LUND UNIVERSITY

Adapting Grafchart for Industrial Automation

Theorin, Alfred

2013

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

Link to publication

Citation for published version (APA):
Theorin, A. (2013). *Adapting Grafchart for Industrial Automation*. [Licentiate Thesis, Department of Automatic Control]. Department of Automatic Control, Lund Institute of Technology, Lund University.

Total number of authors:
1

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

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

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

LUND UNIVERSITY

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

Adapting Grafchart for Industrial Automation

Adapting Grafchart for Industrial Automation

Alfred Theorin

LUNDS
UNIVERSITET

Department of Automatic Control

To Lisa

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

ISSN 0280–5316
ISRN LUTFD2/TFRT--3260--SE

© 2013 by Alfred Theorin. All rights reserved.
Printed in Sweden.
Lund 2013

Abstract

Current trends in industrial automation are the need for customizable production, vertical integration, more advanced sensors and actuators, and shorter time to market. The currently used control systems and languages for control were developed with a more static production in mind. More flexible languages and tools are needed to get a more flexible production. The flexible graphical programming language Grafchart, based on the IEC 61131-3 standard language Sequential Function Charts (SFC), is considered with the focus to make it usable in an industrial context.

Modern compiler techniques are evaluated for JGrafchart, a Grafchart implementation, with focus on extensible automation language implementations. In particular implementing the High Level Version (HLV) of Grafchart as an extension would make JGrafchart more dynamic and enable further research on HLV.

To make Grafchart possible to use at the lowest levels of automation, realtime execution with JGrafchart is considered. For this to be possible the execution engine must be separated from the editor. In the first step the execution engine is still an interpreter, but an order of magnitude faster than before.

Finally Service Oriented Architecture (SOA), a highly flexible software design methodology widely used for business processes, is brought to automation by integrating support for Devices Profile for Web Services (DPWS) in JGrafchart.

Acknowledgements

First of all I would like to thank my supervisor Charlotta Johnsson for always being enthusiastic about what I accomplish, 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 giving feedback on this thesis.

I would also like 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 and Tobias Gerber for fun and interesting 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 issues.

Special thanks go to my family and friends. In particular I would like to thank my mother, Iréne Theorin, for always believing in me. Finally my very special thanks go to my loving wife, Lisa Theorin, for being wonderful in general and for proofreading and giving feedback on this thesis in particular.

Acronyms

AST	Abstract Syntax Tree, a common way for compilers to represent applications.
DPWS	Devices Profile for Web Services, a minimal set of mandatory web service extensions targeted for resource constrained devices.
FBD	Function Block Diagram, one of the graphical IEC 61131-3 standard programming languages.
FC	Function Chart, an SFC or Grafchart application.
IDE	Integrated Development Environment, a software in which applications can be written, compiled, executed, and debugged.
IEC	International Electrotechnical Commission, a standards organization.
IO	Inputs and Outputs, how applications interact with the external environment.
LD	Ladder Diagram, one of the graphical IEC 61131-3 standard programming languages.
PLC	Programmable Logic Controller, a control system used for industrial automation.
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.
SOA	Service Oriented Architecture, a component based software design methodology that is platform and language independent.
SOA-AT	SOA in Automation Technologies, SOA used for automation.

SLOC	Source Lines Of Code, a metric for the size of software programs.
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

1.1 Background

Wikipedia describes automation as: "the use of machines, control systems, and information technologies to optimize productivity in the production of goods and delivery of services." [2]. Some examples are production of cars, consumer electronics, medicine, plastics, paper, gasoline, and chemicals. They are all produced in large factories and the production is highly automated. Producing with less manual labor often means cheaper production, higher production rate, and more stable product quality. In practice, automation is required to be competitive and stay in business. If you are able to automate more or do it better than your competitors you have a competitive advantage. Even small improvements can potentially generate or save millions of dollars. To improve existing automation and increase the level of automation is thus always a relevant topic. However, the easier tasks have already been taken care of and subsequent improvements are thus more sophisticated.

One current trend in industrial automation 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 variants, and the color. Now it is possible to choose among all the available options and colors and get a completely customized car. This is often cheaper than buying one of the default variants of the product [3]. You do not have to buy the supreme variant to get the less common option that you really want, and you can skip all the options that you do not care about. For the manufacturer this means that a more flexible production is needed. However, the control systems and languages used for control were developed with a more static production in mind.

Another trend is that the field devices, the sensors and actuators, used for production are getting more features and are becoming more intelligent. In the beginning you just connected them and used them and that was really all there was to it. Since then the demand for improved precision, performance, power usage, reliability, and diagnostics has grown. Today, devices might have advanced internal controllers to optimize precision and save energy, several configuration possibilities to optimize performance, and they usually collect data to provide statistics and diag-

nostics which can be used to optimize the production. For example, a device which can signal abnormal operation can help identify faulty or poor quality products and avoid lengthy troubleshooting. This is nothing new, many fieldbuses developed in the 90s have support for configuration and diagnostics. However, the current devices have more configuration and diagnostics, demand more bandwidth, and provide a higher degree of flexibility. With the current fieldbuses, bandwidth is still a limiting factor. To avoid fieldbus overload due to low priority diagnostics information the devices can have a separate port for connecting them to an ordinary Ethernet network where they expose themselves, for example as web servers.

IEC 62264-1 [4] classifies the tasks for production in levels, see Figure 1.1. The communication on level 2 and below is often integrated and data can quickly be retrieved at any time. Communication with the upper levels on the other hand is often manual and thus data is not readily available. To integrate all levels is known as vertical integration and is yet another trend in automation.

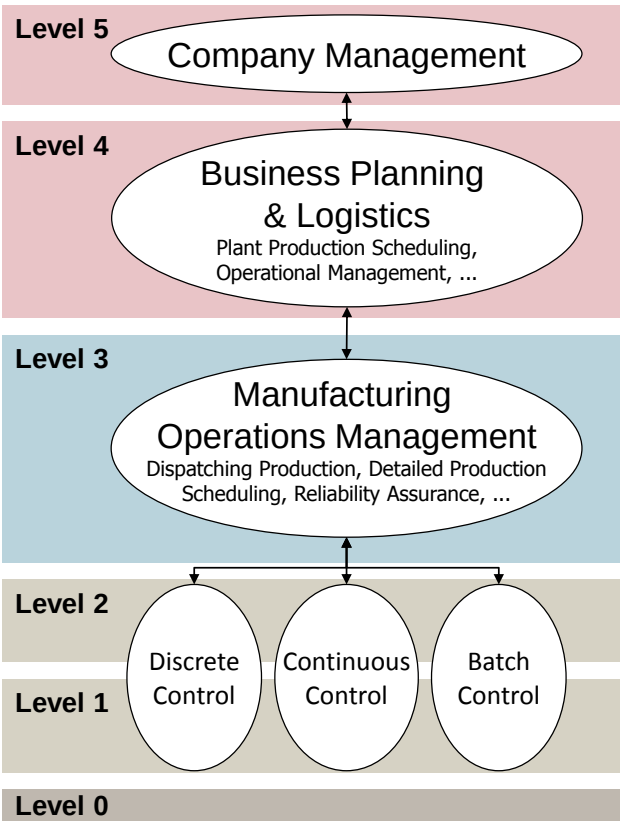


Figure 1.1 Functional hierarchy of production according to IEC 62264-1.

At the same time there is an increasing demand for reduced time to market, that is, to set up and make changes to the production faster. Also environmental impact legislation introduce new boundaries for what is allowed.

Altogether this makes automation a challenging and exciting field.

1.2 Methodology

In this thesis both process automation and robotics have been considered. Process automation includes for example chemical plants and paper mills and robotics includes for example assembly and packaging. The starting point has been current challenges in industrial automation. General approaches have been identified for dealing with these challenges. The following are elaborated in this thesis:

- modern compiler techniques for a graphical automation language.
- realtime execution of the flexible automation language Grafchart.
- Service Oriented Architecture (SOA) for automation.

JGrafchart, an implementation of Grafchart, has been used as an evaluation platform.

1.3 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: *Proceedings of the 14th IFAC Symposium on Information Control Problems in Manufacturing (INCOM'12)*. Ed. by T. Borangiu, A. Dolgui, I. Dumitrache, and F. G. Filip. 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: *Service Orientation in Holonic and Multi Agent Manufacturing and Robotics*. Ed. by T. Borangiu, A. Thomas, and D. Trentesaux. 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.

- T. Gerber, A. Theorin, and C. Johnsson. “Towards a seamless integration between process modeling descriptions at business and production levels - work in progress”. In: *Proceedings of the 14th IFAC Symposium on Information Control Problems in Manufacturing (INCOM’12)*. Ed. by T. Borangiu, A. Dolgui, I. Dumitrache, and F. G. Filip. 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.

1.4 Research Projects

The author is a member of the LCCC Linnaeus Center and the eLLIIT Excellence Center at Lund University. Financial support from the Swedish Research Council (VR) through the LCCC Linnaeus grant is gratefully acknowledged. The LISA project funded by VINNOVA and the research platform SmartFactory^{KL} [5] and are also acknowledged.

1.5 Thesis Outline

In Chapter 2 automation is compared to consumer applications and embedded systems and graphical programming languages are described and compared to conventional programming. Chapter 3 describes the Grafchart language and implementations in detail. In Chapter 4 the challenges for automation are elaborated. In Chapter 5 modern compiler techniques are evaluated for JGrafchart. Chapter 6 describes work toward enabling realtime execution for JGrafchart. In Chapter 7 flexibility is addressed in a broader sense by bringing SOA to automation. Finally, Chapter 8 contains a summary of the work and Chapter 9 lists related future work.

2

Graphical Programming Languages

2.1 Automation 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 [6] for Programmable Logic Controllers (PLC). 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 2.1. For example the applications are executed on the same or similar computer hardware and hardware architecture. Some characteristics are shared with embedded systems. An embedded system is closer to the hardware and is often required to execute its applications in realtime, for example to respond to an event within a given time to behave correctly.

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 producing 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 [7]. There is a demand for spare parts during the entire lifetime of producing 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 [8].

For consumer applications the underlying hardware does not have to be considered, abstraction layers between the application and the hardware take care about

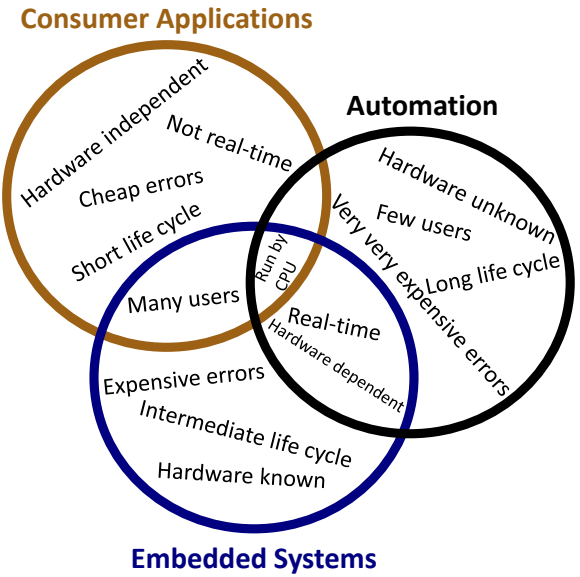


Figure 2.1 Comparison of implementation aspects for ordinary consumer applications, embedded systems, and automation.

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 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. This typically involves many sensors and actuators as well as a considerable amount of hardware configuration.

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 automation, so does the production. Many producing machines take long to start up and they may consume raw material without producing sellable products during startup. Also, many producing machines are dangerous and an error in the program can injure the workers.

Since there is a considerable cost to start up a producing 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 of the biggest reasons why the languages

used for automation differ from other fields. The debugging mode that is only used during development in other fields is practically the only mode in automation. You cannot set breakpoints and step through the code but you need to be able to see everything while it is running to be able to troubleshoot odd behavior without shutting down the machine.

2.2 Graphical Programming Languages in PLCs

The IEC 61131-3 standard defines five programming languages for PLCs [6]: 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 very convenient way to implement applications. Specifically, it suits automation very well.

LD is a replacement for implementing relay logic with physical relays. Engineers thinking in terms of relay logic draw their applications as LD diagrams, see Figure 2.2. Instead of then wiring relays the diagrams can be executed directly in a PLC.

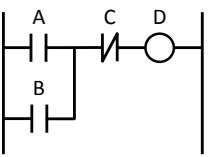


Figure 2.2 An LD diagram equivalent to `D = (A | B) & !C`.

FBD diagrams consist of function blocks whose inputs and outputs are connected graphically, see Figure 2.3. This way it is easier to get an overview of the application.

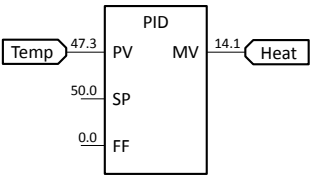


Figure 2.3 An FBD diagram for a PID. The inputs to the `PID` function block are the process value `Temp` (PV), a constant set point of `50.0` (SP), and no feed forward (FF). The manipulated variable (MV) of `PID` is connected to the output `Heat`.

Finally SFC consists of steps representing states, and transitions representing the change of state. It 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. Going the other way, to see how things are connected the code is drawn graphically. A textual language also forces you to name all steps, otherwise you do not have an identifier to use when connecting them to transitions. Working directly with a graphical representation is therefore much more convenient.

Figure 2.4 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 see how things fit together. The textual implementation is more compact but it is much harder to get an overview of how things are connected.

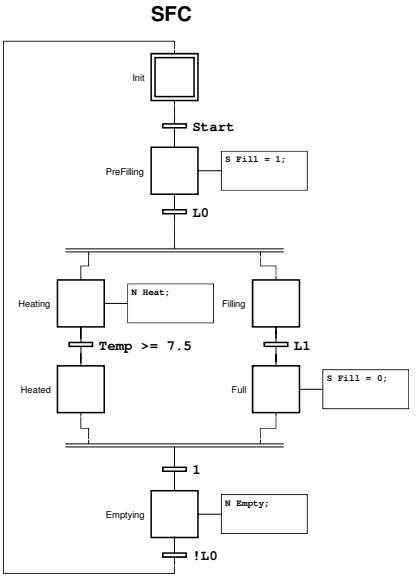


Figure 2.4 An application implemented in both SFC and a minimal textual language.

Visualizing the current state of execution 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

Grafchart

3.1 Grafchart History

Grafchart has been developed by the department of Automatic Control at Lund University since 1991 [9]. It is based on SFC which is well-accepted by the automation community and is extended with inspiration from statecharts, high level Petri nets, and ordinary object oriented programming languages among others. The goal is to make it possible to write large, well structured, flexible, and maintainable applications.

Like SFC, Grafchart uses the state-transition paradigm. Applications written in Grafchart are often referred to as function charts (FC) or step sequences. Grafchart was initially designed to be a very capable and suitable language for batch control [10]. Some of the features added were hierarchical structuring and reusable procedures. It is also extended to facilitate convenient exception handling.

Grafchart has proven to be a very capable and suitable language also for various other automation applications. The state-transition paradigm does not target any specific level in Figure 1.1 and thus Grafchart should work well for implementing sequential applications on any level. It has been used for a wide variety of applications, for example batch control, discrete control, and diagnosis and the paradigm fits them all very well. Grafchart also has potential for formal description, validation, and analysis [10].

There are two implementations of Grafchart, see Figure 3.1. The first one is also called Grafchart and was implemented in Gensym's expert system G2 [11]. Here this implementation is referred to as G2Grafchart. It was hard to continue development of G2Grafchart and it is desirable to rather have a freely available tool built on an open platform. Hence G2Grafchart is no longer used. The second implementation is written in Java and is called JGrafchart. It is actively developed and publicly available for free [9].

JGrafchart is a research tool used in for example the EU/GROWTH project CHEM for control in process industry [12], the EU FP7 project ROSETTA for robotic assembly [13], and a master's thesis for modeling of avionics systems [14].

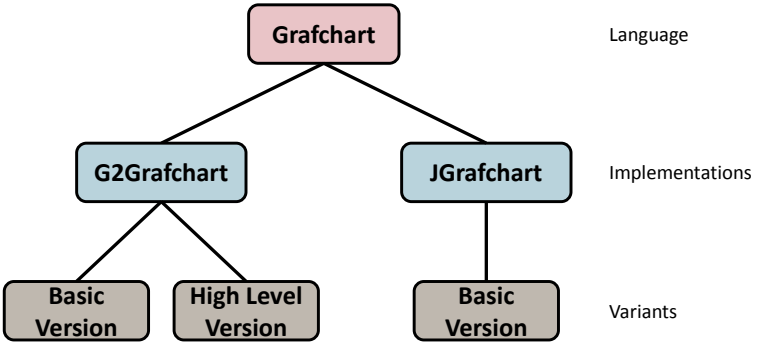


Figure 3.1 There are two implementation of Grafchart, namely G2Grafchart and JGrafchart. There are two variants of G2Grafchart and JGrafchart corresponds to the Basic Version variant.

JGrafchart is also used in education, for example in laboratory exercises on sequential and batch control.

3.2 Grafchart Syntax

Grafchart will be introduced by examples to highlight the main ideas before going into detail.

Core Features

The two main building blocks in Grafchart are steps and transitions. Steps represent possible application states and transitions represent the change of application state. Associated with the steps are actions specifying what to do. Associated with the transitions are Boolean guard conditions.

A piece of a running Grafchart application is shown in Figure 3.2. Two steps are connected by a transition and there are two variables `var` and `cond`. 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 3.3 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 will 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 added to split the execution. A Parallel Join is used to merge the execution again when the parallel paths are completed. In JGrafchart Parallel Split and Parallel Join elements only have two connectors. Figure 3.4 shows how to get more than two parallel paths by combining several Parallel Splits and Parallel Joins.

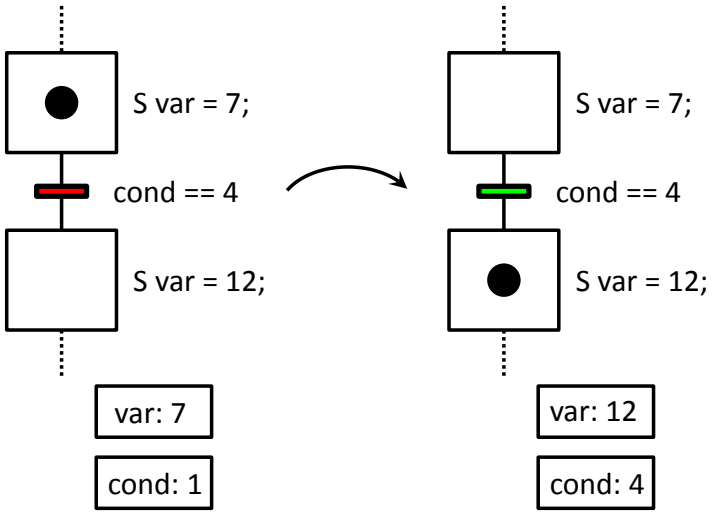


Figure 3.2 A piece of a running Grafchart application showing the main building blocks of Grafchart, steps and transitions. The left part of the figure shows one application state and the right part shows a later application state.

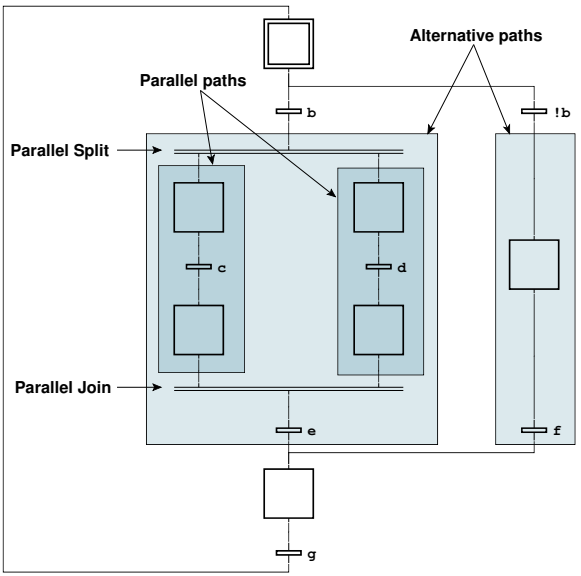


Figure 3.3 A Grafchart application showing how to express alternative and parallel paths.

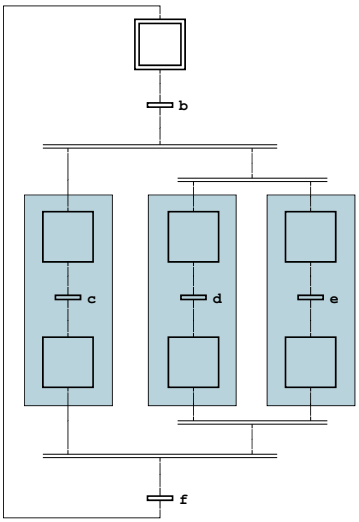


Figure 3.4 A JGrafchart application with more than two parallel paths.

Structuring

A common nonstandard extension to SFC is the Macro Step which contains an internal FC, see Figure 3.5. It is included in Grafchart and makes it possible to split up larger applications into understandable chunks. When a Macro Step is activated its Enter Step is activated. The execution of a Macro Step is finished when its internal state reaches the Exit Step.

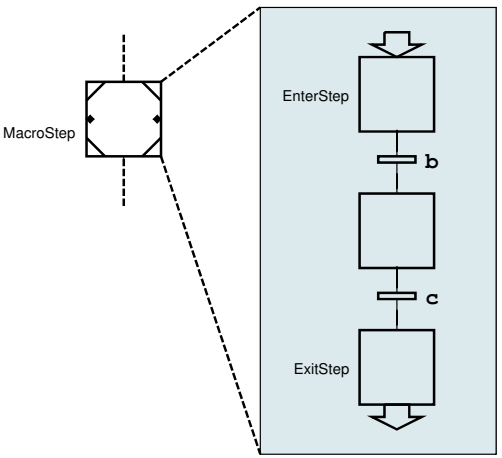


Figure 3.5 A Grafchart Macro Step and its internal FC.

Often the same code is needed in more than one place. In Grafchart it is possible to create reusable Procedures 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 3.6. A Procedure Step is finished when 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 finish before proceeding. Spawned procedure calls terminate when the Exit Step is reached.

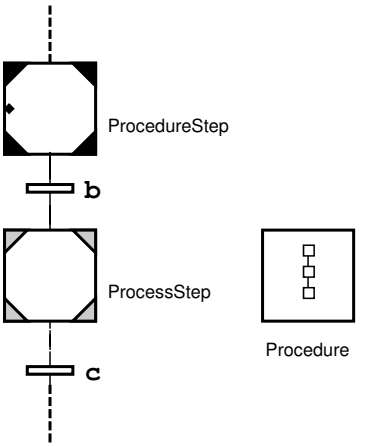


Figure 3.6 A Procedure can be thought of as a reusable Macro Step which can be called from Procedure Steps and Process Steps. Each Procedure Step and Process Step specify which Procedure to call when activated. When `ProcedureStep` is active it will proceed when the procedure call has finished and `b` is true. When `ProcessStep` is active it will proceed as soon as `c` becomes true. The spawned procedure call is unaffected by this.

Yet another way to do structuring in Grafchart is with the 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 larger applications.

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 [15].

Figure 3.7 shows use of the Exception Transition. If `faultA` or `faultB` becomes true while the Macro Step is active the Macro Step will be aborted.

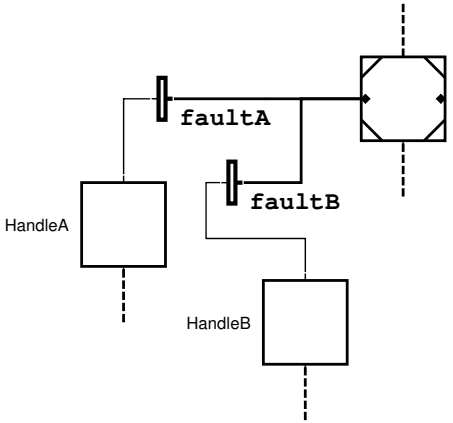


Figure 3.7 Exception Transitions can be connected to Macro Steps and Procedure Steps and are conceptually connected to all steps in the internal FC.

Function Chart Element Summary

Figure 3.8 shows the main FC elements of Grafchart. The Initial Step is activated when the application starts and thus defines the initial state of the application. It is otherwise the same as the ordinary Step. The Step Fusion Set is another construct for exception handling and is used to give a step multiple views, that is, to have one conceptual step appear as several steps. Connection Posts is a way to connect elements without the whole graphical link visible, which can be used to make the FC clearer.

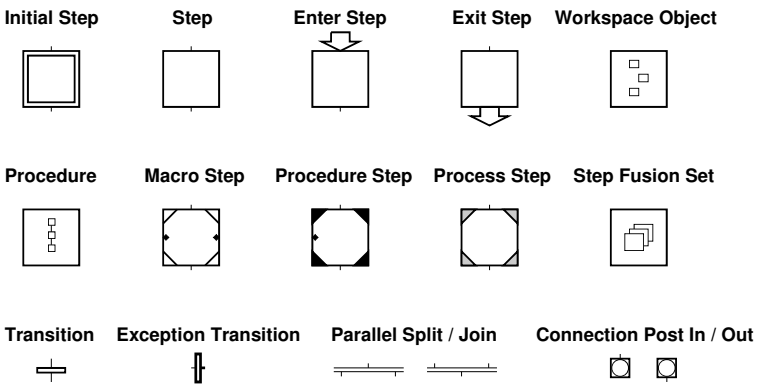


Figure 3.8 The main FC elements of Grafchart.

Variables and Scoping

In Figure 3.2 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 3.9.

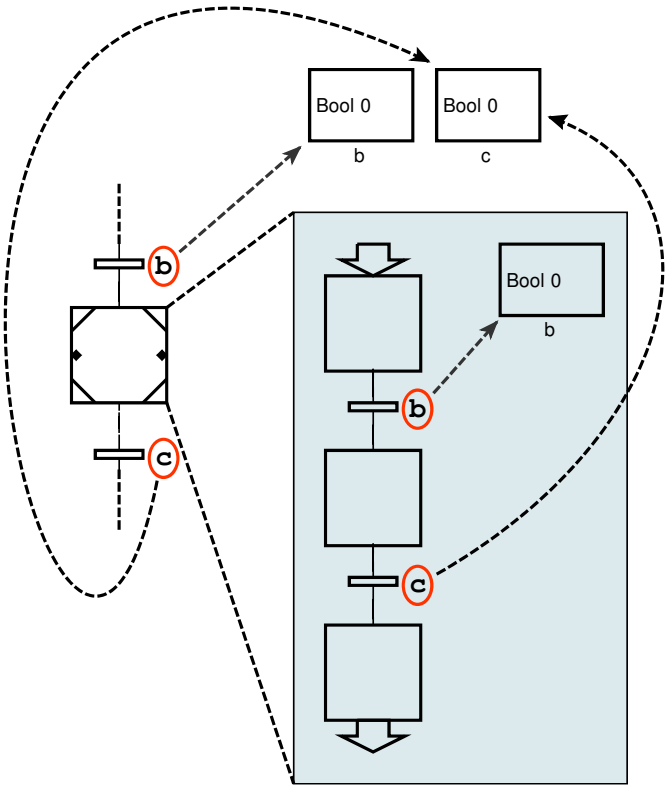


Figure 3.9 Scoping in Grafchart. The arrows indicate the variables used.

Languages

Grafchart consists of three parts which may be considered separate 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 3.2 the S prefix was introduced. Table 3.1 shows the complete list of prefixes.

Table 3.1 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)

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.

When explaining the execution model of Grafchart 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 condition is true. *Firing* a transition involves deactivating the immediately preceding steps and activating the immediately succeeding steps. In the left part of Figure 3.2 the transition is *enabled* since its only immediately preceding step is active, but it cannot *fire* since its condition is false. In the right part the transition's condition is still true but it cannot *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 step is active.

Steps also have some additional properties, namely `x`, `t`, and `s`. `x` is true if the step is active and false if the step is inactive. `t` is how many scan cycles 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 condition is true.". The behavior of an application might then depend on in which order the transitions are evaluated as shown in Figure 3.10.

Avoiding this kind of behavior will guarantee that activated steps are active during at least one scan cycle, which makes it easier to reason about an application. For

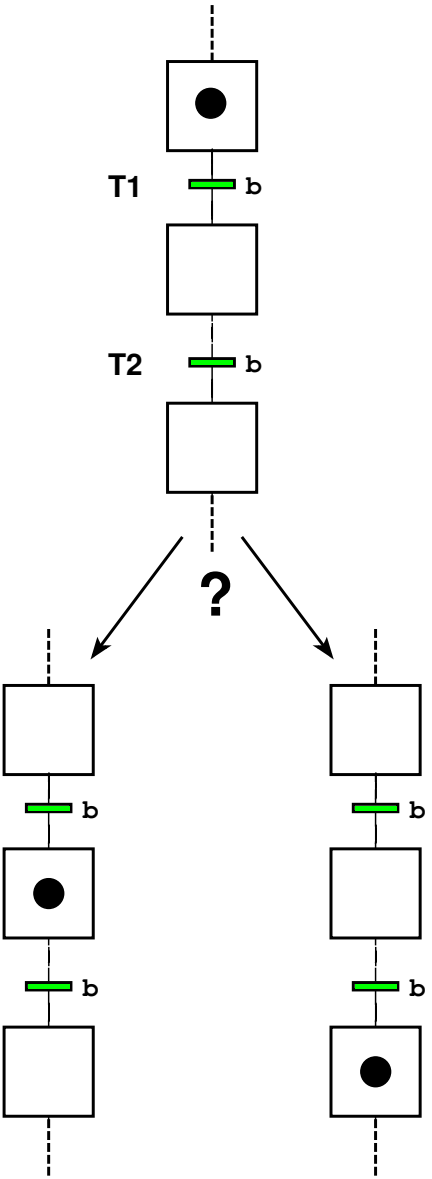


Figure 3.10 With a naive execution model, what happens when `b` becomes true depends on the iteration order of the transitions. If `T2` is checked before `T1` the scan cycle will end in the state to the left. If `T1` is checked first it will fire and when `T2` is then checked it will also fire meaning that the scan cycle will end in the state to the right.

example for an N action the variable will always be true at least one scan cycle and the same goes for the `x` step property.

Another issue is transitions for alternative paths which have conditions that may be true at the same time. This is called a conflict and the transitions involved in a conflict are called conflicting transitions. Conflicts are not allowed in Grafchart. Note that there cannot be conflicts between Exception Transitions and ordinary Transitions since Exception Transitions have priority over ordinary Transitions.

The execution model for Grafchart does the following during each scan cycle:

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

The execution model is compatible with Grafcet [10], the language which SFC is based on, and gives sufficiently deterministic behavior. The remaining nondeterminism 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.

3.3 JGrafchart Specifics

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

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. Integer values are written like ordinary whole numbers, Real values are written using decimal notation, String values are written as the string value enclosed by quotation marks, and Boolean values are written `1` for true and `0` for false.

JGrafchart uses loose typing and automatic type casting. The target data type is determined by the context. For example in Figure 3.11 `b` is cast to a Boolean since this is the data type for transition guard conditions.

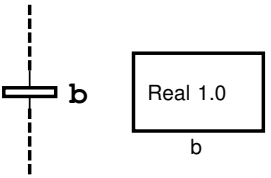


Figure 3.11 The target data type for a transition condition is Boolean. Therefore the value of the Real variable `b` will automatically be cast to a Boolean. In this case it is the value `1.0` that is cast and the condition will therefore be true. A complete list of how values are reinterpreted by a cast is included in the JGrafchart documentation.

Variables and IO

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 (IO) are needed.

One IO possibility in JGrafchart is the IO elements Digital In, Digital Out, Analog In, and Analog Out as well as inverted variants for the Digital In/Out. At the beginning of each scan cycle each In IO is read from the external environment. An Out IO is written to the external environment whenever assigned. How the IO interact with the external environment depends on the chosen IO implementation. A custom IO 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, it is limited to Boolean and Real values.

Another IO possibility is the Socket IO elements. JGrafchart can connect to a TCP server and communicate Boolean, Real, Integer, and String values over a socket using the message protocol: `<identifier> '|' <value> '\n'`. The TCP server is then responsible for interacting with the external environment.

Dynamic References

In most programming languages there are dynamic references which can be used to create more general 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. 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 3.12. The result of a dereference operation can be any element with a name and it is used like when the element is named explicitly.

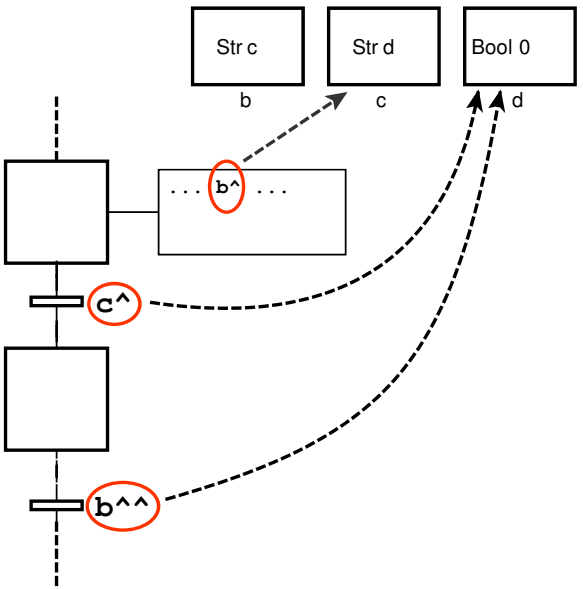


Figure 3.12 The String Variable `b` has the value `c`. As a dynamic reference it thus refers to the String Variable `c`. Similarly the String Variable `c` refers to the Boolean Variable `d`. The expression `b^` will dereference the String Variable `b` and return the String Variable `c`. Similarly `c^` 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 what Grafchart is all about. However, for simple conditional expression evaluations this is much overhead and 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 3.13 shows a small FC written without and with *inline if*. The implementations behave slightly different: The left part executes the initialization in one scan cycle and the rest in the next scan cycle. The right part executes everything in the same scan cycle.

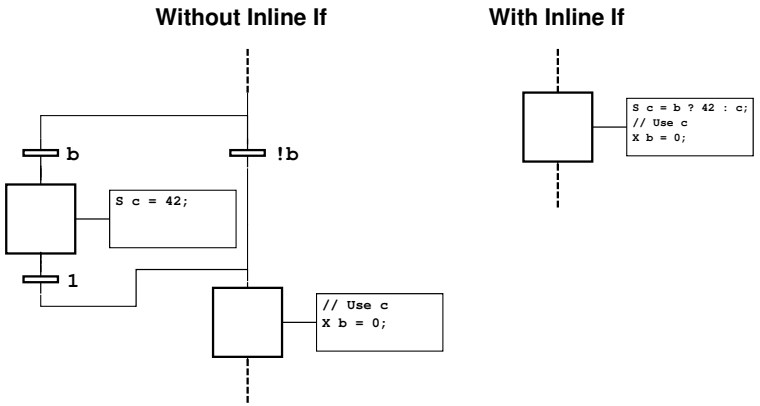


Figure 3.13 The variable `c` needs to be initialized to 42 if `b` is true. The left part shows how to implement this without *inline if*. The right part shows how it can be implemented with *inline if*.

Graphical Elements

The online view of the running application is great for developers and maintenance staff. For the operators of the producing machine on the other hand it is more intuitive with an interactive interface resembling the machine. In JGrafchart it is possible to create interactive operator interfaces with graphical elements. Figure 3.14 shows a JGrafchart operator interface.

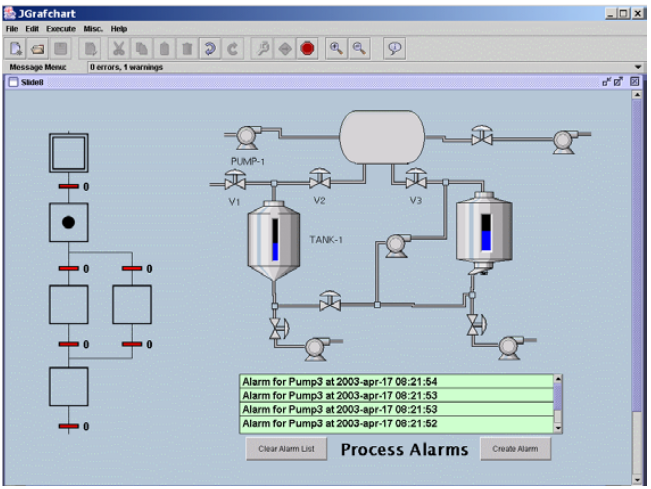


Figure 3.14 An operator interface with process alarms, the executed application, and animated tanks and piping resembling the controlled process.

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

Methods and Functions

JGrafchart provides a library of built-in functions, for example mathematical functions like `sqrt`, `sin`, and `abs`. Elements also have methods, for example getting and setting the location and size of graphical elements can be used to create animations. For steps the `x`, `t`, and `s` properties can also be accessed.

Conflicting Transitions

A common mistake is conflicting transitions. Unfortunately it is also hard to analyze which transitions may be conflicting in an application. In JGrafchart no such analysis is implemented. The user is responsible to ensure that there are no conflicting transitions. If there are anyway, all conflicting transitions will fire.

3.4 Comparative Example

As mentioned in Chapter 3.2 Macro Steps 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 will be even more beneficial.

The bead sequencing process in Figure 3.15 can be controlled by the SFC application in Figure 3.16. In Figure 3.17 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 more linear. The overall control logic is then much clearer. The internal FC of the Macro Step `ReleaseBeads` is shown in Figure 3.18. The other two Macro Steps contain the black/yellow alternative paths in the SFC application and are omitted. Note that the error 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 it.

In Figure 3.19 and Figure 3.20 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 is possible to remove the redundant exceptional case handling inside the Macro Step. By using dynamic references (`^`) and inline if (`?:`) several alternative paths were removed. The resulting application has a linear flow with a minimal number of loops. The state flow is thus as easy as possible.

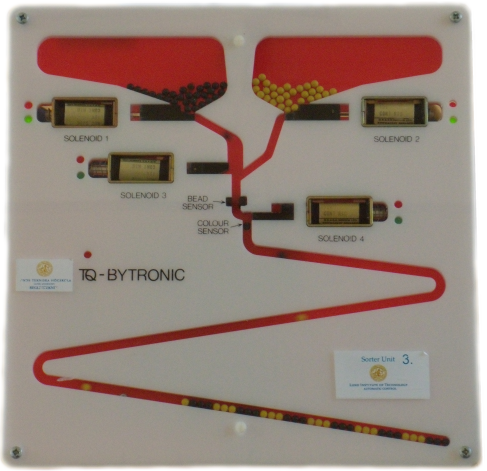


Figure 3.15 The bead sequencing and sorting process. Yellow and black beads are sequenced into a pattern, here 3 black 3 yellow. Turned upside down the process becomes a bead sorter, sorting the beads back into their respective compartments.

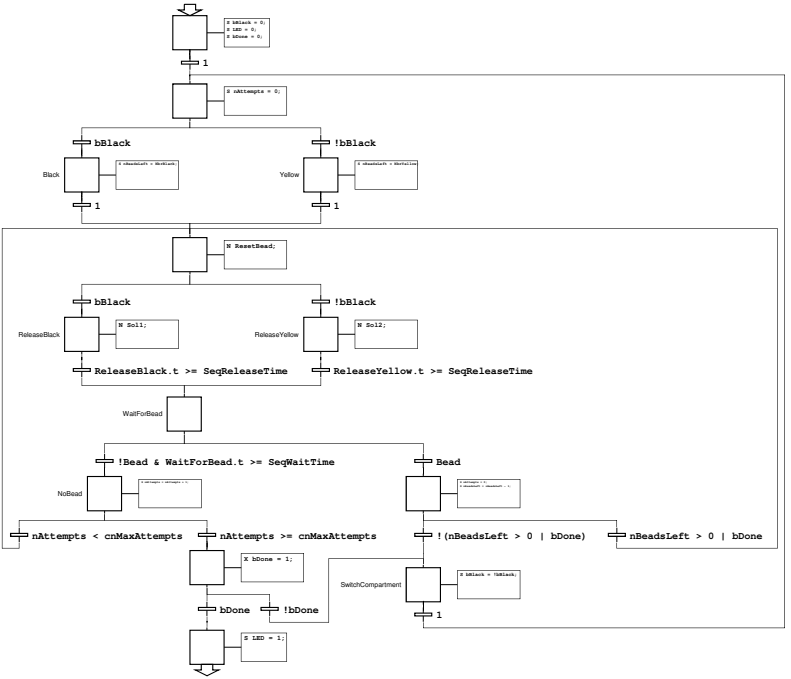


Figure 3.16 An SFC application for controlling the bead sequencing process.

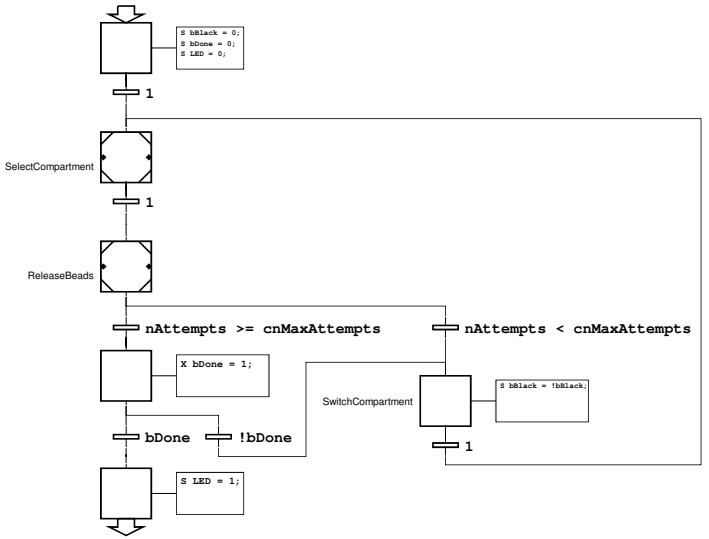


Figure 3.17 The application in Figure 3.16 structured with Macro Steps. The internal FC of `ReleaseBeads` is shown in Figure 3.18.

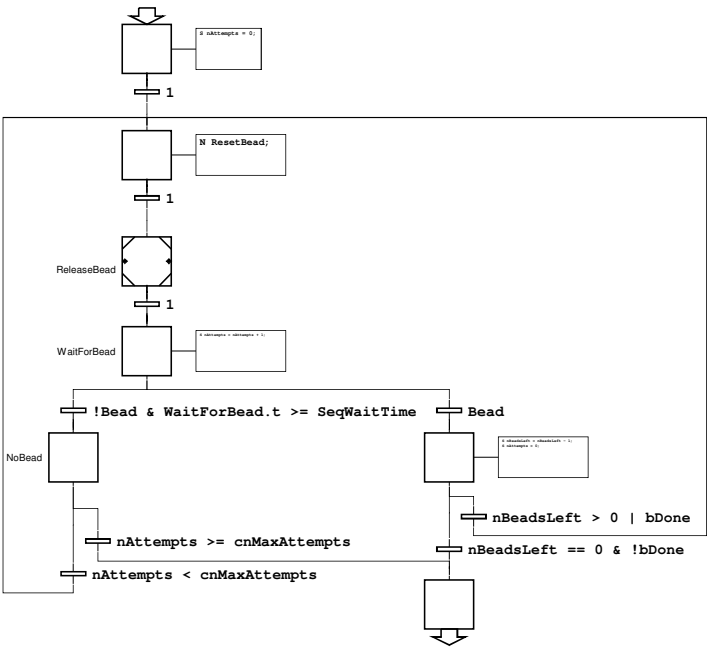


Figure 3.18 The internal FC of the Macro Step `ReleaseBeads` in Figure 3.17.

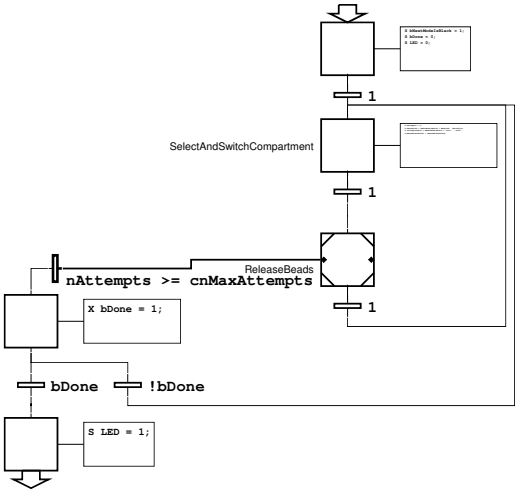


Figure 3.19 The application in Figure 3.17 written with additional features of Grafchart and JGrafchart. The internal FC of `ReleaseBeads` is shown in Figure 3.20.

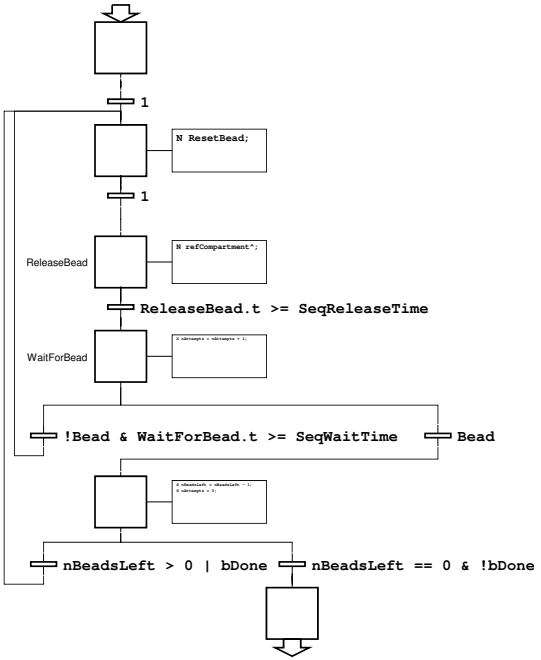


Figure 3.20 The internal FC of the Macro Step `ReleaseBeads` in Figure 3.19.

3.5 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 [16]). They may also contain internal FCs, a feature called multi-dimensional charts.

Objects in G2Grafchart may also have parameters and attributes. The following notations are used to access token and object data in actions and conditions: `inv` for token data, `sup` for object parameters, and `self` for object attributes. For more details about G2Grafchart, see [10].

4

Challenges

The current trends in industrial automation mentioned in the introduction are the need for customizable production, flexibility, vertical integration, convenient handling of more advanced sensors and actuators, and reduced time to market. These trends impose increased requirements on the automation software, including the development tools and languages.

As shown in Chapter 3.4 Grafchart's hierarchical structuring and exception handling mean that it is possible to create better structured and more conceivable applications. Together with Grafchart's reusability features this leads to better customizability and flexibility. This has been proven to work very well for batch control on level 2 in Figure 1.1 [10, 15]. The current Grafchart implementation, JGrafchart, works really well for this purpose. However, it would be even better with additional concepts such as HLV. It would also benefit from a more flexible language implementation where concepts are easier to add and evaluate.

As Grafchart is an extension to SFC it should be possible to execute the applications in the same context. SFC is typically executed in realtime in a PLC. JGrafchart on the other hand runs on an ordinary PC in an interpreted manner. The same Java object instances are used both for execution and application visualization. This is far from realtime. For core robotics applications, a scan cycle time of a few milliseconds is often required. Forces build up quickly and being delayed one scan cycle can be the difference between smooth behavior and destroying something. With the current version of JGrafchart (2.2.0) not even tiny applications can be executed reliably at this rate.

To have flexible and reliable tools and languages for developing applications which can execute in realtime is not enough, there is also a demand for a more flexible architecture. Currently, interaction with similar field devices can be fundamentally different and thus they are not interchangeable. Custom encapsulations can be made to decrease the dependency toward a particular field device, but it is desired to have an architecture which provides a unified way to do this.

In Chapter 5 modern compiler techniques are evaluated for JGrafchart, building the foundation for implementing extensions such as HLV. In Chapter 6 the first steps toward execution in realtime are made. In these chapters the programming

environment for automation applications are considered. Chapter 7 tackles the more general challenge to build the automation architecture for the whole factory in a way that provides flexibility, vertical integration, and convenient handling of advanced sensors and actuators. This is done by evaluating SOA for automation.

5

Compiler Techniques

5.1 Introduction

As mentioned in Chapter 3.5 there are two variants of G2Grafchart, namely the Basic Version (BV) and the High Level Version (HLV). JGrafchart corresponds to BV and extending it with HLV would enable new powerful and flexible ways of writing applications. It is also desirable to retain a pure BV and have both variants available. Since HLV is a superset of BV it is desirable to reuse the BV implementation as base for the 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 chapter introduces modern compiler techniques to an automation language with the goal to make it extensible. Thus the challenge in Chapter 4 to create a more flexible language implementation is addressed. To use modern compiler techniques for JGrafchart is also a natural first step toward realtime execution. This chapter concerns the compilers for the textual languages of Grafchart and is based on publication [17].

5.2 Background

Programming Languages

A programming language can be described by its *syntax*, *semantics*, and *pragmatics* [18]. The *syntax* describes the allowed structure of the language, the *semantics* describe the meaning of the syntactic elements, and the *pragmatics* describe 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 `S temp = 12 * y;`. The *semantics* describes that an assignment means evaluating an expression and assigning 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 for setting up a temporary variable.

Compilers

The task of compilers is to transform written applications into something executable. A typical compiler works in a sequence of phases, see Figure 5.1. 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.

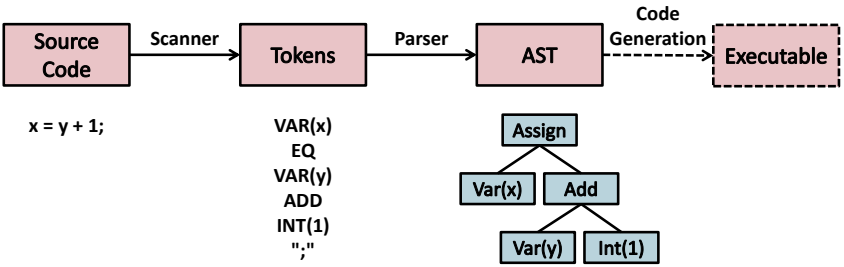


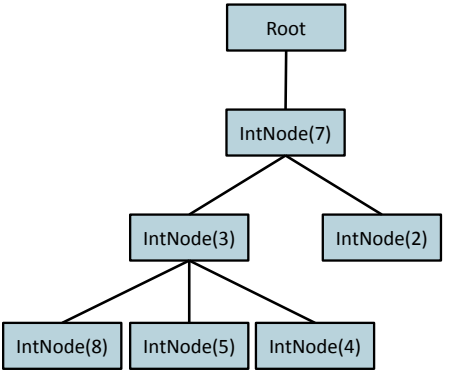
Figure 5.1 The overview of a compiler. The *scanner* transforms the source code `x = y + 1` into a sequence of classified tokens. For example `x` is classified as the token *variable x* and `1` is classified as the token *integer 1*. The *parser* uses these tokens to build an AST representation of the application.

The *scanner* and *parser* check that the application is syntactically correct, a prerequisite for building an AST. An AST must also be analyzed to check if the application is semantically correct. In Figure 5.1 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 [19]. 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 biggest advantage with 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 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*.

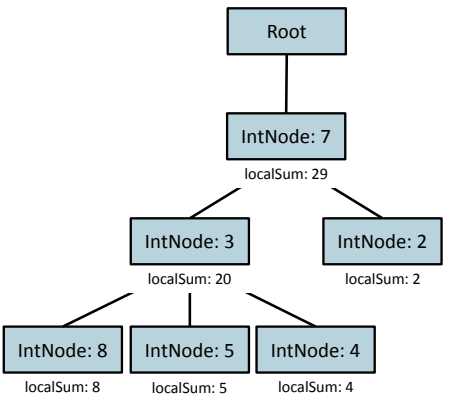
Consider the toy AST with integer nodes in Figure 5.2. It consists of a `Root` node with one `IntNode` child node and each `IntNode` has an integer value and zero or more `IntNode` child nodes.




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

Figure 5.2 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 5.3 a *synthesized attribute* 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 5.3 The AST from Figure 5.2 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 5.4 an *inherited attribute* has been added for the sum of the whole tree.

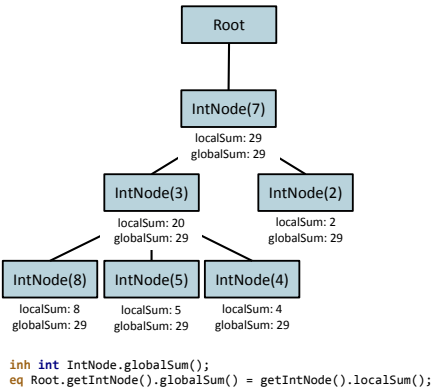


Figure 5.4 The AST from Figure 5.3 attributed with the global sum. This is done by adding an *inherited attribute* called `globalSum` for each `IntNode`. `Root` specifies that for all other nodes this attribute is the local sum of its child node.

Parametrized attributes are attributes which may depend on parameters. In Figure 5.5 a *parametrized attribute* has been added, which checks if the local sum is greater than the supplied argument.

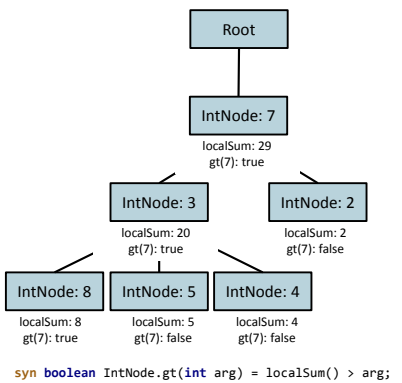


Figure 5.5 The AST from Figure 5.3 attributed with a *parametrized attribute* for each `IntNode`, which checks if the local sum is greater than the argument, here 7.

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. In Figure 5.6 a *reference attribute* to the root node has been added for all nodes.

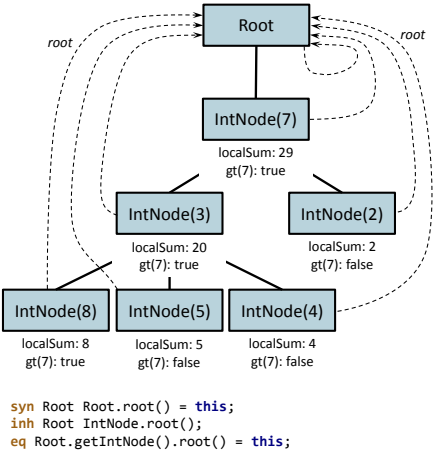


Figure 5.6 The AST from Figure 5.5 attributed with a *reference attribute* to the root node for all nodes. The `Root` node specifies itself as its own `root` with a *synthesized attribute* and as all other nodes' `root` with an *inherited attribute*.

A *collection attributes* is a node local collections to which any node in the tree can contribute. In Figure 5.7 the *collection attribute* `greatSums` which contains all local sums greater than 7 has been added to the `Root` node.

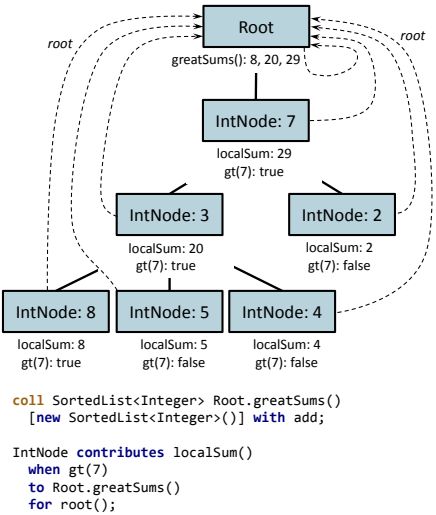


Figure 5.7 The AST from Figure 5.6 attributed with a *collection attribute* which contains all local sums greater than 7. It is defined for all `Root` nodes and the `IntNodes` contribute their local sums to the collection of their `root`.

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 [20]. 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 [21] and Java 7 [22] extensions, the Control Module Language with object oriented extensions [23], and the Optimica extension to Modelica [24]. It was thus an attractive candidate for making the JGrafchart compilers extensible. Other alternatives considered include Eli, Synthesizer Generator, Silver, and Kiama.

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.

5.3 JGrafchart Compilers

As mentioned in Chapter 3.2 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, only conceptual as it is implicitly defined by the editor, and ASTs built by the action and condition compilers. The FC AST is attached to the root node. 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 5.8 implements a controller for a batch tank that is filled until `full`, then emptied until `empty`. This sequence is repeated, 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 5.9.

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 [25]. 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 this

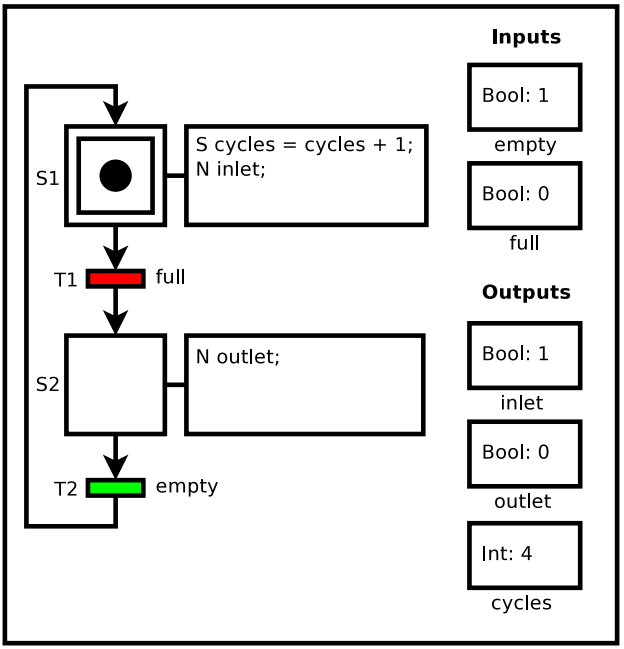


Figure 5.8 A control application for a batch tank that is filled until `full` and then emptied until `empty`.

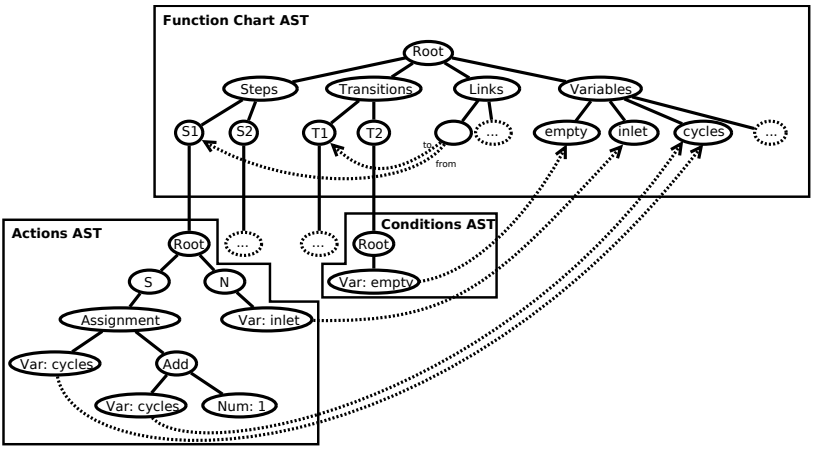


Figure 5.9 The AST for the application in Figure 5.8. The steps, transitions, and variables are transformed into an FC AST by the FC compiler. For each step the actions are transformed into an action AST by the action compiler. The condition of each transition is transformed into a condition AST by the condition compiler.

knowledge to work on these compilers. However, it has the 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.

5.4 Rewriting the Compilers

The following strategy for rewriting the compilers was used, see Figure 5.10:

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

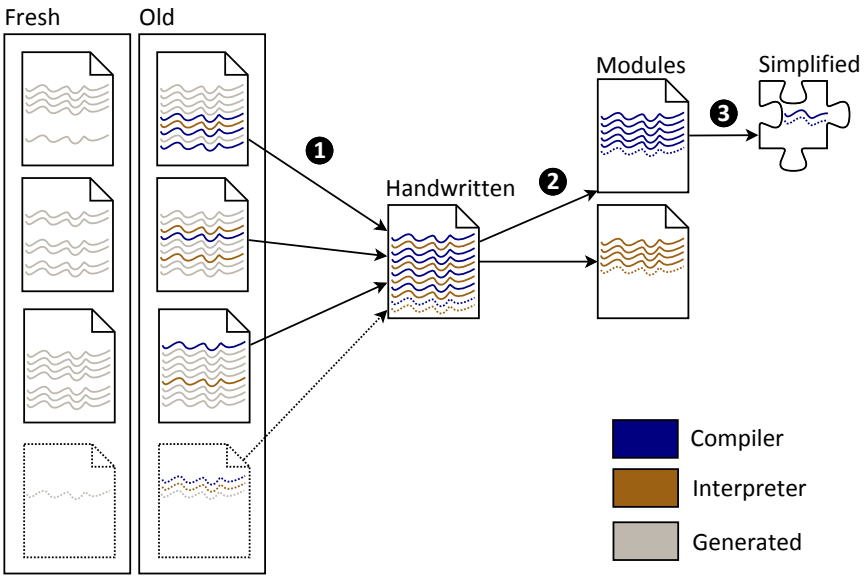


Figure 5.10 How the JGrafchart compilers were rewritten. First the handwritten code was extracted, then it was split up into modules, and finally the compiler module was simplified.

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

Separating 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 large 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 5.2).

Step 2: Split Into Modules

The modules chosen for both language implementations were Compiler, Interpreter, and Utilities. The Compiler module handles the compilation of the AST, the Interpreter 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. In the old implementation they were implemented separately for the action 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 used during both compilation and execution by both language implementations.

Step 3: Simplification

The Compiler modules were transformed piece by piece to ReRAG equations. In parallel, JUnit tests [26] were added to verify that nothing was broken.

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 upwards 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 implementation does is receiving the compilation context and checking the collection attribute, see Figure 5.11.

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

```
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 5.11 The new compile interface implementation.

5.5 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 would also be useful to override variables and procedures. Currently this is not possible in JGrafchart since there is no way to bypass the local context during lookup. The `sup` notation is proposed to bypass the local context, see Figure 5.12. Adding `sup` as an extension to the new implementation was straightforward. This shows that the new implementation is indeed extensible.

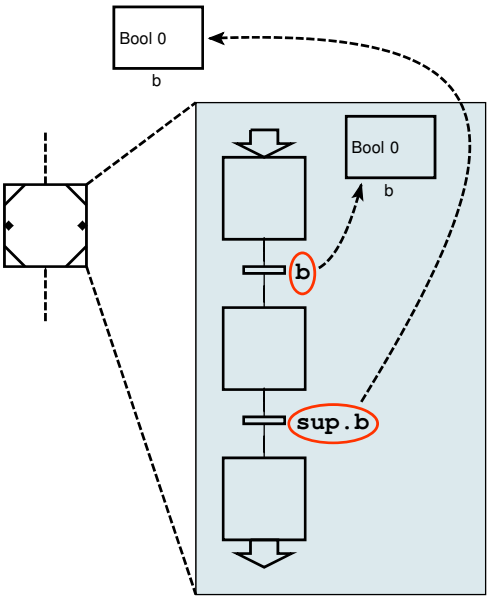


Figure 5.12 Variable bindings with and without `sup`. Without `sup` the local variable `b` is found. With `sup` the local context is skipped and the non-local variable `b` is found.

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 compact as Old. In Total_{excl} Built-ins are excluded. In Total_{fair} Separate Generated has also been excluded.

	Old	New	New_{eq}
Compiler	1537	380	209
Interpreter	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 compact 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 only containing 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 Compiler modules are 75% smaller than the old ones. Also, the new

implementation is not as compact as the old one. Comparing the old implementation to the equally compact New_{eq}, the new implementation is 86% smaller. In addition, the new Compiler modules have been enhanced with several new compiler checks and additional attribution to make the interpretation easier.

The new Interpreter modules are smaller, mainly because duplicated 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 3.12.

Previously, built-in functions and methods were implemented in both the action 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 also 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 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 compact and separate generated files and built-ins are excluded. The the new implementation is then 75% smaller than the old one.

Implementing the `sup` extension only required 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 is fairly large and 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.

5.6 Conclusions

To extend JGrafchart with HLV would enable new powerful and flexible ways of writing 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:

- Extensibility
- Improved maintainability
- Modularized functionalities
- Less code
- Increased robustness
- Degraded compiler performance
- Developers must understand attribute grammars

Improved maintainability and increased robustness lead to less bugs and better quality. The degraded performance is acceptable and not noticeable for the user. On the other hand less bugs increase the tool's reliability and added compilation checks and improved features are also of great value. Improved compiler 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.

6

Realtime Execution - Work in Progress

6.1 Introduction

Currently 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, in some industrial contexts, may be considered showstoppers:

- The execution has to be performed on hardware supporting graphics 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 executing the application.
- Visualization can only be made on one computer.

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

In most IDEs 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 will be separated and the editor will also be used as a visualizer. This chapter describes this work in progress, thus addressing the challenge in Chapter 4 to execute Grafchart applications in realtime.

6.2 Background

JGrafchart has been developed with easy access to all data. It has been easy to create both appropriate and less appropriate dependencies. The focus has been on quickly adding new features. Many shortcuts have been taken, a strategy that pays off well in the short run. However, accumulation of less appropriate dependencies makes it harder and harder to add new features. For new developers it is also harder to understand how things are connected and why. Splitting JGrafchart into smaller self-contained parts enforces removal of several less appropriate dependencies.

Splitting the current implementation into standalone editor, compiler, and execution engine requires deep understanding of all parts of the JGrafchart implementation. This was obtained by fixing bugs, adding small features, and refactoring the code. All existing features were retained but the focus was now instead on creating 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 (Chapter 5) and refactoring the code, the earlier 85,000 lines of code (version 1.5.3.4) were reduced to 45,000 lines of code.

In Chapter 5 a more clean interface between the textual compilers and the editor was created, which was a first step toward a standalone compiler. The textual compilers depend on the FC AST which was still interconnected with the editor. To create a standalone compiler the FC compiler must also be separated. To enable realtime execution, the execution related code must be standalone. This ensures that there are no dependencies on GUI painting operations or other user interactions.

6.3 Standalone 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 large JastAdd module (see Chapter 5.2). After this operation, only the editor worked. The interconnection between the parts had been broken and new explicit interfaces were needed to get the other parts working again.

6.4 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 6.1 shows an overview of the current design. The new IDE contains both the editor, compiler, and an internal execution engine. Hence the previous workflow can still be used. It will also be possible to use an external Execution component consisting 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 use the compiler and the execution engine from a command line.

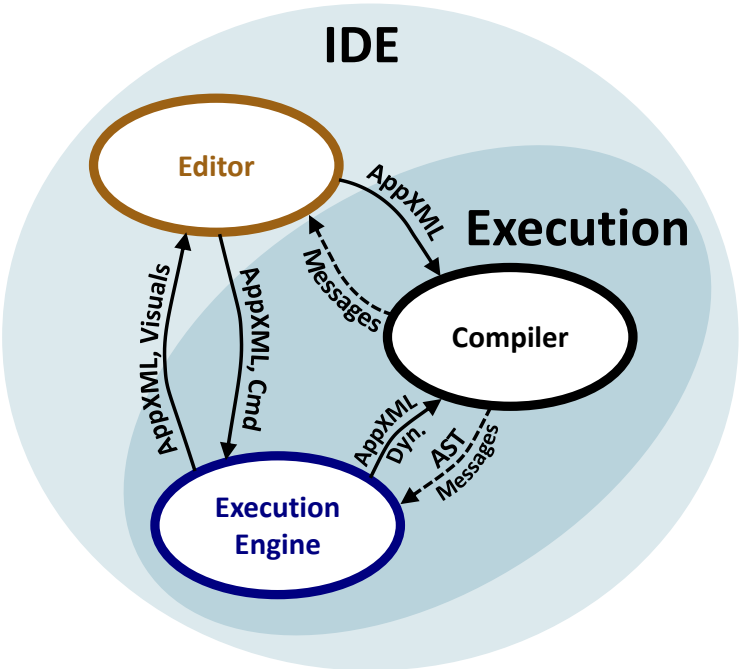


Figure 6.1 This figure shows an overview of the current interface design between the standalone editor, compiler, and execution engine. Most interactions reuse the application XML (AppXML).

The application XML is used in most interfaces and is the same as when applications are saved to file. A big advantage with 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 the dynamic features. Visualization is driven both by 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.

6.5 Standalone 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 had to be adapted to fit the graphics library.

Most compiler code had to be rewritten due to the many dependencies to the editor. The new compiler was written using ReRAGs to make it extensible. JUnit tests [26] 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 interfaces since in 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 operation in the interface. In the new implementation a super class could be used and thus only a single implementation of each operation was required.

A new XML parser was implemented to build the new AST from the application XML. This was straightforward.

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. 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 save format to simplify the compiler implementation and make it more robust. For example the way to save connections between elements was changed. Instead of a sometimes suffixed element identifier, both an element and a port identifier are used. Less reasoning is then needed to find the right ports and there is no longer a risk that a suffixed identifier is identical to another, unsuffixed, identifier.

6.6 Standalone Execution Engine

There is an initial design for how to extend the AST with the interpreter code which is currently being implemented. So far only the basic features are operational.

The new execution engine has been profiled to get an initial estimate of its expected performance. A small application only using constructs currently available in the new implementation is shown in Figure 6.2.

The interpreted execution was profiled by manually instrumenting the execution code and removing the sleep between the scan cycles. The execution time was then

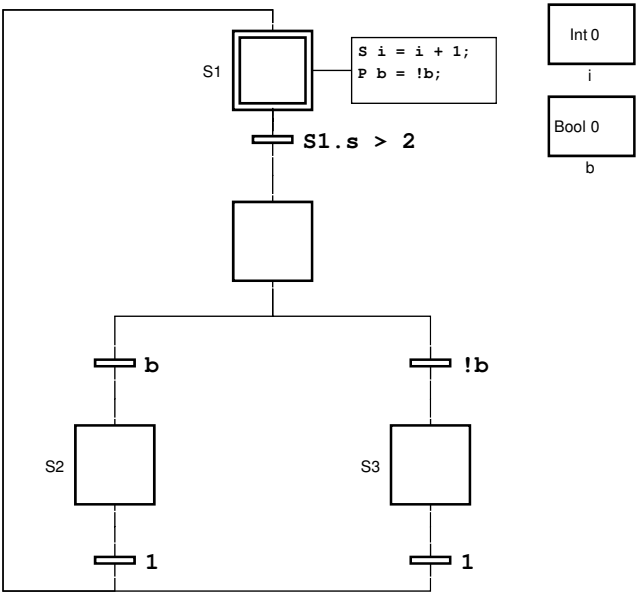


Figure 6.2 The profiled application. Its sole purpose is to exercise the constructs currently available in the new implementation.

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 has only been slightly changed so far. The main difference is that the graphical painting has been removed in the new implementation. Removing the painting in the old implementation also gives an average execution time per scan cycle of 500 ns. The expected performance is thus an order of magnitude faster.

6.7 Editor/Compiler Interaction

In graphical programming languages compilation messages cannot refer to problems by file and line numbers. A nice feature of JGrafchart is that errors 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. Not only retaining but also enhancing this feature was desirable. Using direct references, as done previously, is not possible with a standalone 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 for storing the graphical connections. The rest of the elements were extended with similar unique identifiers.

Figure 6.3 and Figure 6.4 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.

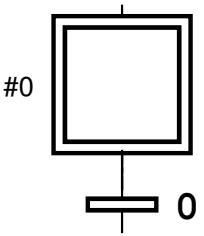


Figure 6.3 Error highlighting with the previous implementation. Warnings are not highlighted at all and the error in the step actions is not highlighted since the actions for the step are hidden.

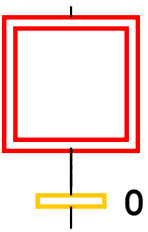


Figure 6.4 Highlighting with the new implementation. The step with an error is colored red and the unconnected transition is colored orange.

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 causing the error and the transition causing the warning are clearly highlighted. 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 shown.

6.8 Evaluation

Reusing the application XML in the interactions works since it contains all information about the application and means that fewer exchange formats are required. A running application also contains visualization data and can receive operator commands. These will be sent separately to limit the communication between visualizer and execution engine.

With all compilers implemented using ReRAGs it is possible to add extensions, for example for generating code that can be executed in realtime. Application refactoring like renaming a JGrafchart variable should also be quite easy to implement using the new AST. Changing running applications can also be addressed when it is possible to edit an application while a previous version of it is executing. With a standalone execution engine, distributed applications can also be implemented more easily.

6.9 Conclusions

Many advantages and opportunities are expected with the new structure:

- Improved execution performance
- Clear interfaces between the parts
- Easier to add new features
- Extensibility
- Improved maintainability
- Increased robustness

The performance measurements indicate that the new interpreter will be 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. With separate parts it is easier to get an overview of each part which improves maintainability and robustness. It will be interesting to evaluate these aspects further as the work progresses.

7

Service Oriented Architecture

7.1 Introduction

As mentioned in Chapter 1, automation needs to be more flexible. A promising approach is Service Oriented Architecture (SOA) which is widely used for business processes [27, 28]. It has also been recognized for use in automation in many research projects [29, 30, 31]. An outcome of the SIRENA project [32] is an open source Devices Profile for Web Services (DPWS) implementation targeted at embedded devices [33, 34]. However, SOA is still barely used for industrial automation. This should be of no surprise as it is a fairly new technology and only mature technologies known to work well in practice are typically considered. Presenting successful realistic examples which provide the anticipated advantages is one step toward convincing the automation community that SOA is worth considering.

The work presented in this chapter enables automation to be built in a service oriented way by making it possible to use JGrafchart as a generic DPWS client. It has been carried out in collaboration with DFKI in Kaiserslautern, Germany and has resulted in publications [35] and [36].

7.2 Background

SOA in Automation Technologies

SOA is a component based software design methodology [37]. A component encapsulates a specific functionality and is called a service. Services are unassociated, loosely coupled, and self contained. They are described with metadata in a way that allows them to be both language and platform independent. To write applications with SOA is known as orchestration and is done by combining many services. This enables a high degree of reusability and flexibility. Using SOA for automation has potential to decrease the engineering effort significantly.

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 [38]. 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. Also, in SOA-AT the timing is often important. In automation a small time delay could be the difference between a stable (safe) and an unstable (dangerous) system.

In SOA-AT the functionality of each field device is encapsulated as a service and the control application is written by orchestrating these services. Control applications can be made hardware independent as similar field devices can expose themselves as identical services. This enables reuse of control applications.

Most SOA tools are tailored for business processes. In [39] several tools were evaluated and DPWS was the tool deemed best suited for SOA-AT. DPWS uses existing WS-* specifications and defines a minimal set of implementation constraints to enable web services on resource constrained devices [40, 41].

DPWS

Plain web services are complicated to use since for each service the desired WS-* 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 7.1. At the root is a DPWS *device*. Below the *device* are *services*, *portTypes*, and *operations* which are the same as for ordinary web services.

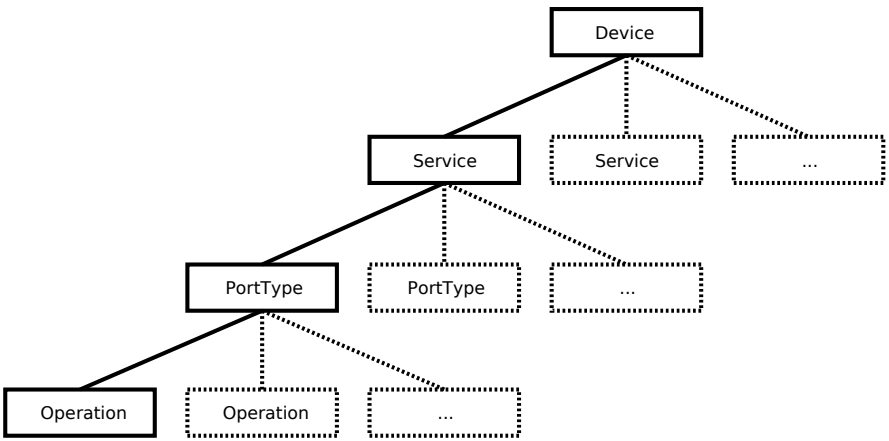


Figure 7.1 The DPWS hierarchy. A DPWS *device* hosts *services*, *services* contain *portTypes*, and *portTypes* contain *operations*.

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`. A *service* encapsulates a specific functionality. In SOA-AT this is the functionality provided by a field device, for example a motor. A *device* hosts *services*, for example it could host a group of co-located motors.

Figure 7.2 shows an overview of the DPWS stack [42]. 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.

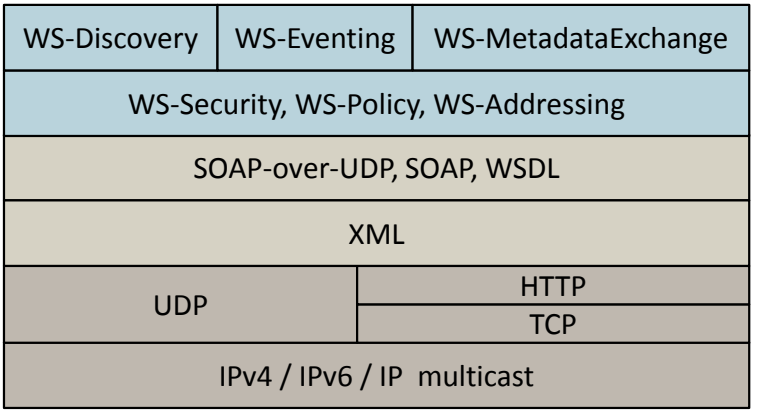


Figure 7.2 Overview of the DPWS stack.

WS-Discovery The WS-Discovery extension makes it possible to dynamically find devices on the local network, see Figure 7.3. 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, sending a *Probe* message should only be necessary when the client joins 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 required devices not found with WS-Discovery the connection must be set up manually.

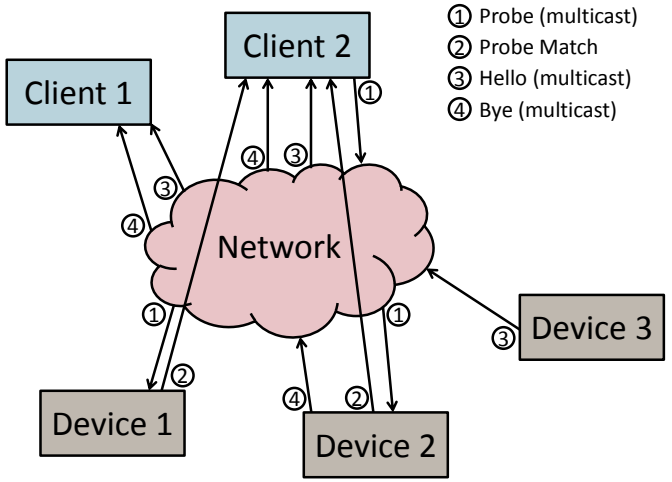


Figure 7.3 Initially Client 1, Device 1, and Device 2 are connected to the network. Client 2 joins and multicasts a *Probe* message. Device 1 and 2 receive this message and each respond with a *Probe Match* message to Client 2. Now Device 3 joins the network and multicasts a *Hello* message which is 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.

WSDL Services are specified using WSDL. Each service is defined by its WSDL description, often 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. Briefly, 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.

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 As devices and clients join and leave the network it is necessary to be able to uniquely identify devices. 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 it is the exact same device.

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

Summary It is possible to create generic orchestration tools by relying on the DPWS mandatory extensions WS-Discovery, WS-Addressing, and WS-MetadataExchange. However, no tool for this 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 7.4 shows a sequence diagram where WS-Discovery is used to detect a device, that is, sending a Probe message and receiving a Probe Match response from it. 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 initiated and finally, when an event happens, a notification operation named `myEvent` is received from the service.

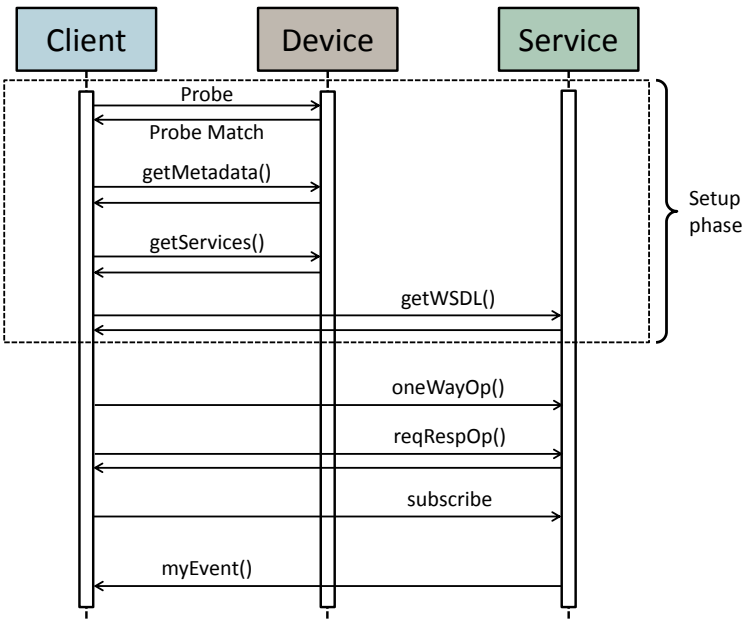


Figure 7.4 A sequence diagram for the interaction between a generic client, a device, and a service hosted by the device. First the device is discovered and the service's WSDL is fetched. Then operations in the service are called, a subscription is started, and finally a notification is received.

7.3 DPWS in JGrafchart

As shown in Figure 7.2, DPWS communication uses XML messages and thus DPWS requires sending strings. The already existing IO mentioned in Chapter 3.3 are Digital/Analog In/Out and Socket IO. Digital/Analog In/Out are insufficient as they only support Boolean and float values. Socket IO 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 for integrating DPWS in JGrafchart was implemented with Socket IO.

The Socket IO 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 7.5. 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.

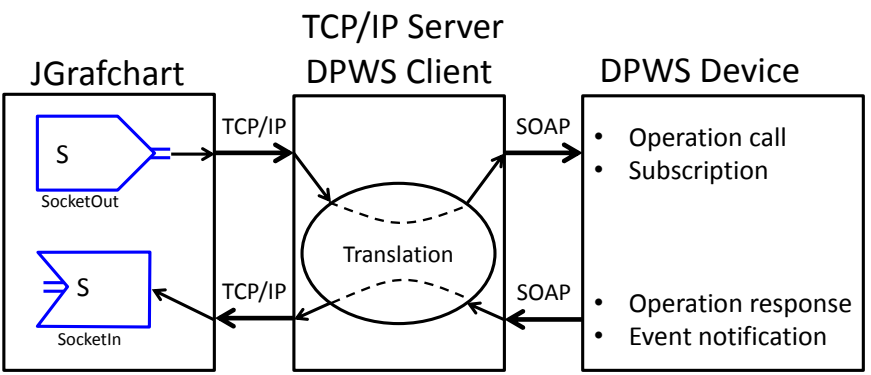


Figure 7.5 Overview of the Socket IO prototype.

JGrafchart only sends a message to the server if the value of the assigned socket output is changed. Since these assignments correspond to DPWS operation calls this means that consecutive calls with identical arguments are not made. A modified version of JGrafchart that sends a message to the server for all socket output assignments was used.

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 IO prototype is shown in Figure 7.6. Calling 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.

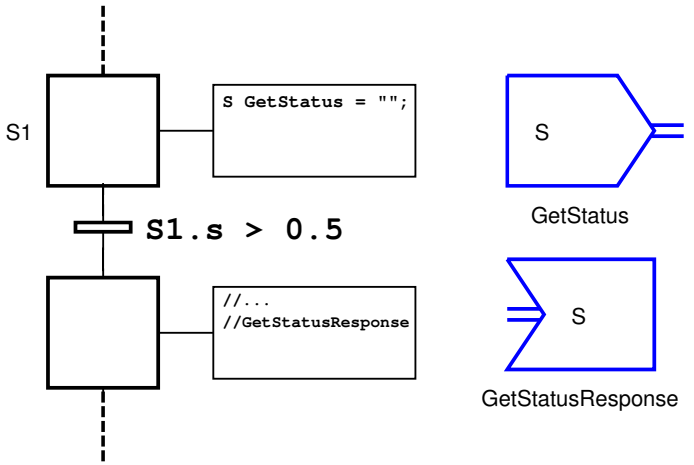


Figure 7.6 A request-response call with the Socket IO prototype. The call is invoked by the assignment to `GetStatus`. The TCP/IP Server translates this into a DPWS call. When the TCP/IP Server receives the response message it forwards it to `GetStatusResponse`. In the application the response is assumed to be available after 0.5 seconds. It would be possible to use extra socket IO to signal when the response is available but it would be even more complicated.

Integrated Generic DPWS

A generic DPWS implementation has been integrated directly into JGrafchart version 2.1.0 and later using the DPWS4J toolkit [43]. By using WS-Discovery, existing devices and device startups and shutdowns are automatically detected. By using WS-MetadataExchange, each service's WSDL is obtained. It is possible to browse available devices, services, and operations in JGrafchart, see Figure 7.7. Device metadata and WSDL documentation are also displayed.

To call DPWS operations a new IO element in JGrafchart called *DPWS Object* is bound to a portType, see Figure 7.8. 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.

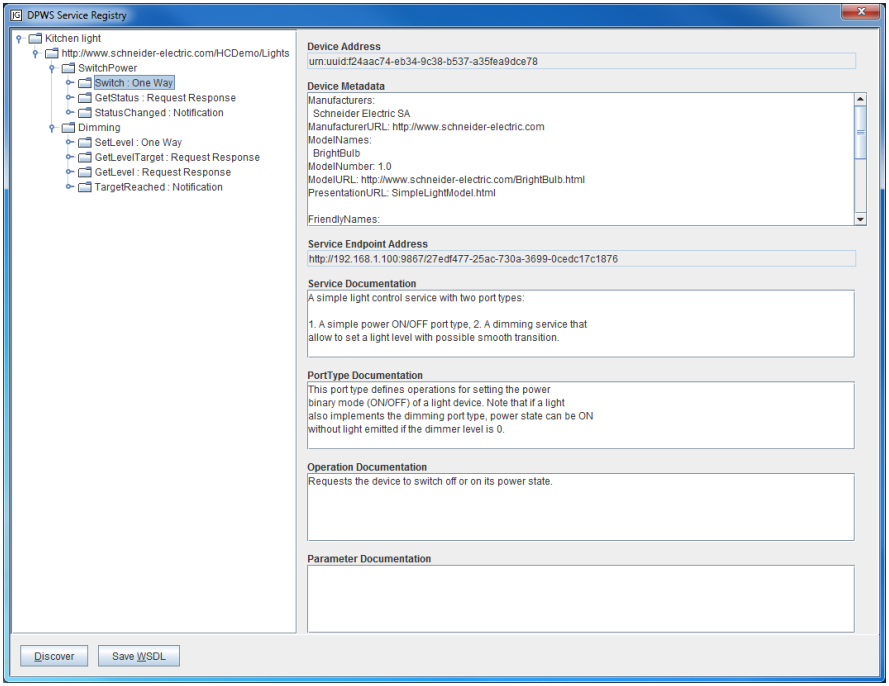


Figure 7.7 The service explorer in JGrafchart showing details about the available devices. The WSDL can also be saved, which is useful for example if the documentation is not sufficient.

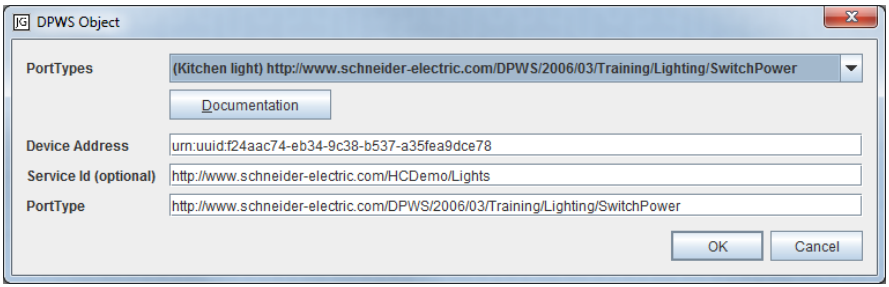


Figure 7.8 The configuration dialog for the new *DPWS Object*. It is bound to a specific portType of a specific service in a specific device.

Calling DPWS Operations

JGrafchart supports all operation types except solicit-response. DPWS operation calls look like ordinary method calls in JGrafchart, see Figure 7.9. Here the DPWS Object `myDPWSObj` is bound to the portType `SwitchPower`. First a 10 minute subscription is initiated. 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 various other DPWS related purposes.

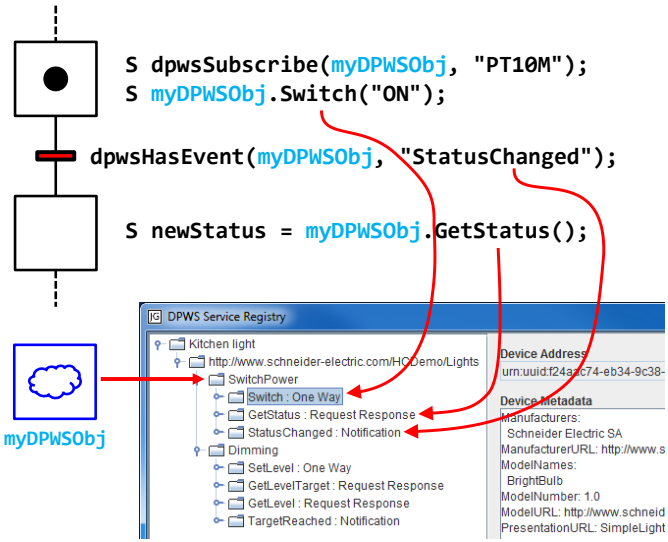


Figure 7.9 How to use the integrated DPWS feature in JGrafchart.

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. Improving this is planned for 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 7.9 has no content while `Switch` has content telling 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 dynamically 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.

Devices might not be present while editing or compiling. The editor, compiler, and execution engine may be on separate local networks and the devices would 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 when compiling, the compiler will just warn about not being able to perform these compilation checks.

7.4 Evaluation

SmartFactory^{KL} is a manufacturer-independent research platform [5] where for example demonstrators are created for evaluating 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 transporting carriers with the 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.

The quality control station was considered, see Figure 7.10. It consists of five devices: a sensor detecting the arrival of carriers, a stopper for stopping 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 sequence for coordinating the station can be modeled as in Figure 7.10. Based on the model a JGrafchart application for coordinating the station was implemented, see Figure 7.12. 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 are 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 selecting all elements with the tag name `value`, and `"sum"` is a built-in handler calculating the arithmetic sum of the selected elements' texts.

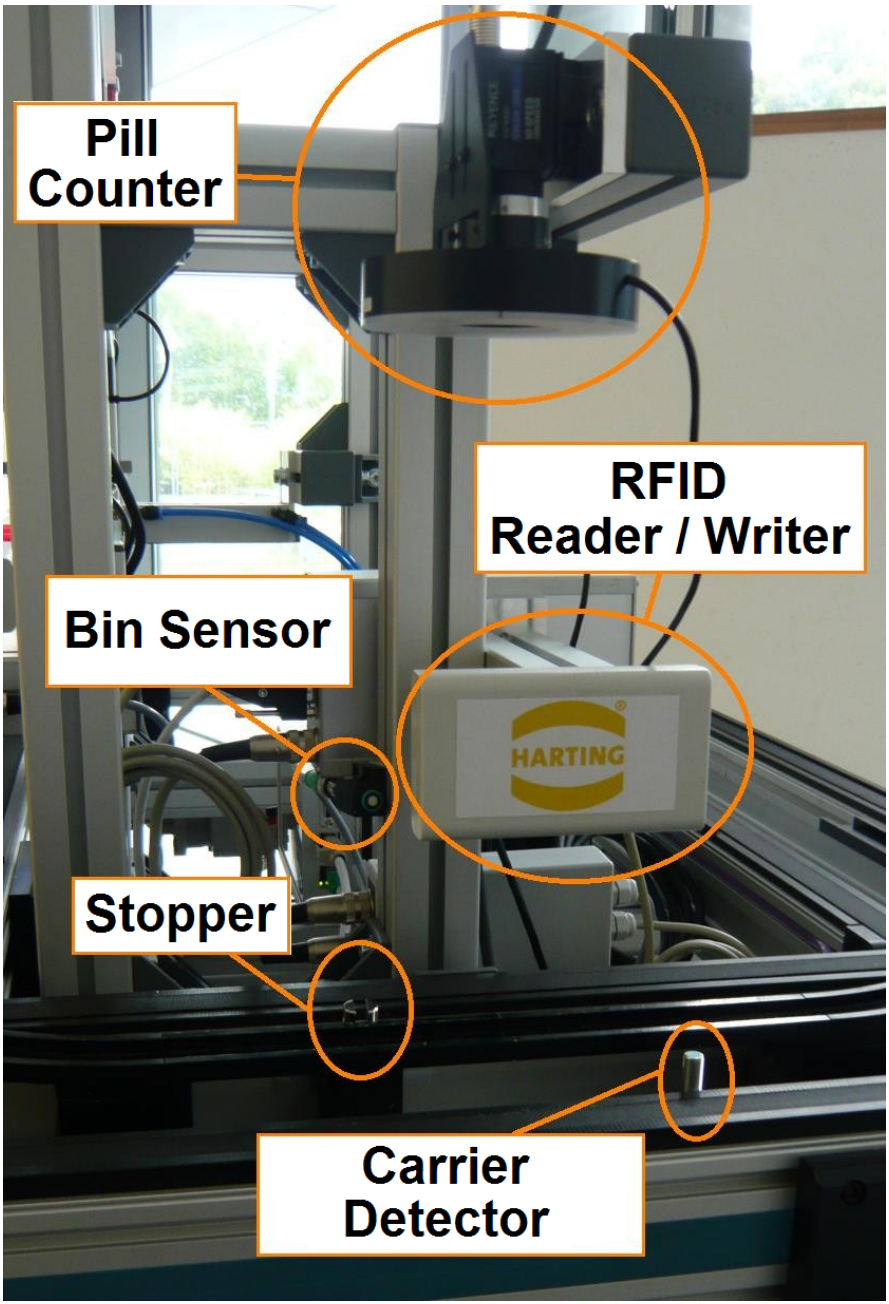


Figure 7.10 The quality control station of the SOA demonstrator.

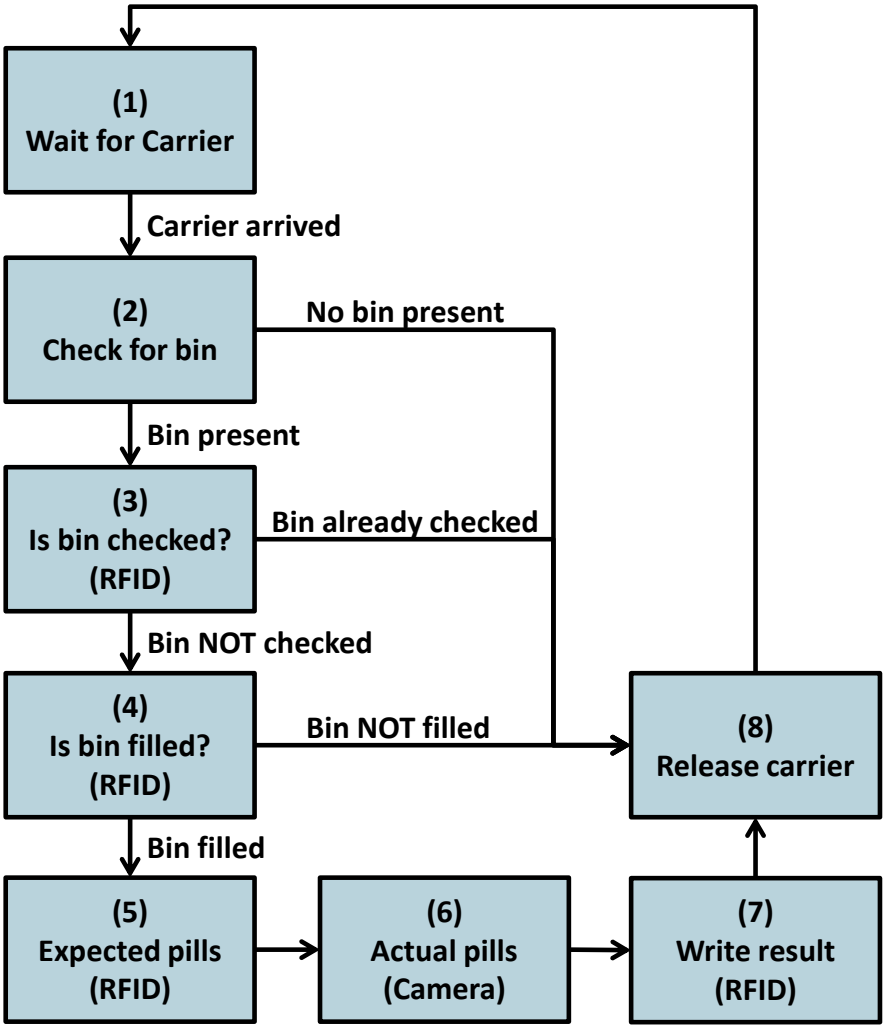


Figure 7.11 A conceptual coordination sequence for the demonstrator in Figure 7.10 where (1) is the initial state.

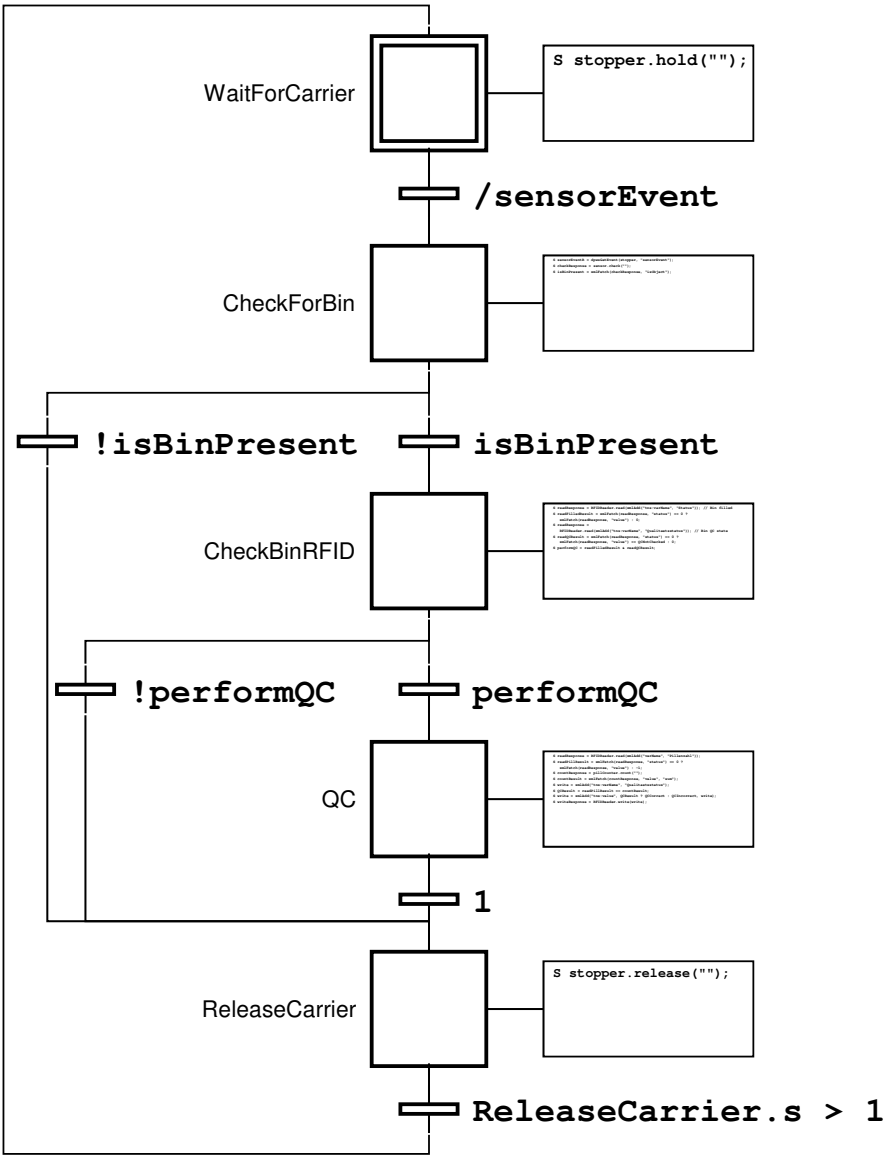


Figure 7.12 An orchestration for the demonstrator in Figure 7.10 implemented with the integrated generic DPWS feature of JGrafchart. The actions are shown to give an idea of the amount of code required, they are not intended to be readable.

7.5 Conclusions

SOA is a powerful design methodology which can improve flexibility and reusability of industrial automation systems. With SOA vertical integration comes for free since services can expose themselves directly to any level in Figure 1.1.

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.

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

8

Summary

Current trends in industrial automation are the need for customizable production, vertical integration, more advanced sensors and actuators, and shorter time to market. The currently used control systems and languages for control were developed with a more static production in mind. More flexible languages and tools are needed to get a more flexible production. The flexible graphical programming language Grafchart, based on the IEC 61131-3 standard language SFC, is considered with the focus to make it usable in an industrial context.

Modern compiler techniques were evaluated for JGrafchart with focus on extensible automation language implementations. In particular implementing HLV as an extension would make JGrafchart more dynamic and enable further research on HLV.

To make Grafchart possible to use at the lowest levels of automation, realtime execution with JGrafchart was considered. For this to be possible the execution engine must be separated from the editor. In the first step the execution engine is still an interpreter, but an order of magnitude faster than before.

Finally SOA, a highly flexible software design methodology widely used for business processes, is brought to automation by integrating support for DPWS in JGrafchart.

Table 8.1 lists the JGrafchart version history related to the work presented in this thesis.

Table 8.1 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 feature added.
2.1.1	Bug fixes.
2.1.2	DPWS feature improved.
2.2.0	DPWS feature improved.
3.0.0	(Future) Realtime execution.

9

Future Work

The ongoing work in Chapter 6 show promising results. The nextcoming work is to complete and evaluate this in detail. It would also be interesting to include the possibility to make changes to running applications in this work.

With the JGrafchart compilers implemented using ReRAGs it is possible to create various extensions. To implement a HLV extension is a natural the next step. Extensions for code generation are also of interest, for example to generate SFC code which can be executed in realtime by industrial PLCs.

Continuation of the SOA work includes linking it to the factory planning phase to get a more seamless commissioning procedure. Part of this work is creating an ontology relating Grafchart to languages used on other levels of Figure 1.1.

A

Other Languages

A.1 Petri Nets

Petri nets is a mathematical language for system modeling [16]. 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 placed in each output place.

With a Petri net model it is possible to analyze a system to determine if it for example is 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 is all of 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.

The properties are interesting since they have a corresponding meaning for the modeled system. For example it can be guaranteed that a dangerous state cannot occur if its corresponding marking is not reachable and a deadlock free Petri net guarantees that the system will not freeze.

A.2 Statecharts

Statecharts are known by many names such as Harel statechart, state diagram, UML state machine, and UML statechart and are used to describe the behavior of systems.

It is an extension to finite state machines (FSM) and consists of states and transitions. Statecharts are event driven, unlike Grafchart which is executed periodically. 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 hierarchically nested states, that is, a state may also have an internal state.

A.3 BPMN

Business Process Model and Notation (BPMN) is a graphical language for business process modeling based on flowcharts and similar to UML activity diagrams [44]. Business processes are used to formalize the proceedings of various tasks, for example the sequence of steps involved in releasing a product.

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 for splitting and joining 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.

Bibliography

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

[2] Wikipedia. *Automation*. URL: http://en.wikipedia.org/wiki/Automation (visited on 2013-01-25).

[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] IEC. *IEC 62264-1: Enterprise-Control System Integration – Part 1: Models and Terminology*. Tech. rep. International Electrotechnical Commission, 2003.

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

[6] IEC. *IEC 61131-3: Programmable controllers – Part 3: Programming Languages*. Tech. rep. International Electrotechnical Commission, 1993.

[7] 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).

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

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

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

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

[12] 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.

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

[14] 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.

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

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

[17] 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.

[18] 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).

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

[20] 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.

[21] T. Ekman and G. Hedin. "The JastAdd extensible Java compiler". *SIGPLAN Not.* **42** (10 2007), pp. 1–18. ISSN: 0362-1340. DOI: http://doi.acm.org/10.1145/1297105.1297029. URL: http://doi.acm.org/10.1145/1297105.1297029.

[22] 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.

[23] 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.

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

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

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

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

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

[35] A. Theorin, L. Ollinger, and C. Johnsson. “Service-oriented process control with Grafchart and the Devices Profile for Web Services”. In: *Proceedings of the 14th IFAC Symposium on Information Control Problems in Manufacturing (INCOM’12)*. Ed. by T. Borangiu, A. Dolgui, I. Dumitrache, and F. G. Filip. Elsevier Ltd, Bucharest, Romania, 2012, pp. 799–804. DOI: 10.3182/20120523-3-RO-2023.00131.

[36] A. Theorin, L. Ollinger, and C. Johnsson. “Service-oriented process control with Grafchart and the Devices Profile for Web Services”. In: *Service Orientation in Holonic and Multi Agent Manufacturing and Robotics*. Ed. by T. Borangiu, A. Thomas, and D. Trentesaux. 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.

[37] Wikipedia. *Service-oriented architecture*. URL: http://en.wikipedia.org/wiki/Service-oriented_architecture (visited on 2013-03-26).

[38] 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.

[39] 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.

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

[41] 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.

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

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

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