Plattform Industrie 4.0, 2013.

[29] IEC. *IEC 60848: GRAFCET specification language for sequential function charts ed3.0*. Tech. rep. International Electrotechnical Commission, 2013.

[30] Wikipedia. *Petri net*. URL: http://en.wikipedia.org/wiki/Petri_net (visited on 2013-03-18).

[31] A. W. Appel and J. Palsberg. *Modern Compiler Implementation in Java*. 2nd. Cambridge University Press, New York, NY, USA, 2002. ISBN: 0-521-82060-X.

[32] D. Harel. “Statecharts: a visual formalism for complex systems”. *Science of computer programming* **8**:3 (1987), pp. 231–274.

[33] Wikipedia. *Business Process Model and Notation*. URL: http://en.wikipedia.org/wiki/Business_Process_Model_and_Notation (visited on 2013-04-08).

[34] Department of Automatic Control, Lund University. *Grafchart*. URL: http://control.lth.se/Research/tools/grafchart.html (visited on 2013-02-07).

[35] C. Johnsson. *A Graphical Language for Batch Control*. PhD thesis 1051. Department of Automatic Control, Lund Institute of Technology, Sweden, 1999.

[36] Gensym. *Gensym G2*. URL: http://www.gensym.com/product/G2 (visited on 2013-03-10).

[37] K.-E. Årzén, R. Olsson, and J. Åkesson. “Grafchart for procedural operator support tasks”. In: *Proceedings of the 15th IFAC World Congress, Barcelona, Spain*. 2002.

[38] ROSETTA. *The ROSETTA project*. URL: http://www.fp7rosetta.org/ (visited on 2014-10-04).

[39] A. Stolt. *Robotic Assembly and Contact Force Control*. Licentiate Thesis ISRN LUTFD2/TFRT--3256--SE. Department of Automatic Control, Lund University, Sweden, 2012.

[40] A. Benktson and S. Dahlberg. *Modeling of Avionics Systems using JGrafchart and TrueTime*. Master's Thesis ISRN LUTFD2/TFRT--5907--SE. Department of Automatic Control, Lund University, Sweden, 2012.

[41] I. Dressler. *Code Generation from JGrafchart to Modelica*. Master's Thesis ISRN LUTFD2/TFRT--5726--SE. Department of Automatic Control, Lund University, Sweden, 2004.

[42] A. Llorente. *Code Generation from JGrafchart to ATMEL AVR*. Master's Thesis ISRN LUTFD2/TFRT--5749--SE. Department of Automatic Control, Lund University, Sweden, 2005.

[43] myvision. *myvision: JGrafchart*. URL: http://www.myvision.de/startseite/fertigungssteuerung/jgrafchart/ (visited on 2014-10-04).

[44] R. W. Lewis. *Programming Industrial Control Systems Using IEC 1131-1*. 2nd Edition. The Institution of Engineering and Technology, 1998. ISBN: 978-0852969502.

[45] R. Olsson. *Batch Control and Diagnosis*. PhD thesis ISRN LUTFD2/TFRT--1073--SE. Department of Automatic Control, Lund University, Sweden, 2005.

[46] A. Theorin, K.-E. Årzén, and C. Johnsson. "Rewriting JGrafchart with Rewritable Reference Attribute Grammars". In: *Industrial Track of Software Language Engineering 2012*. Dresden, Germany, 2012.

[47] Robert D. Cameron. *Four concepts in programming language description: syntax, semantics, pragmatics and metalanguage*. URL: http://www.cs.sfu.ca/~cameron/Teaching/383/syn-sem-prag-meta.html (visited on 2013-03-24).

[48] D. E. Knuth. "Semantics of context-free languages". *Theory of Computing Systems* **2**:2 (1968-1), pp. 127–145. ISSN: 0025-5661. DOI: 10.1007/BF01692511. URL: http://dx.doi.org/10.1007/BF01692511.

[49] G. Hedin. "An introductory tutorial on JastAdd attribute grammars". In: *Generative and Transformational Techniques in Software Engineering III*. Springer, 2011. ISBN: 978-3-642-18022-4.

[50] T. Ekman and G. Hedin. "The JastAdd extensible Java compiler". In: *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. OOPSLA '07. ACM, Montreal, Quebec, Canada, 2007, pp. 1–18. ISBN: 978-1-59593-786-5. DOI: http://doi.acm.org/10.1145/1297027.1297029. URL: http://doi.acm.org/10.1145/1297027.1297029.

[51] J. Öqvist. *Implementation of Java 7 Features in an Extensible Compiler*. Master's Thesis ISSN 1650-2884, LU-CS-EX: 2012-13. Department of Computer Science, Lund University, Sweden, 2012.

[52] T. Ekman. "Design and implementation of object-oriented extensions to the Control Module language". In: *11th Nordic Workshop on Programming and Software Development Tools and Techniques*. 2004.

[53] J. Åkesson. "Optimica—an extension of Modelica supporting dynamic optimization". In: *In 6th International Modelica Conference 2008*. Modelica Association, 2008.

[54] R. W. Gray, S. P. Levi, V. P. Heuring, A. M. Sloane, and W. M. Waite. "Eli: a complete, flexible compiler construction system". *Communications of the ACM* **35**:2 (1992), pp. 121–130.

[55] T. Reps and T. Teitelbaum. "The synthesizer generator". *ACM Sigplan Notices* **19**:5 (1984), pp. 42–48.

[56] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. "Silver: an extensible attribute grammar system". *Science of Computer Programming* **75**:1 (2010), pp. 39–54.

[57] T. Sloane. "Experiences with domain-specific language embedding in Scala". *Domain-Specific Program Development* (2008).

[58] JavaCC. *Java compiler compiler (JavaCC) - the Java parser generator*. URL: http://javacc.java.net (visited on 2013-03-24).

[59] junit.org. *JUnit*. URL: http://junit.org (visited on 2013-04-22).

[60] W. Cunningham. "The WyCash portfolio management system". In: *ACM SIGPLAN OOPS Messenger*. Vol. 4. 2. ACM. 1992, pp. 29–30.

[61] Department of Computer Science, Lund University. *LabComm*. URL: http://wiki.cs.lth.se/moin/LabComm (visited on 2014-09-12).

[62] L. Ollinger, J. Schlick, and S. Hodek. "Konzeption und praktische Anwendung serviceorientierter Architekturen in der Automatisierungstechnik". In: *VDI-Berichte 2143. VDI Automatisierungskongress (AUTOMATION-2011), June 28-29, Baden-Baden, Germany*. VDI Verlag, 2011.

[63] OASIS. *Devices Profile for Web Services Version 1.1*. Tech. rep. Organization for the Advancement of Structured Information Standards, 2009.

[64] E. Zeeb, A. Bobek, H. Bohn, and F. Golatowski. "Lessons learned from implementing the Devices Profile for Web Services". In: *Digital EcoSystems and Technologies Conference, 2007. DEST '07. Inaugural IEEE-IES*. 2007, pp. 229–232. DOI: 10.1109/DEST.2007.371975.

[65] G. Candido, F. Jammes, J. de Oliveira, and A. Colombo. "SOA at device level in the industrial domain: assessment of OPC UA and DPWS specifications". In: *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*. 2010, pp. 598–603. DOI: 10.1109/INDIN.2010.5549676.

[66] IEC. *IEC 62541: OPC Unified Architecture Specification 1.02*. Tech. rep. International Electrotechnical Commission, 2013.

[67] M. Izaguirre, A. Lobov, and J. Lastra. "OPC-UA and DPWS interoperability for factory floor monitoring using complex event processing". In: *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*. 2011, pp. 205–211. DOI: 10.1109/INDIN.2011.6034874.

[68] F. Jammes, A. Mensch, and H. Smit. "Service-oriented device communications using the *Devices Profile for Web Services*". In: S. Terzis et al. (Eds.). *MPAC*. Vol. 115. ACM International Conference Proceeding Series. ACM, 2005, pp. 1–8. ISBN: 1-59593-268-2.

[69] SIRENA Consortium. *The ITEA SIRENA project*. URL: http://www.sirena-itea.org (visited on 2013-03-26).

[70] SOA4D. *SOA4D Forge*. URL: https://forge.soa4d.org/ (visited on 2013-03-26).

[71] A. Theorin, L. Ollinger, and C. Johnsson. "Service-oriented process control with Grafchart and the Devices Profile for Web Services". In: T. Borangiu et al. (Eds.). *Proceedings of the 14th IFAC Symposium on Information Control Problems in Manufacturing (INCOM'12)*. Elsevier Ltd, Bucharest, Romania, 2012, pp. 799–804. DOI: 10.3182/20120523-3-RO-2023.00131.

[72] A. Theorin, L. Ollinger, and C. Johnsson. "Service-oriented process control with Grafchart and the Devices Profile for Web Services". In: T. Borangiu et al. (Eds.). *Service Orientation in Holonic and Multi Agent Manufacturing and Robotics*. Vol. 472. Studies in Computational Intelligence. Springer Berlin Heidelberg, 2013, pp. 213–228. ISBN: 978-3-642-35851-7. DOI: 10.1007/978-3-642-35852-4_14. URL: http://dx.doi.org/10.1007/978-3-642-35852-4_14.

[73] WS4D. *Stack: WS4D-gSOAP (C/C++)*. URL: http://ws4d.e-technik.uni-rostock.de/page/3/?s=stack (visited on 2013-03-27).

[74] SOA4D Forge. *DPWS4J Core*. URL: https://forge.soa4d.org/projects/dpws4j/ (visited on 2013-03-28).

[75] ISO. *ISO 8601:2004: Data elements and interchange formats – Information interchange – Representation of dates and times*. Tech. rep. International Organization for Standardization, 2004.

[76] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. *XML Path Language (XPath) 2.0 (Second Edition)*. Tech. rep. World Wide Web Consortium (W3C), 2010.

[77] SmartFactory^{KL}. *Keyfinder production line*. URL: http://smartfactory.dfki.uni-kl.de/en/content/demo/technological-demo/plant-industry4 (visited on 2014-09-06).

[78] A. Theorin, J. Hagsund, and C. Johnsson. "Service orchestration with OPC UA in a graphical control language". In: *19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'2014)*. Barcelona, Spain, 2014.

[79] W. Mahnke, S.-H. Leitner, and M. Damm. *OPC Unified Architecture*. 1st. Springer Berlin Heidelberg, 2009. ISBN: 3540688986, 9783540688983. DOI: 10.1007/978-3-540-68899-0.

[80] J. Lange, F. Iwanitz, and T. J. Burke. *OPC from Data Access to Unified Architecture*. VDE-Verlag, 2010. ISBN: 9783800732425.

[81] S.-H. Leitner and W. Mahnke. "OPC UA – service-oriented architecture for industrial applications". *ABB Corporate Research Center* (2006).

[82] Unified Automation GmbH. *Wrapper and proxy*. URL: http://www.unified-automation.com/products/wrapper-and-proxy.html (visited on 2014-04-08).

[83] fischertechnik GmbH. *Punching machine with conveyor belt*. URL: http://www.fischertechnik.de/en/desktopdefault.aspx/tabid-24/41_read-62/usetemplate-2_column_pano/ (visited on 2014-10-01).

[84] Acme Systems srl. *Aria G25 - Low cost Linux Embedded SMD module*. URL: http://www.acmesystems.it/aria (visited on 2014-10-01).

[85] Softing AG. *Softing industrial automation: industrial communication*. URL: http://industrial.softing.com (visited on 2014-04-16).

[86] J. Hagsund. *Implementation of Service Orchestrated control procedures in OPC UA for JGrafchart*. Master's Thesis ISRN LUTFD2/TFRT--5953--SE. Department of Automatic Control, Lund University, Sweden, 2014.

[87] A. Theorin, K. Bengtsson, J. Provost, M. Lieder, C. Johnsson, T. Lundholm, and B. Lennartsson. "An event-driven manufacturing information system architecture". In: *In submission*.

[88] Gregor Hohpe. *Enterprise Integration Patterns*. URL: http://www.enterpriseintegrationpatterns.com (visited on 2014-03-28).

[89] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. A Martin Fowler signature book. Addison-Wesley, Englewood Cliffs, NJ, 2003. ISBN: 9780321200686.

[90] The Apache Software Foundation. *Apache ActiveMQ*. URL: http://www.activemq.apache.org (visited on 2014-03-31).

[91] R. Kuhn. Course Material, Principles of Reactive Programming. https://www.coursera.org/course/reactive, 2013.

[92] PRACE. *PRACE – The Productive Robot Apprentice*. URL: http://prace-fp7.eu/ (visited on 2014-10-04).

[93] Orca Robotics. *Orca Robotics*. URL: http://orca-robotics.sourceforge.net (visited on 2014-09-30).

[94] F. D. Group. *Functional Mock-up Interface for Model Exchange and Co-Simulation – 2.0*. Tech. rep. Modelica Association, 2014.

[95] Modelica Association. *Modelica*. URL: https://www.modelica.org/ (visited on 2013-12-08).

[96] A. Theorin and C. Johnsson. "On extending jgrafchart with support for FMI for co-simulation". In: *10th International Modelica Conference*. Lund, Sweden, 2014.

[97] H. Elmqvist, F. Gaucher, S. E. Mattsson, and F. Dupont. "State machines in modelica". In: *Proceedings of 9th International Modelica Conference, Munich, Germany, September*. 2012, pp. 3–5.

[98] JModelica.org. *PyFMI*. URL: http://www.jmodelica.org/page/4924 (visited on 2013-12-08).

[99] The Regents of the University of California. *JFMI - A Java Wrapper for the Functional Mock-up Interface*. URL: http://ptolemy.eecs.berkeley.edu/java/jfmi/index.htm (visited on 2013-12-08).

[100] Todd Fast, Timothy Wall, Liang Chen. *Java Native Access (JNA)*. URL: https://github.com/twall/jna (visited on 2013-12-08).

[101] A. Theorin and T. Hägglund. "Derivative backoff: a process value saturation problem for PID controllers". *In journal submission* (2014).

[102] A. Theorin and C. Johnsson. "An interactive PID learning module for educational purposes". In: *Proceedings of the 19th IFAC World Congress (IFAC'14), Cape Town, South Africa*. 2014.

[103] K. Åström and T. Hägglund. *Advanced PID Control*. ISA-The Instrumentation, Systems, and Automation Society, 2006. ISBN: 9781556179426.

[104] M. Johansson, K. J. Åström, and M. Gäfvert. "Interactive tools for education in automatic control". *IEEE Control Systems* **18**:3 (1998), pp. 33–40.

[105] J. L. Guzman, K. Åstrom, S. Dormido, T. Hägglund, M. Berenguel, and Y. Piguet. "Interactive learning modules for PID control [lecture notes]". *Control Systems, IEEE* **28**:5 (2008), pp. 118–134.

A

JGrafchart Releases

Table A.1 JGrafchart version history related to the work presented in this thesis.

Version	Description
1.5.2	The previous public version.
1.5.3.4	(Not public) The initial version for this work.
2.0.0	Textual compilers implemented with ReRAGs. Additional compilation checks.
2.0.1	Bug fixes.
2.1.0	DPWS support added.
2.1.1	Bug fixes.
2.1.2	DPWS support improved.
2.2.0	DPWS support improved.
2.3.0	DPWS support improved.
2.4.0	LabComm support added.
2.4.1	Bug fixes.
2.5.0	Added debugging support, control library, and PID learning module.
2.6.0	LabComm support improved.
2.6.1	Bug fixes.
3.0.0	(Future) Separate editor, compiler, and execution engine.