TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

ACIN
AUTOMATION & CONTROL INSTITUTE
INSTITUT FÜR AUTOMATISIERUNGS- & REGELUNGSTECHNIK

Model-Based Generation of an IEC 61499 Control Application for a Flexible Robot System

DIPLOMARBEIT

Ausgeführt zum Zwecke der Erlangung des akademischen Grades eines

Diplom-Ingenieurs (Dipl.-Ing.)

unter der Leitung von

Univ.-Prof. Dipl.-Ing. Dr.sc. techn. Georg Schitter
Dipl.-Ing. Dr. techn. Alois Zoitl

eingereicht an der

Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik
Institut für Automatisierungs- und Regelungstechnik

von

Matthias Plasch
Hans-Buchholzer-Straße 36/3
A-4400 Steyr
Österreich

Steyr, 6. April 2012

Gruppe für Industrielle Automationstechnik (IAT)
A-1040 Wien, Gusshausstr. 27-29, Internet: http://www.acin.tuwien.ac.at

Abstract

Today's manufacturing processes need to be fast adaptable to new customer demands, fast market changes and strong business competition. To cope with these requirements and further fulfil high quality requirements and small lot sizes, the appliance of flexible robotic systems gains increasing popularity. High flexibility of such systems can only be achieved if they are fast configurable and reconfigurable.

The intention of this work is to develop a model-based engineering method, which enables simplified programming of a control application for a flexible and modular robotic system. This method is based on Workflow Modelling that has emerged as promising technique to describe and optimize processes in different fields of activities. Additionally, Automatic Code Generation methodologies are applied to produce an executable control application out of the modelled process. Finally, the integration of 3D-Simulation technology into the engineering process allows early validation of the resulting control application, thus enabling iterative development.

A system architecture is presented that includes existing development frameworks and tools, which are extended with required functionalities. In this context, a graphical Workflow Modelling Language is introduced that facilitates the intuitive description of the process, which should be performed by the components of the considered robotic system. Moreover, a Code Generator is presented which generates the resulting control application that is conform with the industrial standard IEC 61499. The control application coordinates the components of the robotic system in order to realize the desired process behaviour.

First appliances of the developed engineering method to example applications, show reasonable results and encourage further development, especially towards more complex applications.

Kurzzusammenfassung

Rasch veränderliche Marktsituationen, individuelle Kundenbedürfnisse und wachsende Konkurrenz erfordern häufig eine schnelle Anpassung von Fertigungsprozessen an die neuen Gegebenheiten. Um diesen Anforderungen unter Einhaltung strenger Qualitätsbestimmungen und kleinen Losgrößen gerecht werden zu können, werden zunehmend flexible Robotersysteme eingesetzt. Die hohe Flexibilität solcher Systeme kann nur bei schneller Konfigurierbarkeit und Rekonfigurierbarkeit gewährleistet werden.

Ziel dieser Arbeit ist die Entwicklung einer modellbasierten Engineering-Methode, welche die vereinfachte Programmierung einer Steuerungsapplikation für ein flexibles, modulares Robotersystem ermöglicht. Der vorgestellte Ansatz basiert auf den Methoden der Workflow Modellierung, welche sich zur Beschreibung von Prozessen in unterschiedlichen Tätigkeitsbereichen zunehmend etablieren. Für die Generierung einer Steuerungsapplikation, basierend auf dem modellierten Prozess, werden Ansätze zur Automatischen Codegenerierung angewendet. Durch Einbindung der 3D-Simulationstechnologie in den Engineering-Prozess wird die Validierung und iterative Entwicklung der resultierenden Steuerungsapplikation ermöglicht.

Die im Rahmen dieser Arbeit vorgestellte Systemarchitektur, umfasst eine Anzahl bereits existierender Entwicklungswerkzeuge und Frameworks. Diese werden um die für die Eingineering-Methode notwendigen Funtionalitäten erweitert. In diesem Zusammenhang wird eine grafische Workflow Modellierungssprache eingeführt, welche die intuitive Modellierung des Prozesses erlaubt, der durch die Komponenten des betrachteten Robotersystems durchgeführt werden soll. Im Weiteren wird ein Codegenerator vorgestellt, welcher eine IEC 61499-konforme Steuerunsapplikation generiert. Aufgabe der Steuerungsapplikation ist es die Komponenten des Robotersystems so zu koordinieren, dass das geforderte Prozessverhalten realisiert wird.

Die ersten Anwendungsbeispiele, bei denen die entwickelte Engineering-Methode eingesetzt wurde, zeigten vernünftige Ergebnisse, welche eine weitere Entwicklung der Methode, für komplexere Anwendungsbeispiele, motivieren.

Acknowledgements

I would like to give my sincere thanks to my supervisors Univ.-Prof. Dr. Georg Schitter and Dr. Alois Zoitl of Vienna University of Technology for supporting my master's thesis. I want to express my gratitude to Alois Zoitl who always had time for discussions and questions, during my short visits to Vienna and several telephone calls. His advices, ideas and constructive comments on my work have been invaluable for me.

I am especially grateful to Gerhard Ebenhofer, for productive discussions regarding the development of ideas and solutions, and the great support he gave me during all of my work. Special thanks to Martijn Rooker and Harald Bauer for many insightful discussions, which often animated me to think out of the box. I would like to extend my thanks to Gerhard Ebenhofer and Martijn Rooker for proof-reading and commenting a major part of the text. Furthermore, I would like to express my gratitude to Profactor for giving me the opportunity to work on interesting research topics and to write this master's thesis.

I am much obliged to my fellow students and friends. The numerous hours we spent together for test preparations and the experiences we had in our spare time are great memories I will never forget.

My family receives my deepest gratitude and love for supporting me on my way, especially in difficult times. I am very grateful to my parents Manuela and Helmut who made my studies possible.

Steyr, April 6, 2012

Contents

List of Figures

List of Tables

1 Introduction

Enterprises that are especially positioned in the production industry, permanently face the difficulties of fast changing market trends, increasing market competition, and rising costs. At the same time, customers demand product variations, small lot sizes, and high quality in a short production time.

To overcome these requirements, the utilization of *flexible* and *mobile robotic systems* needs to be forced. Systems of this kind are equipped with mechanical components such as a mobile platform, which is used to navigate the robotic system within the work cell, or components for manipulating work pieces (e.g., manipulators and grippers). Additionally, these robotic systems make use of a set of sensors, such as cameras, laser sensors as well as sensors to measure physical quantities. The sensors are needed to capture information regarding the surrounding environment, where the robotic system is operating in.

Since *mobile robotic systems* are designed to carry out predefined working tasks, a means of *supervisory control* is needed, that coordinates the functional components, the robotic system is composed of. However, customary engineering approaches usually require much programming effort which is not necessarily restricted to the initial operation, thinking of changes in the working process which need to be done, during the operation life time of the robotic system. Additional difficulties arise if the system's components are provided by different vendors and should interoperate, in order to perform the desired process.

The rising need for fast configuration of flexible robotic systems as well as re-configuration, raises the demand for preferably *user-friendly* engineering tools, which enable *simplified programming*, *configuration* and *maintenance* of such robotic systems. Factory workers, who are not software or automation engineering experts but familiar with the production processes, should be able to program these systems.

Some of the requirements and needs that are mentioned above, are underpinned by the increasing popularity of *Distributed Automation Systems*, used to solve control application problems in the industrial field. These requirements include functional units composed out of several intelligent devices, interoperability between devices of different vendors, and reconfiguration. The main idea of distributed system architectures is to integrate control intelligence into hardware devices, such as actuators or sensors. By linking those *functional autonomous* units together in a communication network, they can interoperate and provide highest flexibility for the realization of a concrete control behaviour. This architecture describes a trend, leading away from clumsy, centralized control units which are difficult to maintain, to distributed intelligence, enabling reusability and effective verification of these modules.

For process descriptions in different fields of work, including business, production or health care, *Workflow Modelling* has emerged as promising technique in order to visualize, clarify, structure and optimize the steps of processes. The basic idea is to describe a process as a combination of intuitive processing steps which are linked together, forming a defined execution sequence. Based on process simulation results, the process description can be modified and adapted to fulfil new requirements. An engineering approach that combines intuitive modelling of processes with the flexibility of distributed system architectures, might meet the requirements for flexible robotic systems that can be programmed and (re)-configured with a high grade of usability.

The intention of this work is to show that *Workflow Modelling* is suitable for the simplified programming of supervisory control application, that is needed to coordinate system components which perform the execution of a predefined process. In contrast to common programming methods, the modelled control behaviour is designed as simple as possible and independent from a specific platform. The modelled control behaviour shall be transformed into a control application by using code generation concepts. The resulting control application can be executed by a distributed automation system.

1.1 Aim of the Work

The aim of this work is to develop a method which enables simplified programming of the behaviour of a distributed system architecture, especially of flexible robotic systems. This means that the control functionality of a supervisory control application, which coordinates the robotic system's autonomous components, shall be determined by means of a process description that is expressed using a *Workflow Modelling Language*. In order to close the gap between the modelled supervisory control behaviour and an executable program, methodologies for *code generation* need to be applied to create a Function Block application, according to the standard IEC 61499. To enable an iterative programming process, the generated Function Block application can be run not only on the hardware target system, but also in combination with a *Simulation Environment*. The simulation results can then be used to improve the modelled supervisory control behaviour, henceforth minimizing design errors, before running the application on the real hardware target system.

Relevant research questions, which are taken in account are “Can graphical Workflow Modelling Languages be effectively used to enable simplified programming of control applications?” or “Is it possible to generate a supervisory control application, according to IEC 61499, by having available shallow information regarding the behaviour of the robotic system’s components as well as a process description?” and “Which prerequisites need to be fulfilled by the resulting system architecture of such a model-based engineering approach?”.

1.2 Objectives of this Work

Facing the main objective of keeping the engineering process as intuitive as possible, the following objectives can be derived.

- Based on a thoroughly literature survey regarding *Workflow Management Technology*, a *graphical Workflow Modelling Language* which allows intuitive modelling of control functionality, has to be developed.
- The behaviour of the components, of which a flexible robotic system is composed of, needs to be described independently on how the functionality is implemented by a specific component vendor.
- A code generation tool needs to be developed which generates and parametrizes an IEC 61499 compliant Function Block Network, that represents the modelled system behaviour. This supervisory control application is generated by using the component behaviour information as well as the designed *Workflow Specification* which describes the control functionality.
- Finally, test cases need to be implemented to show the applicability of the model-based engineering approach, that is proposed in the scope of this work. Furthermore, this test examples should reveal the potential of the implemented prototype which is promising for future development work.

1.3 Guideline Through the Work

In Chapter 2 the fundamentals, as well as the current state of the art are presented which form the theoretical basis of this work. The first part of this chapter deals with the fundamentals of the *Workflow Management Technology.* Secondly, different characteristics of *Workflow Modelling Languages* are discussed and the most important language implementations, which have influenced this work, are presented. Thirdly, the fundamentals of the standard *IEC 61499*, for modelling distributed control systems, are discussed, including a short description of standard compliant development tool implementations. The fourth part deals with a brief overview of 3D Simulation and lists available implementations of 3D Simulation Toolkits. Chapter 2 concludes with an overview of *code generation* technologies.

Based on the introduced concepts, an overall system architecture, which underpins the developed modelling approach, is derived in Chapter 3. Facing a desired, intuitive model-based *engineering process*, the requirements for such a system are identified for the development of a Workflow Modelling Language that satisfies the application domain specific needs as well as for the conception of the code generation method.

Chapter 4 deals with the proposal of a graphical Workflow Modelling Language, which has been developed to overcome the requirements that are stated in the previous chapter. Besides the general language elements, encompassing workflow activity types, data interaction and execution order, some extended concepts including exception handling and possibilities for hierarchical workflow structuring and workflow verification are discussed.

Based on the proposed Workflow Modelling Language, Chapter 5 focuses on the description of the *code generation method* which is needed to generate a Function Block application, out of the modelled Workflow Specification. The generation of Function Block types, representing the workflow activities, is shown based a specific example. Furthermore, the resulting structure of the generated Function Block application is explained.

Chapter 6 deals with the implementation of the presented approach and includes a description of the frameworks and technologies that have been applied in order to produce the program code. The resulting plug-in structure as well as the essential functionality, provided by the implement classes, is explained. In Chapter 7, test cases are introduced which are used to show the results of the implemented approach, that are analysed and discussed. Finally, Chapter 8 summarizes the results of this work and provides development steps, that are considered for further research.

2 State of the Art

This chapter gives an overview of the theoretical background, which is essential for this thesis. The first two sections introduce basic concepts of Workflow Management including important definitions, and characteristics of Workflow Management Systems. Moreover, a set of implemented and available Workflow Modelling Languages is introduced.

The second part provides an overview of the basic concepts of the international standard IEC 61499. Hereby, emphasis is put on the Function Block concept and the reference models which describe the distributed system architecture. Control applications, based on the IEC 61499 standard, can be seen as the preferred outcome of the engineering method which is proposed in the scope of this thesis.

In the third part, a brief overview of 3D-Simulation Environments is given, which discusses general characteristics of simulation tools and lists some example implementations of 3D-Simulation toolkits. The application of simulation techniques enables the iterative improvement of the modelled control behaviour through continuous validation by simulation. Finally, an overview of code generation techniques is given, which are intended and suitable for reducing the programming effort, needed for the implementation of the method described in the subsequent chapters.

2.1 Introduction into Workflow Management

Workflow Management (WfM) technology has been successfully used for describing business processes using Workflow Specifications (see e.g., [25], [20], [54], [52] or [40]). Moreover, WfM provides methodologies for optimization, re-design and standardization of the specified processes. The hereby caused rationalization of workflows and automation of work delivery result in a number of promising benefits:

- In general, modelling leads to more detailed insights into the processes which should be described.

- Process optimization can result in better quality and can simultaneously reduce the cost, as result of the shortened process time.
- If the use of a Workflow Specification helps to improve the process transparency, the current state of a process and the decisions, which have been taken during its progress, can be tracked more easily.
- Correctly defined Workflow Specifications *ensure* that tasks are executed only when the prerequisites, for their correct processing, are fulfilled.

Taking the advantages of this developing technology, business enterprises are more likely able to deal with global competition, reduced *time-to-market* and to meet the requirements of fast changing product and service variants under high pricing pressure [18].

During the development of WfM technology, an apparently endless list of WfM products have been introduced by different vendors and the amount of available tools is growing continuously (e.g., [46] [36] [69] and more[1]). Despite the fact of large product variety, several analyses of WfM products come to the result that all implementations share the same common characteristics [20]. The *Workflow Management Coalition (WfMC)* [65] aims to develop a widespread standard for the implementation of workflow modelling products. As a result of this attempt, a description of the components and characteristics of *Workflow Management Systems (WfMS)* has been proposed [20]. The following sub-sections give brief explanations of the terms related to the WfM technology mentioned above. Further terms and definitions are provided in the glossaries of [66], [25] and [18].

2.1.1 Workflow and Workflow Specification

Before the term *Workflow Management System* can be explained, the definitions of the terms *Workflow* and *Workflow Specification* are given at first.

Definition: The WfMC-glossary [66] defines *Workflow* as:

> *"The automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules."*

Jablonski et al. [25] suggest another definition which says that a *Workflow* has got a defined 'starting point', an organized operation sequence and a defined 'end point'.

[1]An extended list of workflow modelling tools is presented in [56].

Workflows can be partly automated and consist of a collection of work items - so-called *activities* - which are designed to carry out a defined business process or an organizational procedure. Moreover, workflows contain activities which can be composed into *Sub-Workflows*[2]. Activities are associated with *work items* which require *resources* (more generic: *Workflow Participant*) in order to get executed successfully. These resource types can range from human workers to machine resources. At this point it is worth mentioning that the term *Workflow* traditionally is associated with business processes (e.g., "issue an invoice"), but the same concepts can be applied for complex manufacturing procedures [20].

Definition: A definition of *Workflow Specification* is suggested in [18]:

> *"A Workflow Specification is created based on a Workflow Modelling Language, which can be represented either in textual or graphical form, using predefined workflow activities and language specific control structures. The Workflow Specification serves as an abstract representation of the real process. Its execution is supported by the Workflow Management System."*

The term *Workflow Modelling* describes the procedure of creating a Workflow Specification with the help of a Workflow Modelling Language. In order to prevent confusion it is mentioned that the notion *Workflow Model* is mainly used instead of Workflow Specification in the relevant literature (see e.g., [58], [25] or [47]). Figure 2.1 shows the schematic structure of a Workflow Specification, consisting of a set of activities (equivalently named *tasks* in the figure) and the execution order, illustrated by solid arrows. Workflow Management Systems might support the independent execution of a number of workflow instances (workflow *case*) simultaneously. A Sub-Workflow (*block task*) is processed by directing the execution control to its first incorporated task. After completion of all incorporated tasks, the block-task regains execution control and has finished its execution. A *Multiple-instance task* constitutes the concurrent execution of a number of instances of the same task type. Multiple-instance tasks are considered to be completed when a predefined number of task instances have been processed successfully [59].

If a Workflow Specification is being created or changed, its design has to comply with a set of rules at any time[3]. These rules can be derived from a so called *Modelling Paradigm* which is primarily represented by the *Workflow Meta-Model.* A *Workflow Meta-Model* provides a formal description of the workflow modelling environment

[2]A Sub-Workflows is defined as an aggregation of activities and enables hierarchically structuring of a Workflow Specification to reduce complexity.

[3]Usually including data type checks, constraints for connections between activities or rules regarding the structure of a Workflow Specification.

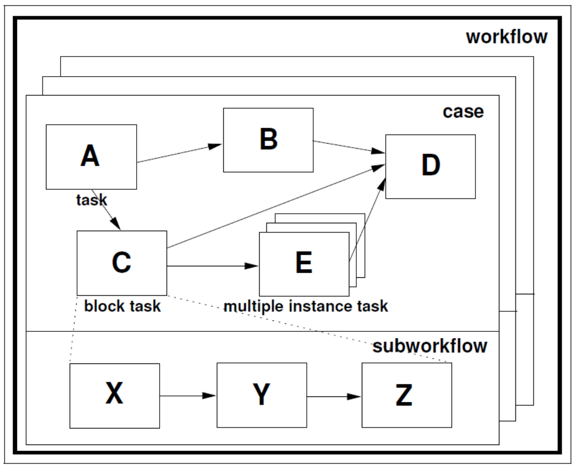


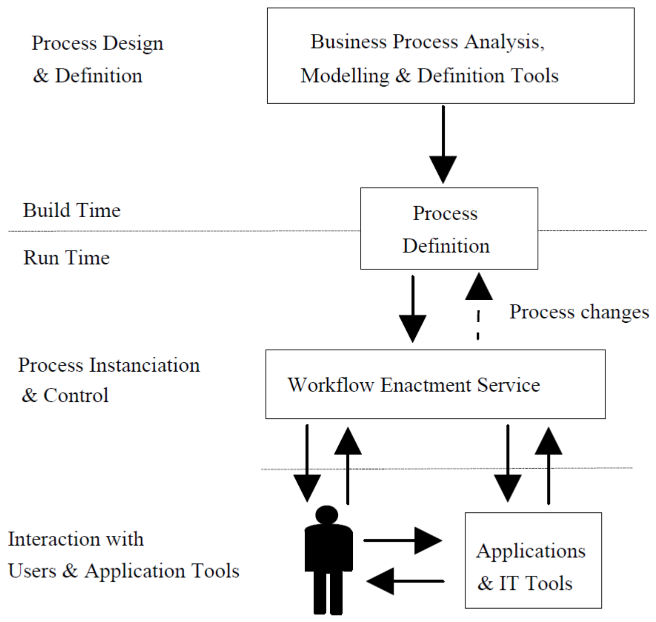
Figure 2.1: Structure of a Workflow Specification [59]

[25]. The Modelling Paradigm defines the syntax and semantics of the *Workflow Modelling Language*.

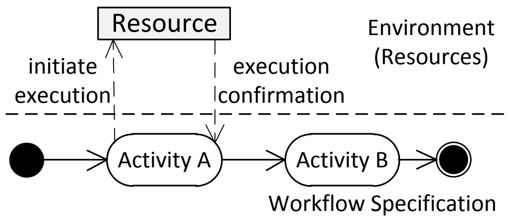
2.1.2 Workflow Management System

A *Workflow Management System* (WfMS) consists of a set of components which support specification, the managed execution and the monitoring of workflows. The functional components of a WfMS are shown in Figure 2.2(a) and can be described as follows [20]:

- Build-time functions for the definition of the workflow, based on a business process analysis, and for the derivation of the Workflow Specification (i.e., the Process Definition).
- The *Workflow Enactment Service*, can be treated as the core control software of a WfMS, providing run-time functions including the interpretation and execution of the Workflow Specification.
- Predefined interfaces enable the *Workflow Enactment Service* to interact with the available *Workflow Participants* (resources) for work item distribution and to fetch the processed working results. Alternatively, the work items can also get processed by available applications and IT tools. The utilization of IT tools by human workers, for work item procession, and the corresponding data interchange is represented by two anti-parallel arrows.



(a) Components of a Workflow Management System [20]



(b) Reactive Workflow Execution

Figure 2.2: Components of Workflow Management Systems and reactive execution of Workflow Specifications

- In some cases it is necessary to dynamically change the *Process Definition* during the execution of the workflow instance, as indicated by the dashed feedback arrow.

A WfMS interacts with its environment, enclosing resources and external services, and therefore can be denoted as an *open, reactive system* [14]. In this context *reactivity* means that the WfMS can respond to events, received from components of the environment. According to Eshuis et al. [14], reactive WfMSs do never process any work items itself, but distribute them among the available resources and trigger their execution. Figure 2.2(b) depicts an example for reactive execution of a Workflow Specification. The WfMS assigns the activity *A* to the *resource* and triggers its execution. After the WfMS has received an execution confirmation from the resource, the execution of activity B is initiated.

Contrary to reactive systems, *active systems* control themselves and cannot react to query events emitted by its environment. Active systems are always called *closed* [14]. Reactivity is an essential characteristic which enables the WfMS to coordinate effectively the available resources. In order to achieve the goals of this thesis, described in Section 1.1, it is necessary to model the required control behaviour by means of a Workflow Modelling Language which supports the *reactive execution* of the specified workflow [4].

2.1.3 Activity Life Cycle

The *Activity Life Cycle* [59] denotes a scheme to describe the sequence of states an activity runs through, beginning with its initial creation. In the second state, the activity is offered to one or a group of resources. Afterwards, the assignment to a concrete resource is initiated either by a resource or by the WfMS. Finally, the execution is commenced and completed or suspended. Generally, transitions from one activity state to another can be initiated either by the WfMS or by the resources to which an activity is assigned to.

Aalst et al. [59] propose an explanation of the life cycle by means of a state chart. The relations described in the *Activity Life Cycle* state chart are useful for the derivation of activity distribution strategies, as well as workflow exception handling strategies (see e.g., [59] [60] [52]).

2.1.4 Workflow Validation and Verification

Validation and Verification analyses are of great importance for organizations using WfM technology in order to ensure correctness of Workflow Specifications. The term *Validation* can be explained as an inspection of the functional behaviour of the Workflow Specification. This check is usually based on the simulation of predefined fictitious workflow cases and the evaluation of the processing results. A Workflow Specification can be checked for *structural correctness* by using *Verification* analy-

[4]Note: Reactive execution of a Workflow Specification does not only depend on the capabilities of the WfMS but also on the formal description of the Workflow Modelling Language (i.e., its semantics and syntactical elements).

ses which encompass syntax and semantic checks as well as data type constraints checks.

A WfMS should periodically perform thorough checks of the Worfklow Specifications to reduce the risks of production loss, decreasing throughput rates, or product recalls [55]. Various *Verification* methodologies have been developed mainly for *Petri-Net* based Workflow Modelling Languages, ranging from algebraic techniques to methods of the graph theory (see e.g., [55], [47], [7], [10] or [31]). Further verification methods are introduced along with the presentation of currently available Workflow Modelling Languages, according to the state of the art research.

2.1.5 Workflow Optimization

Especially in complex and large Workflow Specifications, the application of *Workflow Optimization* strategies is indispensable in order to reduce workflow processing durations. Although there has not been put much effort into the development of optimization methodologies (see e.g., [55] and [7]), two basic optimization patterns can be applied to Workflow Specifications in general, which are described in the following [7].

Two activities *A1* and *A2* can be executed in parallel (concurrently) if the processing of *A2* is independent from the results of *A1* and no critical resources are needed by both activities at the same time. Instead of using sequences of activities of the same type (e.g., $A1 \rightarrow A1$ et cetera), loop-structures should be introduced. With regards to the perspective of a Workflow Management System, the utilization of loop-structures results in the advantage that activities within the loop might only need to get instantiated once. Moreover, activities of the same type can be rescheduled in a way that they get executed by the same resource[5].

Gruber [18] presents *Workflow Transformations* which do not change the semantic of a Workflow Specification but change the specification in a way that certain pairs of activities can be executed in parallel thus leading to improvements.

[5]In this case a resource does not need to reload the context of the activity every time but only restart its execution, for instance with updated input parameters, which reduces the execution time [59].

2.2 Workflow Modelling Languages

Workflow Specifications are developed with the support of *Workflow Modelling Languages* which provide the necessary semantics and syntax for the workflow expression in a formally correct way. Workflow Modelling Languages are either represented in textual[6] or graphical[7] form.

One of the main targets of this work is to develop a graphical Workflow Modelling Language which enables a simplified expression of control behaviour for a flexible robotic system. By keeping these requirements in mind, a thorough literature search, enclosing different Workflow Modelling Language implementations, has been performed. The Workflow Modelling Languages and their characteristics, which have influenced this work most, are presented shortly in this section. Moreover, a set of *modelling aspects* and *design patterns*, which are of great importance for the design of a Workflow Modelling Language and the supporting Workflow Management System, are introduced and explained.

2.2.1 Workflow Modelling Aspects

A collection of aspects which should be considered for the design of a Workflow Modelling Language, has been identified by several researchers (see e.g., [25], [55] and [47]). These viewpoints cover design questions for Workflow Modelling Languages as well as for the functionalities of a Workflow Management System.

- **Behavioural aspect:** The behavioural aspect (or *Control-Flow* aspect) determines the execution sequence of the activities included in the Workflow Specification and therefore describes dependencies of subsequent tasks (see [66] [61]).
- **Informational aspect:** Workflow data and data dependencies between successive activities are considered in the data perspective[8]. There are several possibilities how to transport data within a workflow case and how it is passed from one activity to the next [58].
- **Organizational aspect:** The organizational aspect deals with the allocation of work items to available resources, the satisfaction of allocation constraints and the handling of allocation conflicts. Work item allocation can happen

[6]Examples: BPEL (see Section 2.2.4), Temporal Logic (see Section 2.2.7).

[7]Examples: Petri-Nets (see Section 2.2.3), UML Behaviour Diagrams (see Section 2.2.5).

[8]In general we can distinguish between *Control Data*, needed by the WfMS in order to operate, and *Production Data* which exists independently from the WfMS [58].

statically, which means that activities of a certain class are allocated to a determined resource by default or dynamically. If allocated dynamically, work items are allocated according to defined allocation strategies [59].

- **Functional aspect:** The functional perspective deals with the definition of each task in the Workflow Specification. An overall task description includes input parameters, output parameters, the processing steps and configuration parameters[9]. Tasks may update control data as well as production data variables [18].
- **Operational aspect:** The operational perspective describes the interactions of the WfMS with its environment. Elementary activities can require access to external applications or resources to update Control and Production Data [47].
- **Exception Handling aspect:** Unexpected events occurring during workflow execution might lead to deviations in the execution sequences of the Workflow Specification. Exception Handling strategies are then applied to further ensure correct workflow execution [60].

In the next step the question may appear, how these viewpoints can be applied correspondingly to the implementation of a Workflow Modelling Language. To ease the development process, Aalst et al. [56] propose a collection of design patterns.

2.2.2 Workflow Patterns

Workflow Patterns describe reoccurring *design structures* found in implemented Workflow Modelling Languages and cover the functionalities corresponding to the modelling aspects introduced above [56]. Indications for using Workflow Patterns include the evaluation and comparison of Workflow Modelling Languages in terms of their expressiveness, as well as a guideline for the development of Workflow Modelling Languages and Workflow Management Systems [56]. The following paragraphs give a brief overview of the four groups of Workflow Patterns, identified by Aalst et al. [56]. Detailed information regarding Workflow Patterns are provided in [56].

Control-Flow Patterns

Control-Flow Patterns are concerned with the execution sequence of the activities within the Workflow Specification. The basic pattern types describe control-flow *routing structures*, for instance *activity sequences*, synchronizing patterns including

[9]These may include constants such as execution deadlines and task priority numbers.

AND-Join, *XOR-Join*[10] and the basic branching patterns *parallel split* and *XOR-Split* which can be compared to a if-then-else decision routing pattern [61], [66]. More specific control-flow patterns are described in [61].

Data Patterns

Data Patterns aim to express the possibilities how data object can be represented, transported, and modified within workflow instances [58]. *Data Visibility* patterns express to which workflow constructs (e.g., *task*, *block-task*, or *workflow case* level) data objects are bound to and the scope of the Workflow Specification, where they are visible. *Data Interaction* patterns describe approaches how data transport can be arranged between workflow constructs [58]. These include combined data and control-flow channels, distinct data and control-flow channels, and data exchange through data passing, using shared variables [58].

Resource Patterns

Resource Patterns refer to the *Organizational Aspect* and describe how resources are represented and dealt within a WfMS [59]. Additionally, the distribution of activities among resources is covered by Resource Patterns. *Creation Patterns* determine at design time to which resources activities should be assigned to[11]. Based on this information, *Push and Pull Patterns* describe the current distribution of the activities to the resources, where "push" denotes that the distribution is initiated by the WfMS and "pull" describes the allocation of an activity is initiated by the resource. *Detour Patterns* refer to situations where the distribution of an activity needs to be undone (e.g., suspension of an activity or resource overload). Detailed information about *Resource Patterns* is provided in [59].

Exception Handling Patterns

Workflow Exception Handling is indispensable in a WfMS in order to deal with unexpected events that may arise during workflow execution in order to keep the executed workflow instance (case) in a defined state. Common exception types occur related to *work item failures* including *execution deadline expiry*, *resource unavailability* or *triggered exceptions*. Triggered exceptions are events which require special

[10]The XOR-Join pattern joins different execution branches to one single sequence branch. Control-Flows arriving simultaneously are excluded.

[11]Examples are: *direct allocation* of certain activity types to one designated resource, or *role-based allocation*, where activities are offered to a group of resources, sharing similar characteristics [59].

treatment. A WfMS should be capable of detecting exceptions and provide appropriate handling strategies [60], [31]. There are two general approaches, how exception handling functionalities can be integrated into a WfMS, namely *embedded* into the Workflow Specification as alternative branches or *autonomous* (i.e., independent of the Workflow Specification), which requires the knowledge of the occurring context of an exception [52]. By using an *embedded solution*, the simple definition of handling strategies is enabled and the exception origin is clearly defined within the Workflow Specification. *Autonomous solutions* provide high expressiveness, which enables the description of the exception occurring context by using more complex conditions [52].

The basic *activity handling strategies*[12] can be derived from the Activity Life Cycle and include for instance the *restart* of activity execution, *re-offering* of activity to other applicable resources, and *forcing* the activity to be failed or completed. Execution of Workflow Instances can be *continued* as intended or *removed*, which means that all activity instances are cancelled and removed [60]. *Recovery Actions* denote the finalizing part of an exception handling strategy and are needed in order to compensate the effects, brought by an exception. A common approach is *Forward Recovery* where an alternative path through the Workflow Specification is executed, to compensate the exception effects [31]. *Knowledge-Based Handling* builds on similar, previously happened exception cases, which are used to derive a new recovery action through adaptation of old handling strategies [52] [31].

After gaining some insights into workflow modelling aspects and the resulting design patterns, a selection of Workflow Modelling Languages including their characteristics, which have influenced the development work of an application specific language, are presented in the following sub-sections.

2.2.3 Petri-Net Based Languages

Workflow Modelling Languages based on *Petri-Nets (PNs)* have been thoroughly investigated, since the establishment and development of Workflow Technology (see [55], [7], [10], [54] or [15]). PNs consist of a set of *places* and *transitions* which are connected to each other via directed arcs, whereat two transitions or two places may not be subsequently connected. The current state of a PN is determined by *tokens* that are assigned to places (i.e., the *marking* of places). A state change is induced

[12]Besides the affected work item (activity), also the underlying workflow instance needs to be handled since activity failures also have impact on the workflow instance and the other contained activities [60], [52].

by firing a transition, once the firing rule is satisfied, causing the number of tokens belonging to the preceding places being reduced and passed to the successive places. A formal definition of a Petri-Net is given in [7] or [55].

To improve expressiveness, PNs can be extended to *High-Level PNs* by using *coloured tokens* (i.e., tokens extended with data attributes) to enable the use of firing conditions for transitions[13]. *Timed PNs* allow temporal constraints to be applied by assigning *time delay values* to transitions, places or arcs. Dahms [10] uses *coloured* and *timed* PNs for the description and control of event-discretely modelled production systems.

With reference to Workflow Modelling, activities can be represented by *transitions* or *places* like described in [7] and [15]. Events received from external resources can be represented by transitions, labelled by the considered event, or through tokens appearing in certain places [55], [14]. According to Eshuis [14], PNs do not provide appropriate constructs to represent data objects reasonably.

The main advantage of PNs are their formal semantics and their mathematical definition, enabling effective evaluation and verification methods (see e.g., [30], [10], [7] and [55]). The main problems of Petri-Nets underlie the fact that PNs model *active, closed* systems whereas WfMSs need to be *reactive and open* [14]. *Reactive Petri-Nets*, introduced by Eshuis and Dehnert [15], as well as *Yet Another Workflow Language (YAWL)*, proposed by Hofstede et al. [57], [55] are intended to overcome the disadvantages of classic PNs.

2.2.4 BPEL - Business Process Execution Language

The *Business Process Execution Language (BPEL)* (or WSBPEL - Web Services BPEL) was developed to provide a standard for the description of business processes based on the orchestration of *Web Services*. Activities in BPEL correspond to Web Services which are already implemented. By means of *orchestration*, Web Services providing basic functionalities can be combined in order to create higher levelled services. BPEL allows to specify how to coordinate the execution of Web Services according to a process definition [68]. Workflow Specifications are described textually, based on the BPEL XML-Scheme, and can be interpreted and executed by the *BPEL orchestration engine* which corresponds to the *Workflow Enactment Service* of the BPEL WfMS.

[13]A firing condition uses an attribute value in a boolean expression. If the expression evaluates to *true*, the transition is activated and fired if the firing rule is satisfied [10].

As a main benefit of BPEL, the clear separation of internal business processes (e.g., belonging to a certain enterprise) from processes in the external environment (e.g., business partners, service providers) is possible without affecting the communication channels upon internal process changes [68]. BPEL is platform independent and portable to other Workflow Modelling Languages, due to its XML based language constructs. The combination of existing service implementations to "new" services improves reusability and interoperability [5], [40]. The language extensions *BPEL4People* [41], [42] and *BPEL-SPE* [22] enable BPEL to support *human interaction* patterns and the definition of sub-processes (compareable to sub-workflows). A number of commercial and open-source BPEL engines have been released during the development of BPEL, such as *ORACLE BPEL Process Manager*, *IBM BPWS4J*, *Twister* (Apache), *Microsoft Workflow Foundation* and *Active BPEL* [67].

2.2.5 UML Activity Diagrams

The *UML Activity Diagrams* (UML - Unified Modelling Language) can be effectively used for workflow modelling. A variety of tools are available for the design of UML Activity Diagrams such as *Eclipse UML2Tools*, *StarUML*, *astah**, *Papyrus* and many more [50].

Activity Diagrams are used to describe different *application flows*, which describe the functionalities of a system, using activities[14]. They consist of *activity nodes*, each representing a work item, as well as outgoing transitions [43]. A transition firing means that the execution of the source activity is finished. Every Activity Diagram may contain exactly one *start node* but can contain at least one *end node.* To reduce complexity in Activity Diagrams, the *sub-activity* construct enables the designer to nest functional connected diagram parts into a user defined activity [43], [50]. Activity Diagrams support basic *Control-Flow Patterns* described in Section 2.2.2. In order to receive and emit signals (events) asynchronously, the UML standard provides *receive* and *send* signal nodes [50]. Workflow Specifications based on Activity Diagrams are suitable to be executed by a reactive WfMS [14].

Dumas and Hofstede [12] benchmarked Activity Diagrams with promising results. The notation of sub-activities combined with signal senders and receivers allows effective workflow modelling. Regarding the control-flow and data perspectives (see Section 2.2.1), Hofstede et al. [62] conclude that most of the patterns proposed in

[14]By assuming that the application flows can be automated, they can directly be compared to workflows.

[61] and [58] are supported directly. However, Activity Diagrams lack in providing support for *resource patterns*.

2.2.6 JECA Rules for Workflow Modelling

The functional behaviour of *reactive systems* can be described by using *JECA* rules (*justification-event-condition-action*) which determine how to react (perform an action) on defined events and, at the same time, a specified boolean condition is fulfilled. An action A is performed if and only if the event E occurs *and* the condition C evaluates to *true* [14], [31]. The justification J corresponds to a condition which can be used to disqualify[15] a JECA rule if its evaluation is not relevant in the execution context.

Workflow Specifications based on JECA rules consist of states which are connected among each other through transitions, and have a defined start and an end state. JECA rules label transitions which fire if the corresponding rules are enabled and evaluated thus forming an execution sequence. If the *initial rule* enables the *final rule* through a path of workflow states and rules, for all permitted branches of the Workflow Specification, the workflow is called *correct*. Otherwise execution can result in a *deadlock* [31].

Eshuis [14] suggests the combination of JECA rules and *Activity Diagrams*. A JECA rule then labels a transition which activates the subsequent activity node. Therefore, every node emits a *termination event*, after the associated activity has been executed successfully, which triggers the next JECA rule. Eshuis [14] suggests to only permit the emission of events within an action, since they are executed by the WfMS itself. This restriction is crucial to ensure reactive behaviour. Verification methodologies for Workflow Specifications based on JECA rules are discussed in [31].

2.2.7 Temporal Logic for Workflow Modelling

Temporal Logic extends the classical propositional logic with operators to describe time constraints and temporal dependencies. The temporal operators refer to different periods of time or moments for instance $X \leftrightarrow$ *next time*, $F \leftrightarrow$ *in future* or $G \leftrightarrow$ *generally at any time* which are described thoroughly in [8].

[15] A JECA rule is disqualified if the justification evaluates to *true*.

Attie et al. [3] describe how to model control-flow dependencies with Temporal Logic constructs as global constraints for a Workflow Specification, using a database environment. An *activity scheduler*, which belongs to the WfMS, initiates the distribution of work items among external resources, if all global constraints are fulfilled. In case of a constraint violation, the activity cannot get executed and the execution request is handled at a later time [47]. Temporal Logic based constraints, in combination with a so-called *Model Checking* tool, can be applied for the verification of Workflow Specifications [14].

The main advantages of using Temporal Logic for workflow modelling are high and generalized expressiveness of the language constructs. However, Workflow Modelling based on Temporal Logic usually requires much time effort. The computational work that has to be performed by the WfMS is usually high [47].

2.3 The IEC 61499 Standard

The development of the *Industrial Standard IEC 61499*[16] has been initiated in order to provide a new architecture for the implementation of complex, distributed industrial process, measurement and control systems (IPMCSs) [28]. As basic construct of the IEC 61499, the *Function Block (FB)* is treated as a reusable (small) software component that represents "a piece of hardware" [28], [16]. FBs allow the encapsulation of functionality by means of algorithms and the corresponding input and output data. One of the main aims is to provide an *open architecture* to support the development of Distributed Control Systems, whereat "open" means that the architecture complies with the system demands listed below [9].

- **Portability**: Software tools from different vendors can interpret and use library elements and configuration data correctly, which are originally produced by other software tools.
- **Configurability**: Devices including their software components can be configured by different available development tools.
- **Interoperability**: Intelligent hardware devices originating from different vendors work together and can be exchanged easily (e.g., by replacing a broken device).

Due to its generic architecture, which is described by a set of descriptive models, the standard is applicable to PLC-Systems (Programmable Logic Controller), intelligent

[16]Development started in the early 1990, initiated by the Technical Committee 65 (TC65) as a group of the *International Electrotechnical Commission (IEC).*

hardware devices[17] or Fieldbus protocols.

Firstly, the FB concept and its characteristics will be explained. Secondly, a description of the models which are defined in the IEC 61499 Standard is given. The last sub-section presents current research topics.

2.3.1 The Function Block Concept

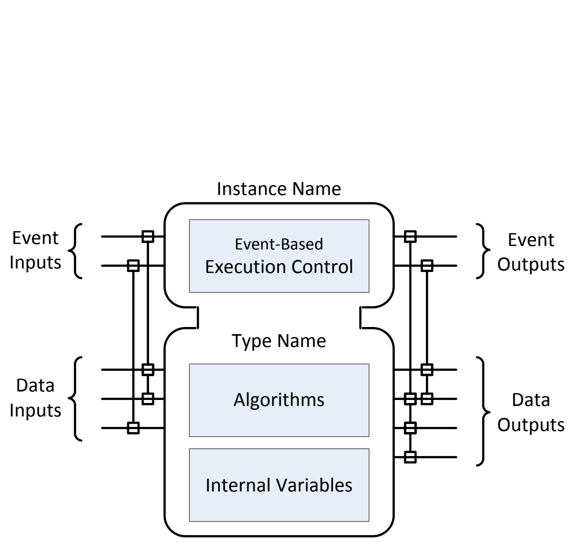
Part 1 of the IEC 61499 Standard focuses on the FB architecture as well as on general concepts for system modelling based on FBs [1]. A FB encapsulates algorithms and internal data which are hidden from the environment. Moreover, the execution of these algorithms can be triggered by events, which are routed to the FBs event inputs. These features characterize an *event-controlled* system architecture. Lewis [28] compares FBs with objects in object-oriented programming languages and concludes that FBs and objects share similar characteristics. Figure 2.3(a) shows the scheme of common FB as defined in [1], which is representative for every *FB-Type*[18] in IEC 61499. Every FB that is used in an *IEC 61499 Control Application* is of a certain *FB-Type* and has a unique *Instance Name*. *Event inputs* and *event outputs* are used to receive and send events, respectively. To enable data exchange between FBs, *data inputs* and *data outputs* are used, which are associated with corresponding events[19]. In the FBs bottom part *Algorithms* and *Internal variables* are stored and hidden within the block. The *Execution Control* is located in the top part and controls the execution of the predefined algorithms, if an appropriate input event is received [28]. To execute the defined algorithms, the FB has to be mapped to a *resource* which provides an *execution runtime environment*.

Figure 2.3(b) shows how the execution of a FB algorithm is managed by the scheduler of the responsible resource. Firstly, data inputs need to have *valid and steady values* ①. Secondly, if the corresponding input event is active ②, the scheduler is ordered by the FB's Execution Control ③ to invoke the algorithm ④. During execution, internal variables may be used to store intermediate results. The resulting data is written to the data outputs (stable values) ⑤. Afterwards, the scheduler is ordered ⑥ to finish execution by requesting the Execution Control ⑦ to send the corresponding output event ⑧. Note that theses phases must be performed in the

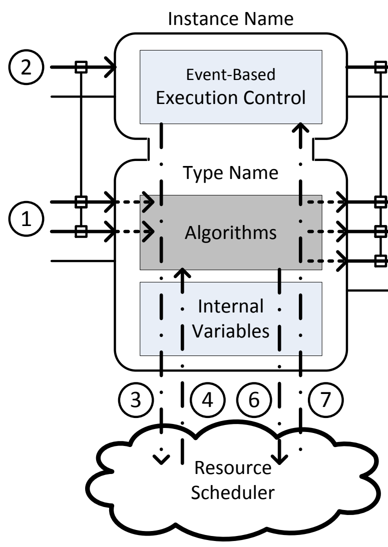
[17]Hardware devices like sensors or actors which contain autonomous control intelligence (e.g., controllers, advanced communication interfaces) are often referred to *Smart Devices*

[18]IEC 61499 defines three different "forms" of FB-Types, which will be explained later in this section

[19]This association is denoted as the *WITH construct* and visualized by the vertical lines, connecting event inputs/outputs with data inputs/outputs [28].



(a) A generic FB



(b) Algorithm execution steps

Figure 2.3: Definition scheme and execution steps for a generic Function Block according to IEC 61499 [28]

described order to ensure the correct execution and consistent data values [16]. The following paragraphs deal with the different Function Block types defined in the IEC 61499 Standard.

Basic Function Block

The Execution Control part of a *Basic Function Block* is determined by a state machine which is notated as *Execution Control Chart* (ECC) [1] [28]. An example of an Execution Control Chart is depicted in Figure 2.4. The *initial state* (i.e., START) corresponds to the entry point of the ECC. Besides, the ECC contains other *EC-States* and *EC-Transitions* which denote possible state transitions. An *EC-Action* is bound to an EC-State and refers to an algorithm or an output event or both, which should be called or fired, respectively, if the state is reached[20]. Every EC-Transition is labelled with a *condition* that can be an event input or any valid logical condition[21]. A transition originating from the *currently active state* is fired immediately if its condition is evaluated to *true*.

[20]Multiple EC-Actions can be bound to an EC-State which are executed consecutively, as shown for the state “OPERATION”.

[21]A logical condition may include internal variables, input data values and event inputs [1].

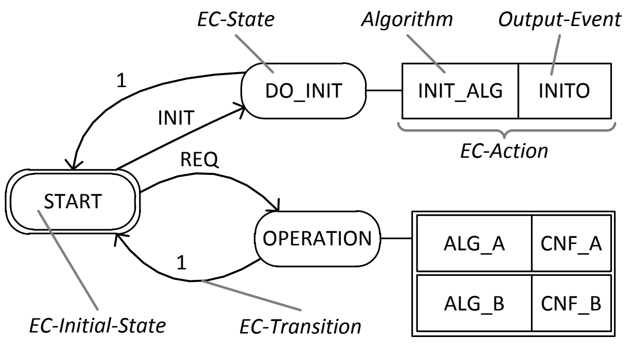
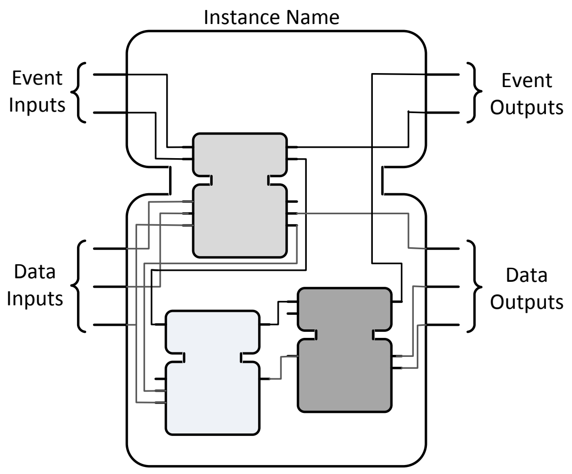


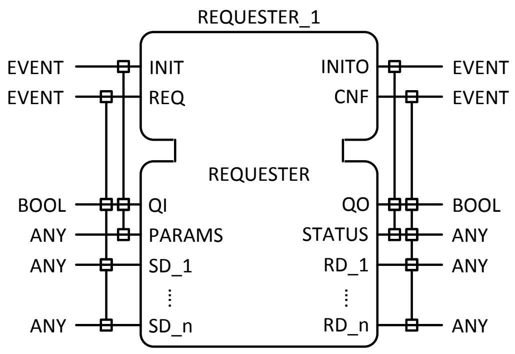
Figure 2.4: Description of an Execution Control Chart according to [1]

Composite Function Block

The behaviour of a *Composite Function Block* is determined by a number of FB instances which are interconnected by data and event connections, in a so-called *Composite FB-Network*. These "internal" FBs can be instances of every existing FB-Type (i.e., Basic, Composite or Service-Interface) [1], [16]. A graphical representation of a Composite FB is given in Figure 2.5(a). Based on the interface definition of the Composite FB, event/data inputs and outputs can be connected to inputs and outputs of Component FBs within the Composite FB-Network [1]. Because of their hierarchical structure, Composite FBs are useful to reduce the complexity in large FB-Networks and to force reusability of implemented functionalities [16]. A similar



(a) Composite FB with incorporated Composite FB-Network



(b) Requester Service-Interface FB [16]

Figure 2.5: Schemes of Composite and Service-Interface Function Blocks

construct is defined by so-called *Sub-Applications* which allow the encapsulation of a FB-Network analogously, with the difference that the internal FB-Network may be executed on distributed resources in contrast to a Composite FB [28].

Service-Interface Function Block

Data or event communication between distributed resources or hardware devices can be achieved by using *Service-Interface Function Blocks (SIFBs)*[22] [1], [16], [28]. Regarding the source, that initiates data/event transfer, the IEC 61499 Standard distinguishes between two situations [1]. Firstly, a transfer request can be triggered by a FB-Network (part) which is executed on a resource, thus naming the corresponding SIFB a *REQUESTER*. The second possibility refers to a *resource-initiated* action (e.g., data received or time-out triggered [16]), naming the SIFB a *RESPONDER*.

Figure 2.5(b) depicts a generic *Requester SIFB*, which allows to configure the communication interface *PARAMS* data input at initialization[23]. Data can be sent and received via the data inputs/outputs *SD_i* and *RD_i*, respectively. In Figure 2.6 the whole communication process for an application-initiated request is displayed, using a *Time-Sequence Diagram* [28], including the view from both involved resources. A *Time-Sequence Diagram* describes the sequence of events during the interaction process, where the time is increased from the top to the bottom. The tags "+", "-" in brackets, denote for event inputs that QI is set to *true* or *false*, respectively and for event outputs that an action has been successful/unsuccessful. A service is

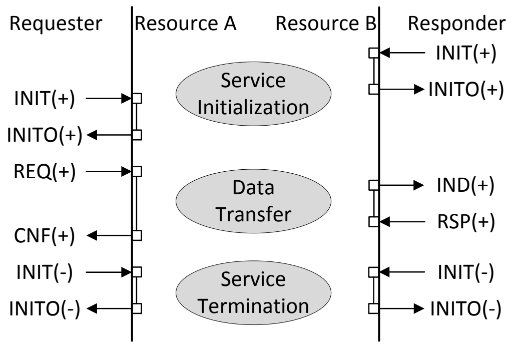


Figure 2.6: Time-Sequence Diagram describing Requester-Responder interaction

[22]SIFBs for instance represent interfaces to a communication network, or to hardware devices such as sensors and actuators.

[23]To initialize a SIFB by setting the INIT event, the boolean *qualifier QI* must be set to *true*. Service termination is achieved by setting QI to *false*.

requested using *REQ* and the response is indicated by *CNF*. In the next step, the Responder receives an indication event *IND*, processes the request, and answers to the Requester by emitting *RSP*.

Adapter Function Block

In order to increase reusability of defined FB-Types with similar behaviour, the concept of *Adapter Interfaces* has been introduced, which enables the sharing of common interfaces among those FBs [28]. This is realized by attaching an *Adapter Function Block* to an FB which should provide a reusable interface and is denoted as the *Adapter Provider*.

2.3.2 Descriptive Models for Distributed Control Systems

In order to meet the requirements of a modular and distributed architecture to support the development of *Distributed Control Systems*, the IEC 61499 Standard defines three generic models which are described in the following [1], [16]. Apart from these models, the *Management Model* describes management functions which are applied to FB-Networks that are executed by resources.

System Model

The *System Model* describes the top view of a distributed control system which consists of a number of devices, sharing a common communication network [1]. In contrast to centralised, monolithic system architectures the System Model describes the approach of distributing the overall control intelligence over several Smart Devices, such as sensors and actuators[24] [16].

Figure 2.7(a) shows the explained relationships. An *Application* can be considered as an aggregation of FBs of different types, together with the event and data connections that form a FB-Network. To make an Application distributed, single or groups of FBs can be assigned (or *mapped*) to different devices. The incorporated resources of a device are responsible for the FB execution.

Device Model

Devices (see Figure 2.7(b)) enclose several resources, which are capable of executing

[24]This means that applications can run either distributed over several devices or on single devices.

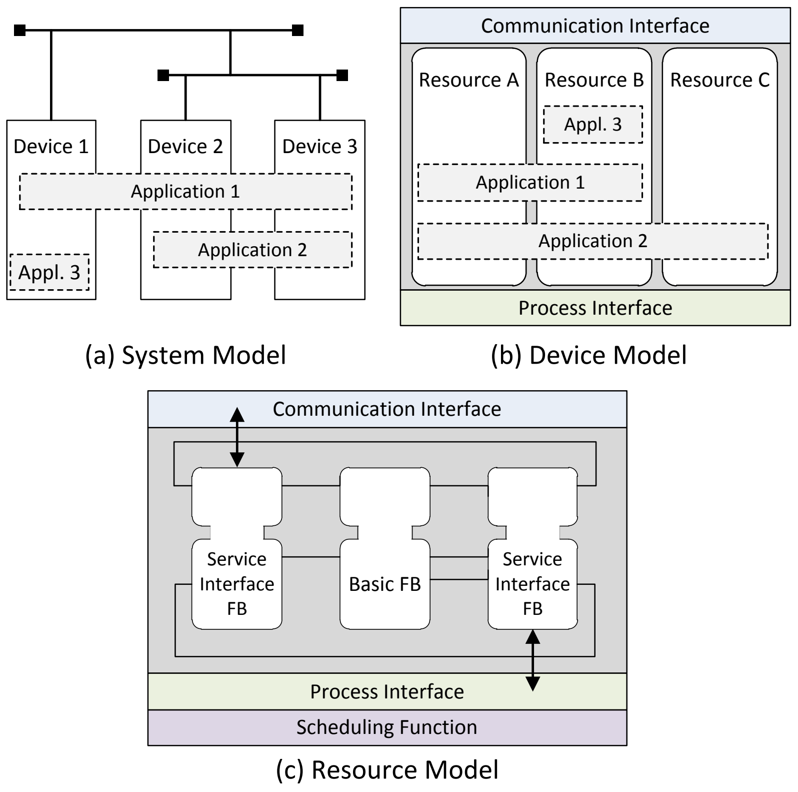


Figure 2.7: Descriptive Models defined in the Standard IEC 61499

FB-Networks (or fragments) independently of each other. Moreover, communication interfaces such as the *Process Interface* and the *Communication Interface* are provided. Where the *Process Interface* enables the resources to exchange data with the physical *process inputs* and *outputs*, the *Communication Interface* provides the means for data exchange within the communication network [28].

Resource Model

Resources are responsible for the execution of FB-Networks or network fragments and provide necessary services. These services include the routing of data and events through FB-Network fragments, as well as the *scheduling* function which controls the execution of FB functionality [16]). Moreover, SIFBs are used in order to map data and events between the FB-Network and the provided interfaces (Process and Communication Interface)[28]. Another important feature of resources is their support for independent operation of the other resources belonging to a device. Therefore, a resource can be configured, halted, resumed or deleted without influencing other resources [28]. Figure 2.7(c) gives an overview of a resource's characteristics.

2.3.3 Related Work

Recent research activities has focused on automation solutions using *high level components* which are built in a *hierarchical structure* out of lower levelled components which perform basic functionalities [34], [70], [24].

The European funded research project *MEDEIA* [34] faces the problem of interoperability in complex automated production plants, that arises due to different design and implementation methodologies for various automation solutions. A model-driven approach is used to represent a production plant hierarchically, consisting of so-called *Automation Components (ACs)*[25]. These can be considered as a combination of hard and software, and supervise lower levelled ACs. Machine vendors can still use their familiar design methods because the AC definition is transformed into a generic AC model. This generalized model allows the transformation to other known design models and further enables *automatic code generation* to support specific platforms [34].

In [70], methodologies are presented to structure automation components in a way, that higher levelled functionalities (i.e., supervising functions) coordinate sub-components across predefined interfaces. This approach to *component based* modelling makes use of *Adapter Interfaces* to reduce network complexity. The *Sub-Application* construct allows flexible[26], hierarchical grouping of functionalities [70].

A combination of IEC 61499 control applications and the features of the *ISA S88 Standard*[27] is proposed in [24], introducing design guidelines for "reconfigurable distributed *batch control* based on reusable software components". Based on the process description, a library of component FBs is generated. To achieve high flexibility, three special FBs that are called *Scheduler*, *Selector* and *Synchronizer* are used to trigger and synchronize execution of the desired functionalities according to the desired procedure. If reconfiguration of the process is necessary, only the procedure sent to the scheduler needs to be changed [24].

[25]An AC contains a formal description of its functionality and interface definitions for its interaction with other ACs.

[26]Sub-Applications support changes during several design stages on application level. Changes in Composite FBs would always result in a new FB-Type.

[27]The ISA S88 Standard is used to model batch processes hierarchically and draws a clear distinction between the process knowledge (i.e., procedures according to a recipe) and the hardware equipment that is used in the plant [6].

2.4 3D-Simulation Environments

This section provides a brief overview of simulation environments, especially *3D-Simulation Environments*. Firstly, the term simulation is explained and related characteristics are discussed. Secondly, some application fields for 3D-Simulation are briefly presented and an overview of available commercial 3D-Simulation tools is given. In the model-based engineering approach, that is proposed in this thesis, a 3D-Simulation tool is used in order to model the components, the distributed system (e.g., a robotic system) which shall be controlled, is composed of.

2.4.1 Simulation Fundamentals

Simulation in general is a method to analyse the behaviour of dynamic systems. Basis of every simulation is the derivation of a model which describes the system and its processes as close to reality as necessary [38], [4].

Computer Simulation is defined as the simulation of a system, with the support of a computer program, where the computer simulation environment is fed with the model data of the system [4]. To perform a concrete empirical experiment trough several simulation cycles, it is necessary to parametrize and vary the parameters of the model. The resulting outputs of the performed simulation cycles should provide new insights into the behaviour of the real system [38]. Common problems during simulation usually arise through limited *calculation capacity* and uncertainties in the simulation model[28].

2.4.2 3D Simulation

3D Simulation provides a very realistic graphical view of modelling objects[29], which are presented in the four-dimensional space time [38]. Since 3D Simulation environments are capable of displaying also *movements* of model components (e.g., robot joints, conveyor belts), they are suitable for simulating *logistics* and *factory production* processes.

[28]This limitation originates in the fact that models are only abstract representations of the real world. Their accuracy can be improved trough permanent model adjustments, according to the simulation needs [38].

[29]Since 3D Simulation tools often are tightly linked to CAD software, most model objects (e.g., work cells or robot systems) are provided as CAD models [2].

According to Bangsow [4], simulation analysis are performed in several implementation phases of a plant, for instance during its planning phase to estimate capacity, utilization and bottlenecks. Furthermore, during the realization phase for performance tests and in the operating phase for quality management and malfunction management. The use of 3D-Simulation environments has emerged to an indispensable practice in the production industry for factory planning, in order to cope with product varieties, small lot sizes, increasing quality requirements and market competition [4].

3D-Simulation Toolkits
Below, three commercial 3D-Simulation Toolkits shall be introduced briefly, which are currently available on the market. The *CATIA* toolchain of *Dassault Systèmes 3DS* [11] encompasses CAD modelling tools as well as engines for the calculation of model kinematics and NC programming. *Siemens* produces the product life cycle management software series *Tecnomatix Plant Simulation* [51]. The toolkit allows simulation, analysis, 3D visualization and optimization of full production or logistics processes. *3DCreate* of *VisualComponents* [64] enables the full 3D simulation of factory and work cell layouts, whereas the involved components can be imported from existing CAD data. In order to make the imported models moving, behaviour and configuration parameters (e.g., kinematics and programming scripts) can be added.

2.5 Automatic Code Generation

This section provides a basic introduction into the methods of *Automatic Code Generation.* After giving a definition of Automatic Code Generation, some general characteristics will be described which apply for several forms of Code Generation. In the second step, code generation approaches are generally treated which are based on *Model Driven Software Development.* Lastly, a selection of available code generation toolkits and frameworks are presented. Within the scope of this thesis, code generation frameworks to generate classes for the implementation of a graphical Workflow Modelling Editor are used. The Workflow Modelling Language is based on a modelling paradigm, that is determined by a data model.

2.5.1 Definition and General Characteristics

Computer programming where program code is generated through a mechanism, which does not require the interaction of a human programmer, is termed as *Automatic Programming*. A related methodology to *Automatic Programming* is denoted as *Generative Programming*, which improves the software engineering productivity through *"automated source code creation"* based on templates, generic models and classes, and code generation engines [33]. Herrington [19] and [63] identify some main benefits and characteristics of Generative Programming that are listed below.

- Generative Programming reduces *development time*, where at the same time *code consistency*, *quality* and *productivity* are improved.
- Furthermore, reusability of standard source code models is increased and the adaptation of existing code can be performed automatically.
- Code generation *rules* can be defined platform-independently, enabling the development of portable code generation functionalities.

According to [19], code generation can be done *passively*, where source code is generated once and the programmer is free to perform changes on it afterwards. *Active* code generation, describes a method where the generator can be run *multiple times* with changed configuration parameters in order to update the code.

2.5.2 Approaches to Code Generation

In general, Generative Programming Tools, especially *code generators*, can have large variations in their architecture [19]. The tools introduced in this section are based on *Model Driven Software Development (MDSD)* approaches which build on formal *model definitions*. These are used by a *Generator Software* to create executable source code, that is expressed in different programming languages [53]. A formal model is based on a formal language, which is used to describe its syntax and semantics [53]. Based on a given formal model, the generation of an expressive representation of the model for a specific *platform* is accomplished through a *Model Transformation*. For instance *Model-To-Code* transformations generate text (e.g., source code) out of a formal model, for a specific platform [33], [53].

An example approach using MDSD is *Model Driven Architecture (MDA)* which is standardized by the *Object Management Group (OMG)* [44]. The MDA approach uses the description of the functional details and specifications of an application, in form of a *Platform Independent Model (PIM)*, to develop the desired software application. In this context a full MDA Specification consists of a single PIM (defining the needs), several *Platform Specific Models (PSM)* and *interface definitions* which together define how the PIM is implemented on concrete platforms [44]. Summarizing, the basic idea of MDA is to strictly separate the functional description of a system from the implementation, how a system uses an associated platform, and to force reusability, portability and interoperability [44], [33]. In the following subsections a short overview of Generative Programming Toolkits and projects which are based on MDSD is provided.

2.5.3 Generic Modelling Environment

"The Generic Modelling Environment (GME) is a configurable toolkit for creating domain-specific modelling and program synthesis environments" [23]. In order to configure the tool properly, the definition of a domain-specific *Modelling Language*, which is based on a modelling paradigm, is needed. As stated in Section 2.5.2 and in [25], the modelling paradigm defines syntax and semantics of the formal language[30]. The domain-specific environment can be generated automatically, out of the definition of the modelling paradigm.

2.5.4 Eclipse Modelling Framework

The Eclipse Modelling Framework (EMF) project aims to provide simplified methodologies for the definition of structured data models [37]. Hereby, the developer should focus on the model description, whereas the EMF tools provide means for generating source code, for the classes represented by the model.

EMF consists of three main components, as described in [13]. Firstly, the *EMF Ecore* framework describes the *Meta-Model* which is used to express a data model with the aid of design tools. Additionally, serialization support for model objects and a functionality to emit signals upon *model object changes*[31] are provided. Secondly, *EMF-Edit* provides generic classes to generate *editors*, based on graphical property sheets, that can be used to create data objects of a model [13]. Lastly, the *EMF-Codegen* framework part allows the automatic generation of source code [13] and supports re-generation of existing code parts.

[30]Moreover, it defines how models may be created and how the model elements are related to each other [23].

[31]These events are called *notifications*, which are fired upon model changes [37].

2.6 Concluding remarks

This chapter dealt with an introduction into the current state of the art and covers the theoretical background, the outcome of this thesis bases on.

Section 2.1 to Section 2.2 discuss the basic concepts of Workflow Management Systems (WfMSs) and Workflow Modelling Languages. It can be concluded that *reactivity* is an important characteristic a WfMS should *have*, and a Workflow Modelling Language should *support* trough an appropriate definition of its language constructs. In Section 2.3 the Function Block Concept as well as descriptive models of the IEC 61499 Standard have been discussed. The IEC 61499 Standard supports modelling of distributed control applications and enables interoperability, configurability and portability. 3D-Simulation Environments have been treated in Section 2.4.2, suitable for *factory simulation*, and consequently for the simulation of distributed system components. Automatic Code Generation (see Section 2.5) allows effective software development by focusing on the development of a model description (which represents the application structure) and generating large source code blocks automatically.

The following chapters deal with the development of a *model driven engineering approach*, as described in Section 1.1. Based on the concepts of Workflow Management and code generation methodologies, a graphical Workflow Modelling Language, which supports reactive workflow execution, as well as a code generator is proposed. A 3D-Simulation Environment will be utilized to model the components of the distributed system and the service functionalities, provided by these components. Finally, a code generation method is presented which generates a FB application, according to IEC 61499, based on a Workflow Specification. The *overall system architecture* and the requirements for this engineering approach are presented in the next chapter.

3 Overall System Design

In Section 1.1 the major aim of this work, namely the development of a method to enable simplified programming of a control application for a robotic system, has been motivated. This programming method builds on the design of a process description (i.e., a Workflow Specification), that is expressed using a Workflow Modelling Language. The second step is to generate an IEC 61499 compliant Function Block Control Application, based on the determined Workflow Specification.

A *system architecture* is proposed which meets the requirements for the implementation of the engineering approach described in Section 1.1. Firstly, the components of this architecture are identified by considering the steps of the desired engineering process. Hereby, the engineering toolchain, which is intended to support the user, is explained. Secondly, the engineering steps are explored in detail and the requirements for the components of the system architecture are identified.

3.1 Engineering Process

In this section, an engineering process to enable simplified programming of a control behaviour and henceforth the generation of an IEC 61499 compliant control application is presented. Based on this engineering process, the main components of the underlying system architecture are identified and explained. Afterwards, the subsequent sections deal with the identification of the requirements for the components of the system architecture.

3.1.1 Prerequisites for the Engineering Process

As described in Section 1.1, it is assumed that the system (e.g., a robotic system), for which a supervisory control application has to be programmed, is composed of a number of functional components. These components need to be identified by the engineer in advance, including the *services* they provide to the system. The description of a component's services shall be henceforth called the *behaviour* of the component, which is expressed by declaring *Service-Functions (SFs)* and their corresponding function *parameters*. The *behaviour* information of a functional component can be already provided by the component vendor, and therefore does not always need to be identified by the engineer. Besides the component identification, the process, which should be performed by the functional components, has to be split up into distinct process steps. This is because the process needs to be modelled by means of a Workflow Specification, for programming a supervisory control application.

3.1.2 3D-Model Design of the Robotic System

In the initial phase, the robotic system has to be modelled in a 3D-Simulation environment. This requires that the system engineer already has a concrete concept how the system should look like and of which functional separable components it is composed of. Hereby, it is assumed that the *appearance* of the functional components is represented by either existing *CAD data*[1] or unavailable parts which need to be constructed.

After the system components have been modelled in the 3D-Simulation Environment, they are extended with the available *behaviour information*, expressed by the SFs. The modelled components combined with the *behaviour information* are henceforth denoted as *Service-Components (SCs)*. Figure 3.1 gives an example of how a *Bin Picking Work Cell* can be modelled, using a *6-DOF robot*[2] combined with a *laser scanner object recognition* system. The modelled SCs are stored in the *Component Library* of the 3D-Simulation environment. An integrated *Behaviour Editor* is used to edit the behaviour information of the SCs.

3.1.3 Design of the Workflow Specification

The behaviour of the *supervisory control application* which coordinates the SCs in order to perform the given process is determined by a Workflow Specification. To enable simplified programming of control behaviour, a graphical *Workflow Editor* is needed that builds on a Workflow Modelling Language. Moreover, the Workflow

[1]Thinking of components for those CAD data has been constructed previously, and can be imported into the 3D-Simulation environment. CAD... Computer Aided Design.

[2]DOF - Degrees Of Freedom.

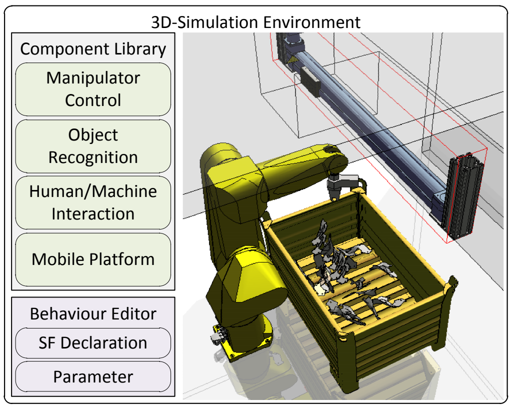


Figure 3.1: Modelling the robotic system consisting of Service-Components

Editor connects to the 3D-Simulation Environment, loads the behaviour attributes of all SCs and fills a palette of Service Activities which are utilized to construct a Workflow Specification. In order to manage error cases, basic exception handling strategies can be applied.

Figure 3.2 shows a possible layout of a graphical Workflow Editor. The left part of the image shows the *activity palette* that contains a list of available activities, corresponding to the modelled SCs. Moreover, the image depicts a Workflow Specification example with activities and connections, that define the *execution sequence.*

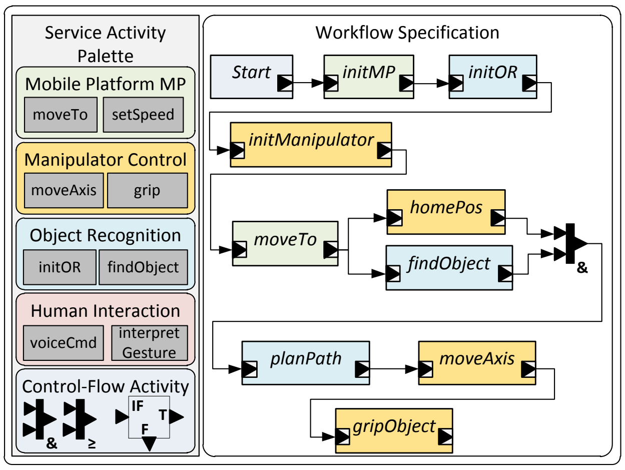


Figure 3.2: Schematic view of a Workflow Editor

3.1.4 Automatic Generation of the Control Application

With the Workflow Specification as the input, a *code generator* is used to generate an IEC 61499 control application that is represented by a Function Block-Network (FB-Network). The goal is to automatically create a FB-Network which directly represents the control behaviour that is defined by the modelled Workflow Specification.

3.1.5 Simulation and Execution of the Control Application

Once the control application has been generated, an Engineering Tool for FB-Applications can be used to map and download the supervisory control application to distributed resources. An IEC 61499 compliant *runtime environment* is run on the device, needed to execute the FB-Network. In order to test the overall behaviour and further improve the Workflow Specification and the SC models, the FB runtime environment can be coupled with the 3D-Simulation Environment. In that way it is possible to treat the modelled SCs as the real hardware components by simulating their behaviour. Using the simulation tool allows the engineer to iteratively improve and check the interaction between SCs before running the application on real hardware.

3.2 Resulting System Architecture

The steps of the Engineering Process described above are summarized in Figure 3.3. With reference to the described Engineering Process, it is determined that the system architecture of the engineering approach is composed of the components listed below:

- An available *3D-Simulation Environment* is used in order to model the Service Components of the distributed system, which needs to be controlled.
- A *graphical Workflow Editor* needs to be implemented to enable the design of Workflow Specifications.
- To generate an executable control application according to IEC 61499, a *code generator* needs to be developed.
- The execution of the control application is performed by an IEC 61499 compliant *Runtime Environment* which is supported by the chosen target platform.

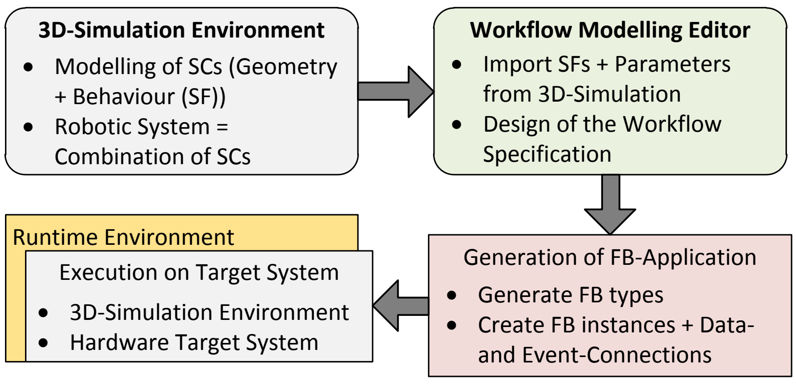

3D-Simulation Environment
Modelling of SCs (Geometry + Behaviour (SF))
Robotic System = Combination of SCs
Workflow Modelling Editor
Import SFs + Parameters from 3D-Simulation
Design of the Workflow Specification
Runtime Environment
Execution on Target System
3D-Simulation Environment
Hardware Target System
Generation of FB-Application
Generate FB types
Create FB instances + Data- and Event-Connections

Figure 3.3: Overview of the Engineering Workflow

- An *Engineering Environment* for the development of Function-Block Control Applications is used to initiate the execution of the generated FB-Network by the Runtime Environment.

The next sections of this chapter, provide a more detailed overview of the requirements that need to be met by the components of the system architecture.

3.3 Modelling Component-Behaviour

As mentioned above, the modelled robotic system is composed by a set of modelled Service-Components (SCs). A SC encompasses the geometrical representation (e.g., CAD data) as well as the component's functional behaviour[3] expressed by Service-Functions (SFs). Analogous to the notion of functions in high-level programming languages, a SF may require function parameters to be passed when the service is invoked. Moreover, the SF may return a set of *return values*.

By using only SF declarations, which can be considered as an interface to the SC, it is ensured that the behaviour of the SC itself can be implemented platform-independently. A possible composition of SCs to a robotic system and a schematic example of how SFs can be declared is shown in Figure 3.4. These Service-Components are intended to interact with each other. For example, the SC *Mobile Platform (MP)*, which is responsible for moving the whole robot construction, needs to interact with

[3]This behaviour description should not be mixed up with behaviour that corresponds to a simulation model object, for instance calculated robot kinematics, that define how robot axes can move.

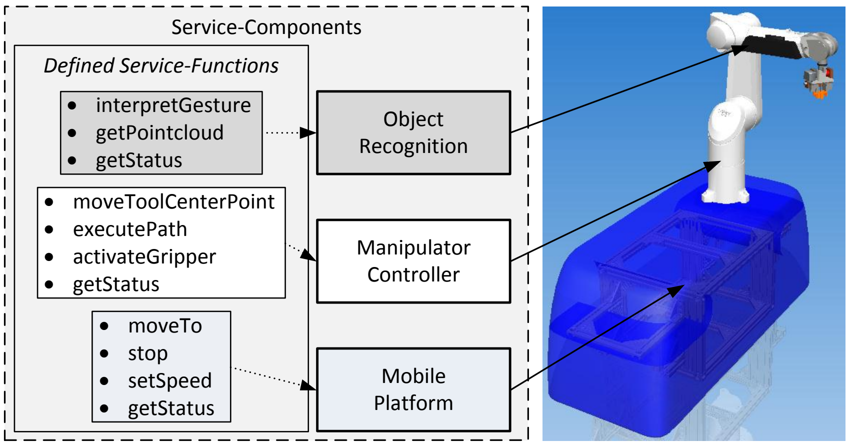


Figure 3.4: Service-Components and Service-Functions of a modelled robotic system

the SC *Object Recognition (OR)*. In the following the case of an obstacle being detected by the Object Recognition system, while the robot is moving, is considered. The Mobile Platform is notified to stop and a new motion path needs to be calculated to avoid the obstacle. An OR Service-Component could be realized as a *3D image processing system* that evaluates pictures taken by a stereo camera system.

Figure 3.4 depicts a scheme how Service-Functions can be declared for a SC. For example, the Mobile Platform could offer the SFs *moveTo*, which initiates a movement of the robotic system to the specified position, *stop* which causes the cancellation of a movement, *setSpeed* to set the velocity and *getStatus* which returns a value indicating the current operating status of the SC. At this point it is worth mentioning that the means of communication between SCs and the supervisory control application have to be well defined. This problem is treated in detail within the next section.

3.4 Requirements for the Workflow Editor

In this section, the basic requirements for the Workflow Editor and the Workflow Modelling Language are discussed. As stated in Section 2.1.1, the development of a Workflow Specification builds on the syntax and semantics of a Workflow Modelling Language which expresses the workflow. Generally, the work that has been done in the scope of this thesis regarding Workflow Management focused on designing a *graphical* Workflow Modelling Language. This is because in the author's opinion, a Graphical User Interface (GUI) provides more intuitive ways of designing for the end user, instead of textual languages.

3.4.1 Available Workflow Activities

A Workflow consists of a set of activities which are linked to each other, building a defined execution sequence. To enable Workflow Modelling based on the available SCs, the *first* requirement on the Workflow Editor is to provide a functionality which allows to load the SF declarations (of all SCs) from the 3D-Simulation environment. Then the SF can be considered to be equivalent to an activity of a Workflow Specification. As a possible realization, the loaded SFs can be used to fill an *activity palette* of the graphical *Workflow Editor*. This *activity palette* contains a list of the defined activity types, which can be used within the Workflow Modelling Editor in order to create a Workflow Specification. Henceforth, this type of activities are denoted as *Service Activities (SA).*

Apart from SAs, a set of *Control-Flow Activities (CFA)* is required to provide semantic means for branching of the control-flow, based on logical decisions, as well as merging different control-flow branches according to [61].

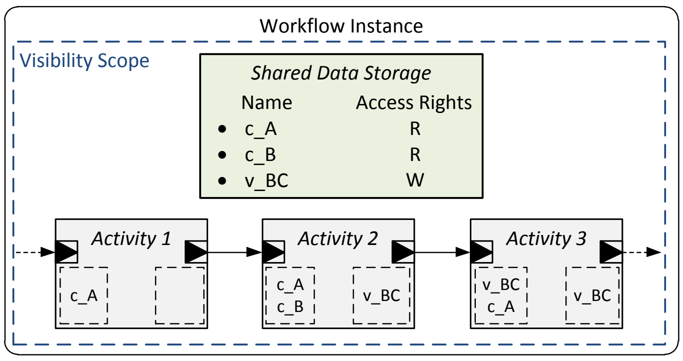
3.4.2 Management of Data Objects

According to Workflow Data Patterns, explained in [58], the definition of rules, which describe how data objects are represented in Workflow Modelling Languages and how they are transferred between activities, is of great importance. Furthermore, the *visibility scope* for data objects within a Workflow Specification has to be determined once the object is being created [58].

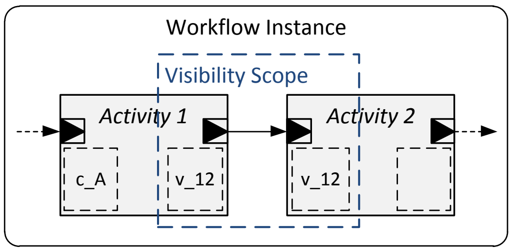
Regarding the visibility scope and taking into account that finally an IEC 61499 Control Application shall be generated based on a Workflow Specification, three possible visibility scopes have been identified which are listed as follows and shown in Figure 3.5. Note that in Figure 3.5 constants are identified with a *"c"* prefix and variables, which allow read and write access, are written with the *"v"* prefix.

- **Workflow Instance**
  Visibility scopes encompassing a whole Workflow Instance[4] are suitable especially for *constants* that are generally accessible. Additionally, *shared* data

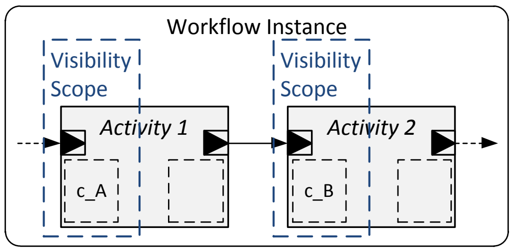
[4]Also referred as Workflow Case in [58] and [59].



(a) Workflow Instance



(b) Local - Subsequent activities



(c) Local - Constant data objects

Figure 3.5: Possible data visibility scopes in Workflow Instances

variables can be used for write access within the Workflow Instance. However, this case requires functionalities in order to handle concurrent access to a variable, especially if two in parallel executing branches affect the same data objects (see Figure 3.5(a)).

- **Local** - *within the scope of subsequently connected activities*
  This visibility scope applies for activities which are connected directly to each other. Note that these data objects are commonly intended to be passed directly to the next activity inputs (see Figure 3.5(b)).
- **Local** - *as constants defined uniquely for a certain activity*
  Constants can be defined for certain activity data inputs and consequently they are only visible in the scope of one workflow activity (see Figure 3.5(c)).

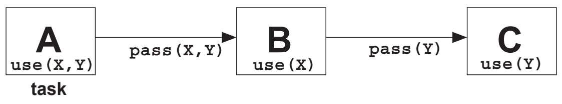
Data objects have a determined *data type*, that needs to conform with the data types which are supported by the WfMS, especially by the *Workflow Enactment Service* which is responsible for the execution of Workflow Instances.

After establishing rules for the visibility of data objects, there is interest in how data is exchanged between activities in a Workflow Instance. A set of *Data Inter-*

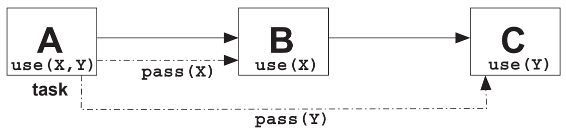
action patterns, explaining three approaches to exchange data, is described in [58]. The first method uses *combined* data and control-flow channels between activities, whereas the second possibility is based on distinctive data and control-flow channels. A completely different approach is to use a shared data storage instead of passing data. Aalst et al. [58] localize the main disadvantage of the first method by the fact, that the whole set of data objects, needed in the executing branch, has to be passed through all activities, independent from whether all activities access all data objects or not. Figure 3.6 shows a schematic representation of the approaches for data interaction[5]. Nevertheless, combined data and control-flow channels will keep the Workflow Specification *simple to read and understand.* Modelling data channels additionally in the Workflow Specification would result in an overloaded graphical representation. In that case it would be easier to create the supervisory control application directly, using an IEC 61499 compliant engineering development environment. Data exchange based on shared memory additionally requires mechanisms to manage concurrent data access.

3.4.3 Reactive Workflow Execution

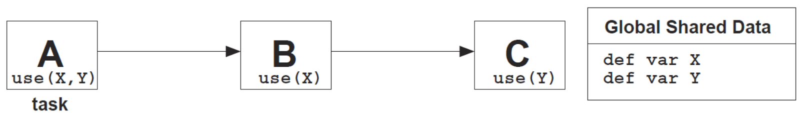
According to [14], a reactive WfMS should never process a work item but should allocate it to an external resource and only trigger its execution. By introducing



(a) Combined data and control-flow channels



(b) Distinct data and control-flow channels



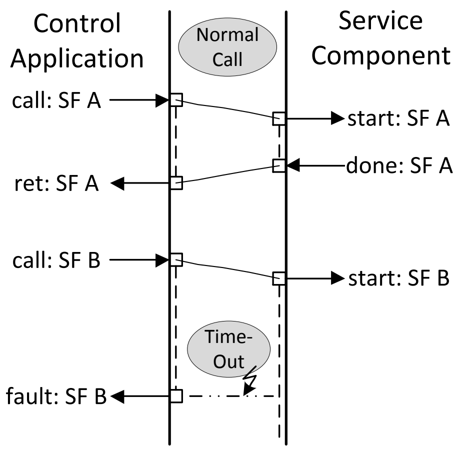
(c) Shared data storage - no data passing

Figure 3.6: Approaches to exchange data between activities [58]

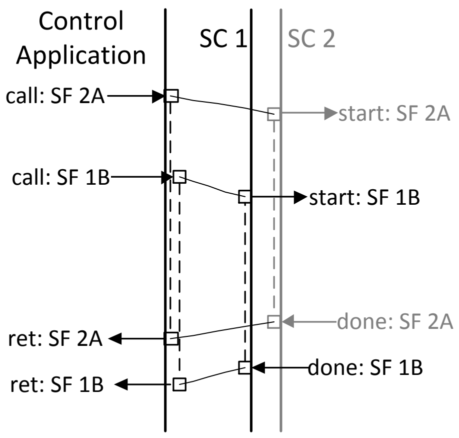
[5]To avoid confusion: Note that activities are denoted as *tasks* in Figure 3.6.

Service-Components this central requirement is already satisfied, because the WfMS, which executes a Workflow Instance, only invokes the desired Service-Functions. Furthermore, the WfMS waits for the SC to respond and the execution is continued afterwards by calling subsequent services.

What still needs to be defined is how the communication between the executing *supervisory control application* and the SCs, that provide SFs for the workflow execution, is arranged. It can be distinguished between a *synchronous communication* solution and an *asynchronous communication* solution. In case of synchronous communication, the execution of a control-flow branch is paused after the invocation of a SF, until a response of the SC is received. The asynchronous method allows a SF provided by a different SC to be called, while the previously called SF is still being processed. Figure 3.7 depicts the Time-Sequence Diagrams for synchronous and asynchronous communication. To handle the case of a SC which does not respond to the control application, a configurable *time-out* error needs to be introduced. After a service request has been submitted and if no response is detected within a given time span, the emission of a *fault* event is triggered, as shown in Figure 3.7(a). The supervisory control application can again react on the reception of this event by means of an exception handling strategy. In Figure 3.7(b) a SF, which belongs to a different SC, is called while the previous request (i.e., *SF 2A*) is in progress. Note that in case of a call dependency of the subsequent SF (i.e., *SF 1B*) the use of the asynchronous communication method does not result in a shortened execution time. If the execution of a subsequent SF depends on the processing result of the preceding SF, concurrent execution is impossible.



(a) Synchronous Communication



(b) Asynchronous Communication

Figure 3.7: Communication methods for the interaction with Service-Components

Another possible solution can be described almost analogously to *synchronous communication* with the difference that the SC is not responsible for sending an answer. Instead, the current status of the SC and the produced output of the service invocation needs to be polled[6] by the control application itself. With reference to the characteristics of a reactive WfMS, the synchronous communication method fulfils the requirements described in [14].

3.4.4 Basic Exception Handling

In case of a *Service Activity (SA)* that is processed erroneously, at least basic exception handling concepts need to be implemented in order to make the WfMS capable of reacting to such execution failures. A set of different, applicable exception handling strategies is introduced in [60], [31] and [52].

As stated in Section 2.2.2, exception handling functionalities can be integrated into the WfMS either *embedded* or *autonomously*. Since the occurring context of the exception is clearly defined when using an embedded solution, handling strategies are realized by executing an alternative execution branch of the Workflow Specification. In that case, no special language constructs are required for the Worfklow Modelling Language because common activities can be used to determine a handling strategy. Contrary to an embedded solution, autonomous exception handling strategies require to be determined independently of the Workflow Specficiation and the occurring context needs to be known exactly. Autonomous handling strategies can be defined in separated handling editors using a set of predefined exception handling primitives. These primitives do not necessarily represent activities but affect the Workflow Enactment Service directly. Approaches to autonomous exception handling are explained in [57] and [52].

Besides exceptional events which are in general thrown as a result of a failure and therefore require to be handled, it is demanded from the system to react on events sent from a defined source, which can be emitted at any time during the execution of the workflow. To comply with this requirement, special *event reception* activities need to be introduced, which start a *new control-flow*, if an event from a registered source is received. These events can be used to trigger the execution of a certain workflow branch.

[6]Polling denotes a query functionality for data values.

3.4.5 Defined Start- and End-Points

According to the definition of a *workflow* given by Jablonski et al. [25], a uniquely defined *start-node* and one *end-node* are required in a Workflow Specification. The start-node determines the origin of the control-flow and can be represented for instance by an activity, which triggers the execution of the workflow.

With reference to start-nodes in UML Activity Diagrams [50], State-Machine Diagrams [50] or Extended Workflow Nets [57], a start-node does not include the execution of an activity that is associated with it. It just serves as a marker for the beginning of the workflow. In hierarchical Workflow Specifications, *sub-workflows* also contain a start- and an end-node. The end-node marks the end-point of a Workflow Specification where the control-flow terminates. If the end-node is reached in a sub-workflow, the control-flow is passed back to the Workflow Instance in which the sub-workflow has been called. Generally, if the control-flow is split in the specification, all control-flows have to join in the end-node. However, the UML Activity Diagrams and State-Machine Diagrams [43] accept more than one end-node, and the execution of a Workflow Instance is finished if one of the end-nodes is reached.

However, in some cases the end-node of a Workflow Specification can be considered to be the last activity, which is executed within the worfklow instance. Note, that this rule cannot be applied to sub-workflows in general, since the control-flow needs to be passed back to the calling sub-workflow level after a defined end-point of the executed sub-workflow has been reached.

3.4.6 Syntactical Check of the Workflow Specification

To enable the correct generation of a supervisory control application, syntactical correctness of the Workflow Specification is crucial. Therefore, the Workflow Specification needs to be checked periodically, based on a set of integrity rules. This test checks the basic structural elements a Workflow Specification needs, including the *start-* and optional *end*-points, valid connections between activities, no loosely activities and a few more checks that will be explained later in this thesis. The *data type integrity* should be checked by the graphical *Workflow Editor* itself, every time when a new connection between activities is established.

3.5 Requirements for the Code Generator

Based on the design of a Workflow Specification and the SCs' behaviour descriptions, the next step of the presented approach is to generate an IEC 61499 control application. The control application coordinates the Service-Components of the modelled robotic system, to perform the predefined process. A *code generator* which produces a FB-Network on request, which is ready to be executed by an IEC 61499 compliant runtime environment, needs to be implemented. The requirements needed for such a code generator are presented in this section.

3.5.1 Generation of Needed FB-Types

The generation of an IEC 61499 control application out of the Workflow Specification requires the generation of FB-Types, representing the modelled workflow activities, in advance. Based on this application specific type library, a FB-Network can be generated by creating FB instances and connections between them.

Service-Components have been introduced to enable reactive workflow execution, by invoking their provided Service-Functions once the corresponding activities are reached by the control-flow. As already stated, to allow communication between the control application and an external Service-Component, a communication method has to be determined. The communication is then controlled by *interfaces* which need to be realized by generated FB-Types that connect to the affected Service-Components. Consequently, these *interfaces* have to manage several Service-Function calls, which are invoked through the workflow execution.

Apart from the interface FB-Types, that are used to interact with SCs, a Workflow Specification contains instances of different types of Control-Flow Activities (CFA). Since different CFA instances generally are differently parametrized, one representative FB-Type has to be generated for every CFA instance. The generation of representative FB-Types for workflow activities is treated in detail in the next chapter. After generating the necessary FB-Types, the supervisory control application can be generated in the next generation step.

3.5.2 Generation of the FB Control Application

A generator functionality to traverse the Workflow Specification is required in order to generate the FB control application. For all modelled workflow activities, instances of the generated, representative FB-Types need to be created. As a main requirement, an effective method to directly transform so-called *activity connections* (i.e., connections between workflow activities) into corresponding event and data connections, according to the IEC 61499 Standard, needs to be developed. Basically, this means that activity inputs and outputs have to be associated with events and *Activity Parameters* are associated with data inputs and outputs.

After generating the supervisory control application, by creating FB instances and event and data connections according to the Workflow Specification, an additional FB-Network fragment needs to be generated to enable the initialization of the FB-Network. Especially, FBs representing interfaces to Service-Components need to get initialized to operate properly. Moreover, the initialization procedure needs to be triggered, either upon the hardware resource has been started or through an external signal (e.g., trough a closed switch).

The generated FB-Network has to be executable by an IEC 61499 compliant runtime environment, which runs on a hardware resource. To enable the iterative improvement of a Workflow Specification, the supervisory control application should be able to operate in combination with the 3D-Simulation environment. The test results of the simulation can then be used for further development of the modelled robotic system and the Workflow Specification.

3.6 Concluding Remarks

In this chapter, an *engineering process* has been proposed which enables simplified programming of a supervisory control application. This approach is based on the modelling of the desired robotic system and its components, and the specification of the workflow which describes how the considered system should behave.

With reference to the described *engineering process*, a *system architecture* has been proposed. A *graphical Workflow Editor* as well as a *code generator* need to be developed and implemented. The remaining supporting tools are already available and can be applied directly (i.e., 3D-Simulation Environment and the Engineering Environment for distributed FB-Applications).

In the first step, a method to express *behaviour information* of the components, which belong to the robotic system, is introduced. The subsequent sections deal with the requirements for the development of a suitable *Workflow Modelling Language* and the *code generator* for the generation of a control application.

One of the main requirements for the Workflow Modelling Language is to enable *reactive Workflow Modelling.* This feature is realized by the introduction of Service-Components and their associated Service-Functions. Moreover, treating *data objects* in Workflow Specifications needs to be clearly defined, including data representation, transport, and visibility scopes. The Workflow Editor allows to design a Workflow Specification which is expressed by the developed Workflow Modelling Language. Hereby, the declared Service-Functions need to be used and therefore, the Workflow Editor has to provide a functionality to load the determined *component behaviour* to make them available as activities in the *activity palette.*

For the generation of the supervisory control application, a *code generator* needs to be implemented which satisfies the following main requirements. The automatic generation of the control application, according to IEC 61499, needs to be done in two steps. At first, FB-Types which represent the modelled activities of the Workflow Specification need to be generated. Based on the generated FB-Type group, the supervisory control application has to be generated, by traversing the Workflow Specification, and by creating the necessary FB instances and the corresponding data and event connections.

In the next chapter, a *graphical Workflow Modelling* Language is proposed to meet the requirements, for simplified programming of a supervisory control behaviour based on Workflow Modelling.

4 A Graphical Workflow Modelling Language

This chapter focuses on the design and implementation of a graphical Workflow Modelling Language, according to the identified requirements. A major requirement for the modelling paradigm, which consequently defines syntax and semantic of the Workflow Modelling Language, is to support the *reactive* execution of Workflow Instances.

The development process of the Workflow Language Concepts and the Language Elements that are explained in the following sections, was influenced by a thorough survey of the characteristics, advantages and disadvantages of the Modelling Languages which are introduced in Section 2.2. The first section deals with an explanation of the activity types including their characteristics and behaviour. Moreover, a description format for Service Component behaviour is introduced, which is necessary to load this component information into the Workflow Specification Editor. Afterwards, in the second section, extended concepts of the Workflow Modelling Language are discussed. A Workflow Specification example is treated and the possibilities of *Exception Handling* are shown. Furthermore, structuring methods for Workflow Specifications are discussed.

4.1 Language Elements

In this section, an overview of the language elements is presented. Firstly, Workflow Activities are introduced and the distinct types of Service Activities and Control-Flow Activities are discussed. The graphical notation which is used for the activity scheme images is similar to the representation used in the example chapter of [37]. Section 4.1.3 deals with the execution sequence and furthermore describes how data objects are treated in the Workflow Specification. Additionally, a solution to express the combination of control-flow and data channels by means of a single connection between activities is explained. Along with the description of the activity types, a set of construction rules is defined, which is of great importance in order to design correct Workflow Specifications. The generation of a supervisory control application can only be performed successfully if the Workflow Specification is modelled correctly.

4.1.1 Service Activities

As described in Section 3.4.1, *Service Activities (SA)* directly represent Service Functions (SF) within a Workflow Specification. Figure 4.1 depicts the scheme of a SA, having input and output *Activity Interfaces* and lists of *Activity Parameters (AP)*, that are written in squared brackets and associated with the corresponding Activity Interfaces. APs can be of different data types, which are supported by the WfMS. Service Activities in general do not need to have any input APs but require to have one output AP at least which serves as a *status value* that can be used to return *error codes*.

Once the control-flow reaches the Activity Interface “Input”, the corresponding Service-Function is invoked by the WfMS. Moreover, the corresponding APs are considered valid at this moment and passed to the SC. According to the current implementation of this Workflow Modelling Language, the communication between the service requester and the SC is managed synchronously. This means that the control-flow of the executing workflow branch is suspended until a response is received or a time-out error occurs.

If the service request could be processed correctly, the control-flow and the resulting output APs are forwarded to the Activity Interface “Output”. In case of errors, occurring during the service execution including *Connection Failures* and normal *SF errors*, the control-flow is passed to the Activity Interface “Fault”. By following the rule of using the first output AP as *status value*, the workflow designer can check this AP and use it to choose an appropriate *exception handling branch*. The current implementation of this Workflow Modelling Language uses the “Fault” Activity Interface to enable *embedded exception handling*. Note that the values of the

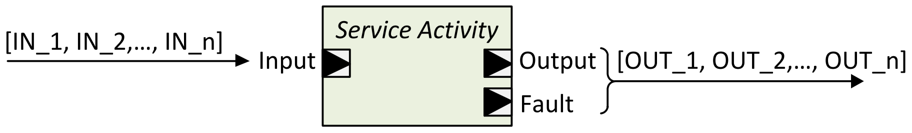


Figure 4.1: Scheme of a Service Activity with parameter inputs and outputs

output APs, in case of a processing error, need to be defined within the Service Component. In case of connection failures, the status value is changed accordingly. Table 4.1 shows reserved error numbers. User defined error codes can be utilized to express certain reasons for the failed execution of a SF.

As stated in Section 3.4.1, the behaviour of Service-Components which is determined by SF declarations is loaded into the graphical Workflow Specification Editor from the 3D-Simulation Environment. Service-Components are defined by its geometry and behaviour description but we have not yet specified the *format* that determines how behaviour information is expressed.

Behaviour Description for Service-Components

Behaviour of a SC is expressed by Service Functions and their function *parameters*. This behaviour information has to be added as an *attribute* to the corresponding 3D-model component in the 3D-Simulation Environment. In this approach the behaviour information is packed into an XML format. The XML-Schema [1] behind the behaviour description can be derived from the relations between Service-Component, Service-Functions, and parameters, which are displayed in Figure 4.2.

SCs are identified by a unique *name* and can have additional attributes, for instance a dynamic parameter to set the component's *IP address* which is utilized to establish a TCP (Transmission Control Protocol) connection, and need to provide at least one SF. A SF is labelled with a unique *name* in the behaviour description of the SC. This SF name has to be equal with the *function name* that is defined in the specific implementation of the SC, and requires at least one output *parameter*. Note that by requiring at least one parameter the recommendation, stated in Section 4.1.1, to reserve one output parameter for the status of the SC, is fulfilled. Parameters are *named* uniquely and are of a certain *data type*. The boolean flag *inputParam* denotes whether the parameter is defined as input parameter (*true*) or as output parameter (*false*).

Value	**Explanation**
0	No error - Successful execution
1	Connection error - Broken Client/Server connection
2	Time-Out error - No response from the SC
≥ 3	User defined errors

Table 4.1: Possible error codes returned by Service-Components

[1] An XML-Schema determines how an XML document has to be structured.

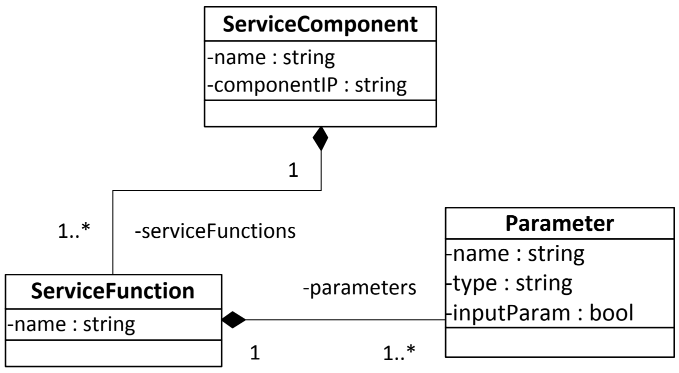


Figure 4.2: Relations between Service-Components, Service-Functions, and their parameters

For example, the XML behaviour description for a *mobile platform* Service-Component is expressed as shown below. The data type *ARRAY_OF_POINT_3F* is a complex data type.

```
<?xml version="1.0" encoding="UTF-8"?>
<Servicefunction:ServiceComponent componentIPAddress="192.186.0.1"
name="MobilePlatform">
  <servicefunction name="moveTo">
    <parameter name="pointX" type="REAL"/>
    <parameter name="pointY" type="REAL"/>
    <parameter name="pointZ" type="REAL"/>
    <parameter name="STATUS" type="INT" inputParameter="false"/>
  </servicefunction>
  <servicefunction name="setSpeed">
    <parameter name="velocity" type="REAL"/>
    <parameter name="STATUS" type="INT" inputParameter="false"/>
  </servicefunction>
  <servicefunction name="stop">
    <parameter name="STATUS" type="INT" inputParameter="false"/>
  </servicefunction>
  <servicefunction name="getStatus">
    <parameter name="STATUS" type="INT" inputParameter="false"/>
    <parameter name="Position" type="ARRAY_OF_POINT_3F"
    inputParameter="false"/>
  </servicefunction>
</Servicefunction:ServiceComponent>
```

The sample behaviour description contains four SF definitions, namely *moveTo*, *setSpeed*, *stop* and *getStatus* according to the schematic SC model in Figure 3.4. Component behaviour representations like this are added as an attribute to each Service-Component that needs to be modelled.

4.1.2 Control-Flow Activities

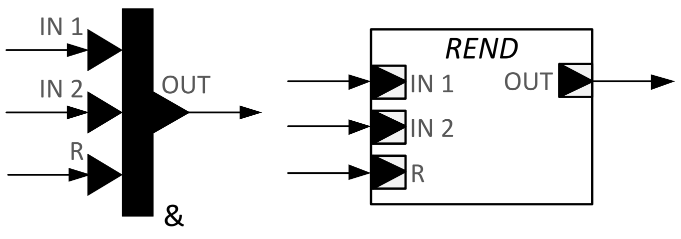
Control-Flow Activities (CFAs) enable the workflow designer to change the execution sequence of Service Activities by *branching* or *merging* control-flows. In contrast to Service Activities, CFAs are never bound to a certain component of the modelled robotic system but are defined as fixed language constructs of the Workflow Modelling Language. Consequently, Control-Flow Activities are commonly executed by the WfMS itself.

Section 2.2.2 introduces a set of Control-Flow Patterns which are commonly implemented among several Workflow Modelling Languages (see e.g., [57], [68], [43], [36], [18], or [14]). An extensive survey of implemented *control-flow routing constructs* in various Workflow Modelling Languages has been performed. Based on this literature research, a collection of eight essential control-flow constructs are selected and implemented as Control-Flow Activities.

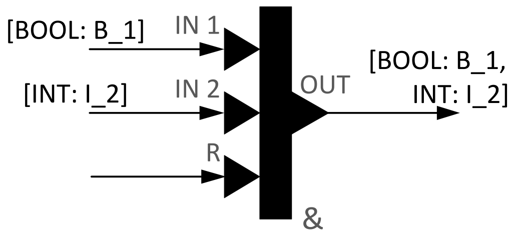
REND (AND-Join)
The *REND* Control-Flow Activity (i.e., Rendezvous) synchronizes[2] two incoming control-flow branches into a single branch. This means that the control-flow is passed to the outgoing branch, once both incoming branches have been enabled. Figure 4.3(a) shows the scheme of the CFA type in two different notations, with the incoming branches *IN 1* and *IN 2* and the outgoing branch *OUT*. In the case, the synchronization has to be cancelled, the *Reset* input *R* can be triggered. Initially, if an instance of a REND activity is created, the Activity Interfaces *IN 1*, *IN 2*, and *OUT* are *untyped*. The data types of the input interfaces are determined by establishing connections from preceding activities to the Activity Interfaces. Once a connection is established, the corresponding data types are acquired from the source Activity Interface.

Apart from synchronizing the control-flow, also the incoming data channels need to be *merged* as displayed in Figure 4.3(b). It is considered that every Activity Interface

[2]Due to its synchronization functionality, the activity type is also called *AND-Join* (see [18] or [56]).



(a) Scheme of the Control-Flow Activity type *REND* in two different notations



(b) Merging of incoming data channels

Figure 4.3: Scheme of the Control-Flow Activity type *REND* and an example of merging data channels

has a list of Activity Parameters (AP) and henceforth data types are associated with it. The REND activity *concatenates* the contents of both AP lists of the input interfaces and creates a copy of the combined list which is finally assigned to the *OUT* interface.

While designing the Workflow Specification in the Workflow Modelling Editor, the following *rules* are applied to *REND* activities. Firstly, a REND CFA is only correctly used within a Workflow Specification if at least the Activity Interfaces *IN 1*, *IN 2* and *OUT* are connected. Secondly, the *OUT* interface cannot be connected to a subsequent activity if one of the input interfaces *IN 1* or *IN 2* are not connected. This is because otherwise the data types for the output interface would not be correctly defined. Finally, deleting connections having the target *IN 1* or *IN 2* leads to the deletion of any outgoing connections from the *OUT* interface. The *Reset* input does not need to be connected if it is not in use.

A typical situation which indicates the appliance of a *REND* CFA is given by a subsequent Service Activity depending on data values, which are produced in separated control-flow branches. The REND activity is then used to synchronize both control-flows and merge the produced data objects into one control-flow. Using the new combined data and control-flow, which provides the necessary data objects, the activity can be invoked correctly (compare with Figure 4.3(b)).

MERGE (OR-Join)

The *MERGE* Control-Flow Activity merges two incoming branches into one but in contrast to the REND activity *without* synchronizing them. This means that an incoming control-flow at an arbitrary input Activity Interface is immediately forwarded to the output Activity Interface. In Figure 4.4, the MERGE activity is depicted schematically with its input and output Activity Interfaces. The MERGE activity requires both combined control-flow and data channels, which arrive on the input interfaces *IN 1* and *IN 2* having the same data types. This restriction is necessary since a subsequent activity of the MERGE activity requires to be called with the same parameter types during the whole lifetime of a created Workflow Instance.

Similarly to the REND CFA, after creation of a new MERGE activity instance in the Workflow Modelling Editor, the data types of all interfaces are not defined. Once one of the input interfaces is connected to a preceding activity, the data types are defined for both, the second input interface as well as the output interface.

Using MERGE activities in a Workflow Specification requires the following design *rules* to be obeyed. MERGE CFAs are only applied correctly if all Activity Interfaces are connected. The *OUT* interface cannot be connected to a subsequent activity, if none of the input Activity Interfaces is connected to preceding activities. This results in the fact that the output data types would not be defined. Deleting both connections, having the targets *IN 1* and *IN 2*, leads to the deletion of all outgoing connections from the *OUT* Activity Interface.

MERGE Control-Flow Activities are generally used to unify two execution branches, which especially have been split previously in the Workflow Specification (e.g., using an *IF* CFA) and it is not defined which of these branches is taken before workflow execution. Additionally, we can use this type of CFA to model an *execution loop* within the Workflow Specification.

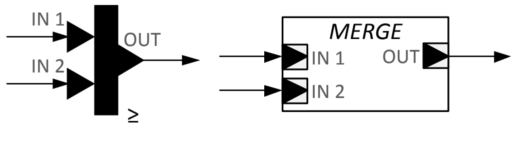


Figure 4.4: Scheme of the Control-Flow Activity type *MERGE* in two different notations

SPLIT (Fork)

Apart from merging and synchronizing, control-flows can be split into two or more concurrently executing branches, according to Figure 4.5. In this Workflow Modelling Language implementation, the *SPLIT*[3] CFA is realized implicitly by defining activities to have output interfaces, that support the *Fan-Out* of outgoing connections to subsequent activities. As a consequence of multiple output connections, besides the control-flow, also the data objects are available to the first activity in every parallel branch.

IF-Decision (XOR-Split)

By introducing the *IF* Control-Flow Activity, a means of splitting the Control-Flow, depending on the evaluation of a boolean condition, is provided. A schematic overview of the IF activity type is presented in Figure 4.6. The Activity Interfaces *IN 1* and *IN 2* are used to specify the data values which are used within the expressed condition, that is specified as a constant string-type value at interface *COND*. It has to be noted that the interface *COND* does not accept any incoming connections[4] since the expression of the boolean condition may not change during execution. The input parameter *IN 2* can also be set as a *Constant Parameter*. *Constant Parameters* are introduced in Section 4.1.3.

Condition expressions must result in boolean values and have to contain the names of the input Activity Interfaces (e.g., "IN_1" or "IN_2"). Moreover, conditions need to be expressed in *Structured Text (ST)*[5] syntax. For example, it is assumed to compare two *integer* data values, where *IN 1* receives an integer parameter from the preceding activity and *IN 2* has a *Constant Parameter*. To check for equal integer values, the condition expression needs to be set as "IN_1 = IN_2".

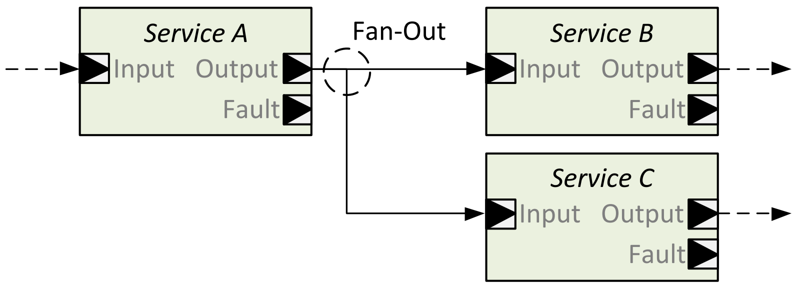


Figure 4.5: Connection Fan-Out in order to split the control-flow

[3]Alternatively named as *Fork* pattern according to [56].
[4]Activity Interfaces which do not allow any connections are denoted with missing arrows.
[5]For detailed information regarding the programming language *Structured Text* see [16] or [48].

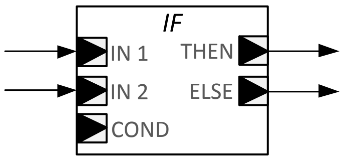


Figure 4.6: Scheme of the Control-Flow Activity type *IF*

Once the combined control- and data-flow arrives at the *IF* activity's interface *IN 1*, the condition *COND* is evaluated. A control-flow arriving at interface *IN 2* does *not* trigger the evaluation of the condition. In case of a positively evaluating condition, the control-flow is passed to the output interface *THEN*, otherwise *ELSE* is activated. Both of the output Activity Interfaces are untyped.

An IF activity is used correctly within the Workflow Specification if the *IN 1* interface is connected to a preceding activity and the combined data- and control-flow carries *at least one* data object, that can be included in the condition expression. The input Activity Interface *IN 2* may remain unconnected, requiring a constant parameter value to be specified. In the current implementation of the Workflow Modelling Language, the condition expression is not verified in terms of syntax correctness. Errors occurring due to incorrect condition expressions are reported during the generation of the supervisory control application.

WAIT Control-Flow Activity

The *WAIT* Control-Flow Activity enables the workflow designer to *delay* the control-flow for a specified time-span. To the best of the authors knowledge, there are only a few possibilities to model time-delay functionalities in Workflow Specifications. For instance in Petri-Net based languages, *delayed transitions*, which fire upon a defined time-span is exceeded after their enabling, are introduced in [10]. Analogously, it can be considered that Petri-Net tokens have to reside in a place for a given minimum time duration [10], [29]. These methods are also applicable to UML Activity and State-Machine diagrams [43].

WAIT Control-Flow Activities provide a similar functionality such as time-spans that are defined for Petri-Net places or State-Machine states. In this case, the control-flow is bound to the WAIT activity for the specified time duration, before it is passed to the subsequent activities.

Figure 4.7 depicts the schematics of the WAIT activity, having two input Activity Interfaces *Start* and *Stop* as well as one output interface *OUT*. To set the time-span,

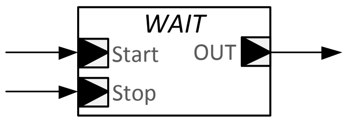


Figure 4.7: Scheme of the Control-Flow Activity type *WAIT*

the interface *Start* has one Activity Parameter of the data type *TIME* associated with it. For a fixed time-span a Constant Parameter can be set. After the combined control- and data-flow has been passed to the *Start* interface, the activity is executed by starting a *timer*. Finally, the control-flow is passed to the output interface *OUT* and the WfMS continues the execution of the branch. By passing the control-flow to the interface *Stop*, the running timer is cancelled, causing the interruption of the control-flow.

There are no special rules which have to be obeyed to ensure the correct behaviour of the executing Workflow Instance. The Activity Interface *Stop* does not require an incoming connection if the possibility of cancelling the control-flow delay shall not be given. *WAIT* Control-Flow Activities can be used in order to model for instance processing time of real-life workflows.

CALCULATE Control-Flow Activity

The *CALCULATE* CFA enables the workflow designer to include simple calculations into a Workflow Specification. Hereby, the calculation involves at least one *numerical* input Activity Parameter (AP) of the Activity Interface *IN*. A similar activity construct is provided in the specification of the *Visual Programming Language (VPL)* [36]. Figure 4.8 shows the scheme of the *CALCULATE* CFA. The input AP *CONFIG* requires *two* Constant Parameters of type *STRING* to be specified. One parameter holds the *expression* which shall be evaluated and may involve several input APs, received by the interface *IN*. Similar to the condition expression of the *IF* CFA, the calculation expression has to be determined in *Structured Text* notation. As value for the second input AP, the resulting output data type (i.e., the data type of the output AP of interface *OUT*) has to be defined. This AP allows

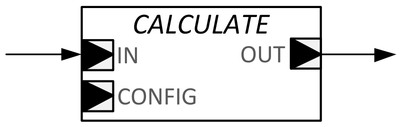


Figure 4.8: Scheme of the Control-Flow Activity type *CALCULATE*

only numerical data types to be set (see data types according to [28]). Specifying the output data type manually results in the advantage that the input parameter type can be *converted* to a different numerical data type. However, it is the user's responsibility to choose the output data type appropriately in order to avoid *data loss*[6].

START Control-Flow Activity

In Section 3.4.5, the importance of defined *start*- and *end*- points has been pointed out. The *START* CFA is unique in the Workflow Specification which implies that only one instance of a START activity can be created[7]. Marking the origin of the control-flow is the main purpose of this activity type.

Moreover, the START activity never can represent a work item that has to be performed. This feature is expressed by defining this CFA as the only activity type which does not have any input Activity Interfaces but one output interface, as displayed in Figure 4.9. Once a new Workflow Instance is created and execution is commenced by the WfMS, the *START* activity produces the initial control-flow and passes it to the output Activity Interface *OUT*.

As stated in Section 3.4.5 in contrast to START activities, there is no necessity to introduce *END* activities within the Workflow Definition, that is located on the highest hierarchy level. An END CFA is used in sub-workflows to mark the ending of such a workflow element, denoting that the control-flow can be passed back to the calling workflow level.

SIGNAL (External Trigger)

Similar to the *START* activity, the *SIGNAL* CFA is capable of initiating a new *control-flow* on a triggering request that can originate from the external environment (i.e., from a resource). In combination with a subsequent branch of activities that is

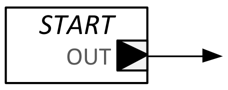


Figure 4.9: Scheme of the Control-Flow Activity type *START*

[6]For example, the conversion from *INT* to *REAL* is possible without data loss, whereas the reverse conversion results in data loss.

[7]Uniqueness of the START activity is related to the scope of (sub-)workflows, meaning that one START CFA is also required in every sub-workflow.

connected to the *SIGNAL* activity this structure corresponds directly to the *External Trigger* pattern which is described in [61].

External Triggers can be modelled in several Workflow Modelling Languages. For example, the *UML Activity Diagrams* provide *Event Acceptor* nodes which initiate a new control-flow if the defined event is received [43], [50]. Further examples are given for Petri-Net based languages, where special places are filled with a token in case a defined event is received by the WfMS (see [14]), and *JECA rules* which can be triggered trough external events (see e.g., [31] and Section 2.2.6).

Figure 4.10 shows how the *SIGNAL* activity is defined in this modelling approach. The missing arrows on the input Activity Interfaces denote that incoming connections are not supported. Instead, *Constant Parameters* need to be set for every input interface. Parameter *SourceID* is of type *STRING* and holds the *connection endpoint ID* of the connection, that is used to receive signals from. In this context, the *boolean* Constant Parameter *UDP_TCP* determines whether the signal is received using UDP Multicast (*false*) or a TCP (*true*) connection[8]. By using the Constant Parameter *RCVData*, a set of data objects can be defined which are received along with the signal. For example, the value *BOOL, INT* enables the SIGNAL Activity to receive two data objects, of the given data type. If the connection endpoint receives an event, a new control-flow combined with data objects is started and passed to the output interface *OUT*.

4.1.3 Combination of Control- and Data-Flow

As stated in Section 3.4.2, this Workflow Modelling Language builds on the combination of *Control- and Data-Flow* with the major aim to keep Workflow Specifications

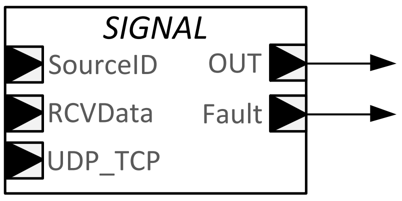


Figure 4.10: Scheme of the Control-Flow Activity type *SIGNAL*

[8]In case of UDP (User Datagram Protocol) the SIGNAL activity corresponds to an endpoint of a *UDP Multicast* communication. TCP (Transmission Control Protocol) determines a Client/Server connection.

as simple as possible. The *control-flow* defines the execution order of the activities within a Workflow Specification (see Section 2.2.1). In graphical Workflow Modelling Languages it can be considered the control-flow is bound to predefined *paths* within a Workflow Specification, which consist of several *Activity Connections (ACs).*

For the following discussion, a section of a Workflow Specification containing two Service Activities *Service A* and *Service B* with their corresponding Activity output and input parameters as shown in Figure 4.11 is considered. The solid, black arrow denotes the *Activity Connection*, having *Service A* as source activity and *Service B* as its target activity. Furthermore, lists of Activity Parameters are defined for the connected Activity Interfaces.

With reference to the *data visibility scopes* that are introduced in Section 3.4.2, it can be derived that the output Activity Parameters of *Service A* are also visible for the input interface of *Service B*, according to Figure 3.5(b). Consequently, the *data values*, which are stored within the Activity Parameters of *Service A*, are denoted to be *combined* with the control-flow leading from *Service A* to *Service B.*

In order to pass *data values* from one activity to the next, output Activity Parameters need to be assigned to input Activity Parameters of the subsequent activity. This functionality shall be named *Parameter Mapping.* Note, that the *Parameter Mapping* defines which data values, originating from the preceding activity's output parameters, are passed to which input Activity Parameters at the moment the corresponding SA is invoked.

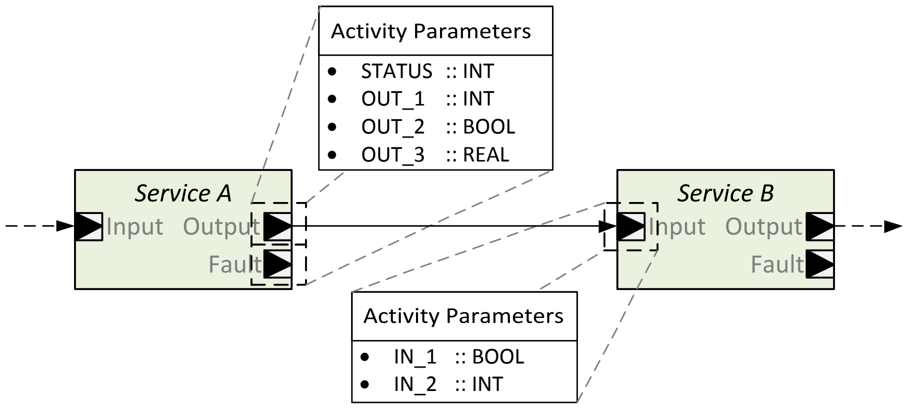


Figure 4.11: Connected Service Activities with corresponding Activity Parameters

Mapping of Activity Parameters

To enable data transport between two connected activity instances, *Parameter Mappings* need to be established. This is accomplished by providing a so-called *Parameter Mapper* for every input Activity Interface[9]. A *Parameter Mapper* is realized as an indexed table that associates output APs of the *source* Activity Interface to input APs of the *destination* Activity Interface.

Figure 4.12 shows the relations described above. In this example, the source parameters *OUT_1* and *OUT_2* are mapped to the target parameters *IN_2* and *IN_1*, respectively. Observe, that the *data types* of the source parameters must conform to the target parameter data types at any time during workflow design. Since type correctness is crucial for the correct execution of a Workflow Instance, the data type check is performed everytime the mapping table has changed.

The following rules should be kept in mind when establishing *Parameter Mappings* in Workflow Specifications. Firstly, any information stored in mapping tables is deleted, once an Activity Connection which leads to the corresponding Activity Interface is removed. This means that previously defined mappings have to be re-established if the Activity Connection is reconnected.

Secondly, if the preceding activity does not offer appropriate parameter data types, a *Constant Parameter* of the required data type can be defined instead. Lastly, the

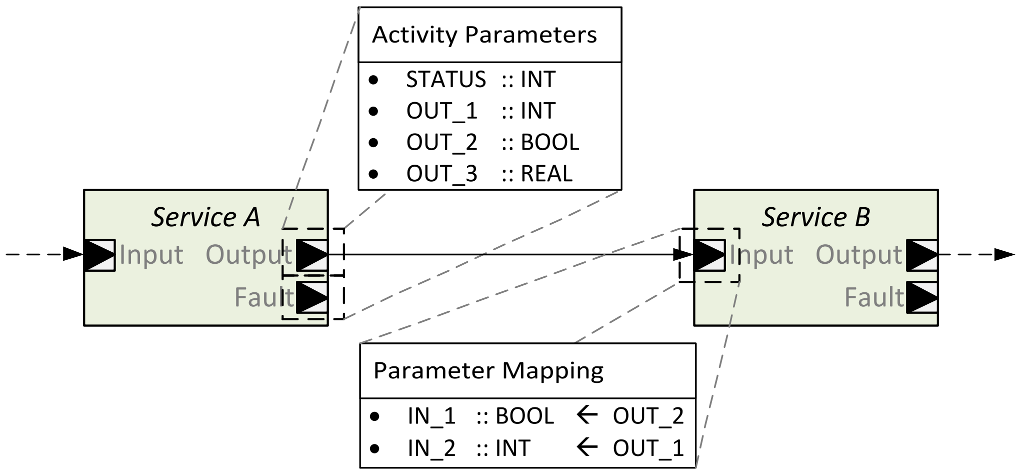


Figure 4.12: Mapping of source Activity Parameters to target Activity Parameters

[9]Note that *Parameter Mappers* are also provided for Activity Interfaces which do not allow the presence of incoming Activity Connections. For example, referring to the *SIGNAL* CFA in Section 4.1.2, which requires a ConstantParamter to be set, instead.

generation of an IEC-61499 FB control application out of the Workflow Specification can only be performed if all input Activity Parameters are mapped to source Activity Parameters or *Constant Parameters*.

Constant Parameters
With reference to Section 3.4.2, *Constant Parameters*, which hold constant data values of a specific type, can be defined either visible within the scope of the whole Workflow Instance as shared parameters (see Figure 3.5(a)) or locally, defined within the scope of an input Activity Interface (see Figure 3.5(b)).

Constant Parameters can be mapped to input Activity Parameters, independent from parameters which are provided by the source activity of an established Activity Connection. For some activity types, which do not allow incoming Activity Connections to be established to corresponding input Activity Interfaces, Constant Parameters provide the only possibility to set a constant input data value.

4.2 Extended Concepts

This section focuses on further concepts of the proposed Workflow Modelling Language. Some of them are not yet implemented. However, in this case solution strategies are suggested and explained. The first part provides a concrete example of a Workflow Specification. Secondly, the example is used to demonstrate how *embedded Exception Handling* strategies can be realized. In the third part, a method for structuring Workflow Specifications into hierarchical levels using sub-workflows is explained. Although, this feature does not exist yet, possible application fields for hierarchical structures in workflows will be introduced. Finally, possibilities for verifying Workflow Specifications are discussed.

4.2.1 A Workflow Specification Example

In the following, a section of a Workflow Specification is considered that describes a *Bin Picking Process*. Objects which are stored in a box should be grasped by a robot manipulator and put out of the storage box (see e.g., Figure 3.1). The Workflow Specification contains Service Activities of an *Object Recognition (OR)* system, a *Path Planning (PP)* unit and a *Manipulator Controller (MC)* of a robotic system.

In order to react upon crucial execution errors properly, the Workflow Specification includes exception handling branches.

Firstly, an object recognition system (e.g., a laser sensor) has to scan the storage box in order to obtain a *point cloud* of the objects. Furthermore, the scanning results are analysed and the output of the OR process is assumed to be an array of *4x4-matrices*, containing floating point values (data type *MATRIX4F*[]). Every matrix describes a transformation from the defined *world coordinate system* to the *centre point* of a correctly detected object. Secondly, the OR results are passed to the *Path Planner* which chooses one object and generates a *RobotProgram* (data type *PATH*), that can be submitted to the *Manipulation Controller*. The complex data types *MATRIX4F*[] and *PATH* are application-specific. Finally, the MC initiates the grasping of the detected object.

Service Activities

Figure 4.13 shows the section of the Workflow Specification as described above. It is important to know that this functionality could be realized within a sub-workflow in order to simplify the overall workflow structure[10]. In the following, a brief description of the Service Functions and their required input Activity Parameters and produced output data values is given.

The Service Activity (SA) *doOR* performs the object recognition procedure. The produced array of *transformation matrices* is accessible for the System Components (SC) *OR* and *PP*. Similarly, it is assumed that the *RobotProgram* is accessible for both the *PP* and the *MC*. An overview of the input and output parameters of the SAs is given in Table 4.2 and Table 4.3. Every Service Function (SF) provides at least the output Activity Parameter "STATUS".

Hereby, the Activity Parameter *ObjectID* denotes which of the detected object positions should be used for the calculation of the grasping path (i.e., *RobotProgram*). The parameter *ModelFileName* refers to a file that contains *shape information* of the object which should be detected (e.g., a CAD model file).

[10]Modelled within a sub-workflow this specification section would require start- and end- activities in order to mark defined start points and end points.

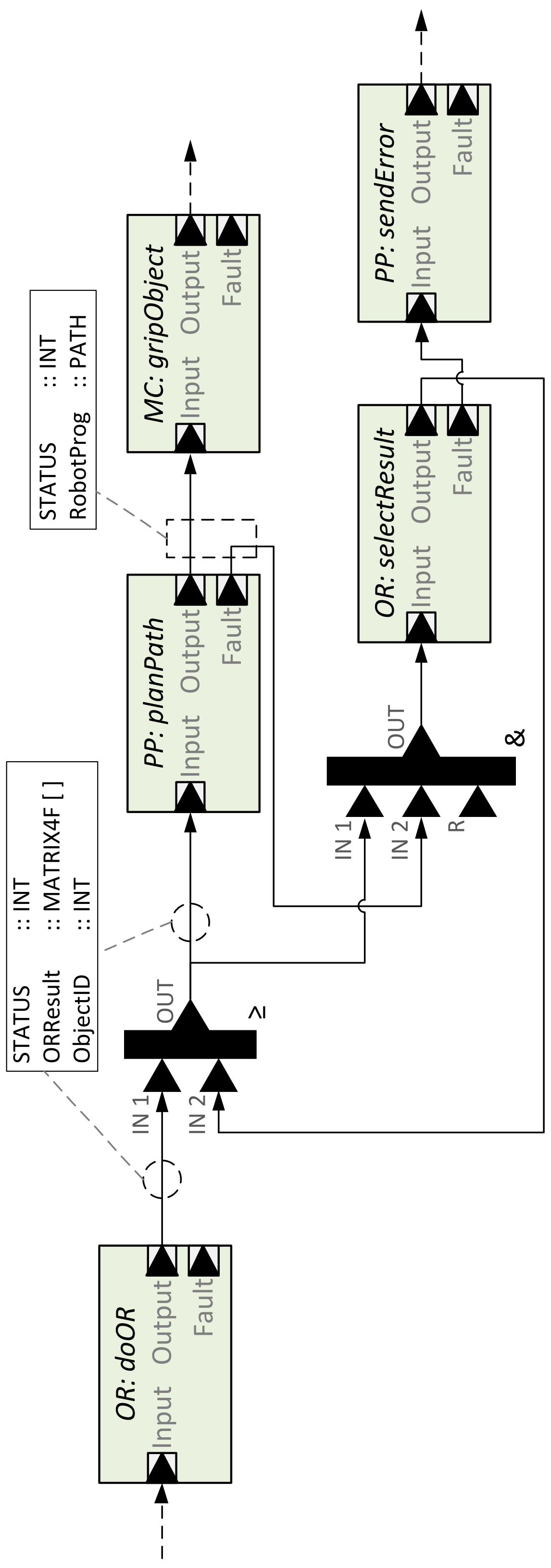


Figure 4.13: Workflow Specification section for the Bin Picking process, including exception handling branches

Service	**Inputs**
doOR	[ModelFileName::STRING]
planPath	[ORResult::MATRIX4F[], ObjectID::INT]
gripObject	[RobotProg::PATH]
selectResult	[ORResult::MATRIX4F[], currentObjectID::INT]
sendError	[]

Table 4.2: Input Activity Parameters of the Service Activities

Service	**Outputs**
doOR	[STATUS::INT, ORResult::MATRIX4F[], ObjectID::INT]
planPath	[STATUS::INT, RobotProg::PATH]
gripObject	[STATUS::INT]
selectResult	[STATUS::INT, ORResult::MATRIX4F[], ObjectID::INT]
sendError	[STATUS::INT]

Table 4.3: Output Activity Parameters of the Service Activities

If the service *planPath* cannot calculate a grasping path for a given matrix transformation that corresponds to a certain detected object, the SA throws an exception and the control-flow is forwarded to the *Fault* output. This might be the case, for instance, if there are only path options calculated that would lead to the collision of manipulator axes with an obstacle in the work cell. A possible solution to this problem is to try grasping another object that has been detected.

The SA *selectResult* is used to select an alternative matrix transformation and returns a new *ObjectID*. Additionally, the transformation that is referred by the input value *currentObjectID* is removed from the matrix array. If no alternative objects are available, an exception is thrown and the control-flow is passed to the *Fault* output. Finally, the SA *sendError* is used to propagate an error signal, for instance to the highest Workflow Specification level, which can be received using a *SIGNAL* CFA (see Section 4.1.2).

4.2.2 Realization of an Exception Handling Strategy

As stated in Section 4.1.1, in case of *erroneous* execution of a Service Activity, the *Fault* output interface can be used to route the control-flow to an exception handling branch. This form of exception handling, realized within the Workflow Specification is called *embedded Exception Handling* [52]. In Section 2.2.2, basic exception handling strategies are presented for embedded as well as autonomous exception handling methodologies. The example that will be presented below is based on a so-called *Forward Recovery* handling strategy. This means that the effects, that are caused by the exceptional event, are compensated through the execution of an alternative workflow path.

For this example, it is assumed for simplification reasons of the considered case, that the SAs *doOR*, *gripObject* and *sendError* execute correctly and henceforth, no handling strategies need to be constructed for these activities. An exception handling strategy to manage a *Path Planner* error is shown in Figure 4.13. The exception handling branch encompasses the workflow activities *REND*, *selectResult* and *sendError*. According to the figure, the service *planPath* receives the data values *ORResult* and *ObjectID* from the SC *OR*[11].

It is considered that the PP cannot calculate an appropriate path (i.e., *RobotProgram*) for the given *ObjectID*. Therefore, the control-flow is forwarded from the *Fault* output to the input of the *REND* CFA. To compensate the exception effect, the SA *selectResult* is used to choose an alternative *ObjectID*. However, *selectResult* requires inputs according to Table 4.2 which are not available within the combined control- and data-flow originating from the SA *planPath*. Because of this, it is necessary to synchronize the branches, shown in Figure 4.13, using the *REND* CFA to make the desired data objects available.

Considering the case, the SA *selectResult* can select an alternative *ObjectID*, the OR result is updated and the “new” *ObjectID* is forwarded, with the *ORResult* and *STATUS* values, to the *MERGE* CFA and consequently passed to the SA *planPath* again. If the SA *selectResult* cannot find an alternative, which means that there are no other detected objects left, the whole process part is considered to be failed and an error signal is sent by *sendError*.

At this point it is worth mentioning that the *status value* is always valid, even in the case of erroneous behaviour of a SA. If the control-flow is routed to the *Fault* output interface and exception handlers are installed, the status value can be used to distinguish between different exception types. For instance, an *IF* CFA can then be used in order to select the correct exception handling branch.

Summarizing, the issue, stated in Section 2.2.2, can be confirmed that embedded exception handling solutions tend to increase the complexity of Workflow Specifications, especially if a variety of exceptions should be managed. As a main advantage, embedded exception handlers clearly define the exception origin within the Workflow Specifications. Furthermore, the constructed handling strategies can be

[11]The control-flow, including the data values, is forwarded trough the *MERGE* CFA.

retraced easily, requiring the workflow designer to be familiar with the rules of the Workflow Modelling Language.

4.2.3 Hierarchical Workflow Specifications

The application of hierarchical structuring methodologies in the field of *component based modelling* enjoys increasing popularity in research (see Section 2.3.3). Several benefits of hierarchical structures can be directly applied correspondingly to Workflow Modelling Languages which support the *sub-workflow* construct. Utilizing sub-workflows in Workflow Specification results in the following advantages (see [34], [70], and [6]):

- The combination of *functional connected* Service Activities into a sub-workflow leads to a simplified representation of a complex Workflow Specification.
- Once a sub-workflow has been defined, it can be *reused* in order to compose a more complex functionality. In this way higher level functional components can be created of existing lower level service compositions.
- The *root workflow* can then be considered to have a coordination functionality, supervising the execution of lower level hierarchically composed sub-workflows.

In the following, a possible application for sub-workflows with the goal to simplify Workflow Specifications, which are intended to describe the behaviour of a supervisory control application (as needed in this work), is presented. A robotic system, which consists of a set of SCs, including for instance a *Mobile Platform (MP)*, a *Manipulator*, an *Object Recognition (OR)* system and a *Human/Machine Interaction (HMI)*[12] system, is assumed.

The SCs are required to interact among each other in order to make the system reactive on signals which are propagated from SCs, in case of special events. For example, in case of the HMI detecting a gesture which means “Stop”, the MP should stop moving immediately. In this case, the HMI would send an event that can be received by either the SC Mobile Platform (MP) directly or by the *supervisory control application* which then has to forward a corresponding request to the MP. Additionally, unresolvable exceptions originating from Service Components can be propagated to the control application (i.e., the root level workflow) which then can initiate compensation strategies.

[12]A *Human/Machine Interaction* SC interacts with the OR system and is capable of interpreting for instance human gestures or voice commands.

Independently of how this event propagation is handled, in the author's opinion, it is practicable to pack *functionally coherent services* into sub-wokflows, including the necessary embedded exception handling strategies and services which send events to a given address. To mention an example, the combination of OR services and the Manipulator in order to grasp a part in a storage box (see example in Figure 4.13) is suitable to be packed into a sub-workflow.

Applying this methodology to functional connected Service Activity compositions, reduces the complexity of Workflow Specifications very likely. The requirement for a Workflow Management System (WfMS), which executes hierarchical Workflow Specifications, to support reactive execution is still fulfilled.

4.3 Concluding Remarks

In this chapter a Workflow Modelling Language, which complies with the requirements for the modelling approach, presented in Section 3.4, has been proposed. The first part focused on the description of the language elements and concepts, including Service Activities and Control-Flow Activity types as well as the combination of Control- and Data-Flow. Additionally to the characteristics of Service Activities, the description format for Service-Components and their corresponding Service-Functions has been introduced. Based on a thorough research on control-flow patterns, implemented in different Workflow Modelling Languages, a set of common Control-Flow Activities is presented. As an important concept for the data passing between activities, the *Parameter Mapping* method has been explained.

Extended Concepts of the Workflow Modelling Language, including a Workflow Example with exception handling strategies, considerations on hierarchical Workflow Specifications, and Workflow Specification verification are discussed in the second part. An example Workflow Specification has been discussed to show the possibilities of embedded exception handling strategies and to reveal their advantages and limitations.

5 Generation of a Control Application

In this chapter, the automatic generation of an IEC 61499 control application which is intended to coordinate the Service-Components of a modelled robotic system is discussed. This generation is based on a Workflow Specification that is expressed with the aid of the introduced Workflow Modelling Language.

With reference to Section 3.5, a set of requirements for the *code generator* has been identified. The main objective is to generate a FB-Network which provides the desired *supervisory control functionality*, and is ready for execution through an IEC 61499 compliant runtime environment. As a main requirement, the control application should be able to be executed either on real hardware platforms, or by coupling the runtime environment with the 3D-Simulation Environment to control the modelled Service-Components.

The first section provides a brief overview of the code generation by describing its processing steps. The second part presents a general approach for generating FB-Types, where especially Basic FB-Types and Composite FB-Types are considered. Thirdly, a selection of examples for the generation of representing FB-Types for Control-Flow Activities is presented. The fourth section deals with the generation of FB-Types representing interfaces to Service-Components. Lastly, the generation of the supervisory control application is explained.

5.1 Code Generation Process

This section focuses on a description of the code generation procedure, which is based on the requirements and explanations provided in Section 3.5. The main goal of the code generator is to automatically create a FB control application which represents the control behaviour, described by the corresponding Workflow Specification.

After structural and syntactical checks of the Workflow Specification have been performed correctly, the generation process starts by generating representative Function Block Types for the modelled workflow activities. According to the language elements of the developed Workflow Modelling Language, both Control-Flow Activities as well as Service Activities require representative FB-Types to be generated. Based on the generated FB library group, a FB control application is generated by creating FB instances of the generated FB-Types. The modelled activity connections are represented by event and data connections. Additionally, to make sure that the created FB instances are initialized correctly, a special section of a FB-Network is generated. This network section performs a *supervised* initialization, before the execution of the control application is commenced.

Finally, the resulting FB control application can be executed by an IEC 61499 compliant runtime environment. The runtime environment is executed on a hardware target platform, or coupled with the 3D-Simulation environment, thus enabling iterative programming cycles.

5.2 Generation of FB-Types

This section deals with generation of FB-Types that are needed in order to express the defined Workflow Specification by a FB-Network, providing supervisory control functionality.

The language elements of the developed Workflow Modelling Language encompass a set of eight Control-Flow Activities (CFAs). These constructs enable the workflow designer to influence the execution sequence during runtime, for instance by branching the control-flow based on a given condition. For every modelled CFA instance, a representing FB-Type has to be created based on generation rules which are determined for the every CFA type. These generated FB-Types are henceforth denoted as *Control-Flow Activity FB-Types (CFA FB-Types)*. The certainty of having different types of *data values* being bound to the control-flow, implies the generation of a new FB-Type for almost every CFA, although the FBs internal logic would be the same (e.g., consider two REND activities with different input data types).

According to Section 3.5.1, the invocation of Service-Functions (SFs) is accomplished by using FB-Types which act as interfaces to the corresponding, external Service-Components (SCs). To enable communication between a component and the supervisory control application, a connection needs to be established. In this approach a synchronous communication solution based on a TCP connection service is used because synchronous communication supports reactive workflow execution [14]. Since one FB-Type is generated to connect to a certain SC, the FB has to provide a FB-Interface which serves several SF calls that affect the same SC. An approach to generate these so-called *Service-Component FB-Types (SC FB-Types)* is presented later in this chapter.

In IEC 61499 compliant Engineering Environments, FB-Types are defined in two steps (see e.g., [17] or [21]). Firstly, the *interface*, which encompasses inputs and outputs such as events and data variables of the FB-Type, needs to be constructed. Secondly, the FB's behaviour has to be defined, whereat different concepts are used for the behaviour definition, depending on the FB's *general type* (i.e., Basic, Composite, or Service-Interface). The following sub-sections, describe how the generation of FB-Types is accomplished.

5.2.1 General Design Rules

Before the generation of FB-Types is explained in detail, a set of design rules is introduced below, which should be kept in mind during the whole FB code generation process.

Both, FB-Types and FB instances need to have *unique names*, by which they are identified within the type library and the FB-Network, respectively. The same rule applies to *interface elements*, this means input/output events, as well as input/output data variables. Ambiguous event and data variable names are not permitted according to the IEC 61499 standard definitions [28], [1].

Regarding the generation of FB-Interfaces, naming conventions for inputs and outputs (data and event) have been made in order to allow an easy classification within the interface. For example, the name prefixes "IN" and "OUT" are used in order to mark data inputs and data outputs. In general, the definition and use of consistent naming schemes prevents program development mistakes to happen. At the same time the program code readability is improved.

5.2.2 Generation of FB-Interfaces

This section deals with the generation of FB-Interfaces, based on the information that is provided by Activity Interfaces and their associated Activity Parameters

(AP). The method to generate FB-Interfaces which is presented in the following, can be applied to several FB-Types (i.e., Composite FBs or Basic FBs).

Apart from event/data inputs and outputs, FB-Interfaces are characterized through compositions of single events (input or output) with sets of data variables. These event to data variable associations are called *WITH constructs* [28], [9]. They describe the data values that are passed to a FB-Algorithm, whose execution is triggered through the corresponding event.

According to Section 4.1, activities are modelled having a set of Activity Interfaces which act as source and endpoints of Activity Connections. Furthermore, it is defined that every Activity Interface possesses a list of APs, each of them having a defined data type. Compared to the WITH construct, the definition of Activity Interfaces and corresponding APs can obviously considered to be *analogous*. Figure 5.1 depicts a scheme how Activity Interfaces and their corresponding APs are mapped to events and data variables. The upper part of the image displays a general activity instance, with its Activity Interfaces *IN* and *OUT*, and the associated APs.

For the generation of the FB-Interface, the Activity Interface combined with the APs is directly represented as *one* WITH construct. This *mapping rule* is highlighted by the solid arrows, leading from the Activity Interface to the event and data variables which together correspond to WITH constructs. The data types of the generated data variables are matching with those of the defined APs. For activities having multiple Activity Interfaces on both, the input and the output sides, the mapping rule can be applied similarly to generate the related FB-Interface.

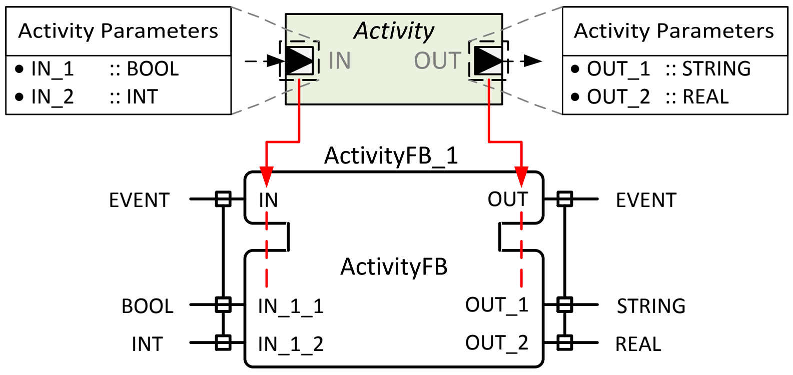


Figure 5.1: Mapping of Activity Interfaces and Activity Parameters to events and associated data variables

In case of several events being generated on one interface side, it has to be assured that every event and data variable is named uniquely. Considering CFA FB-Types, input and output events are named equally to the corresponding Activity Interface. For SC FB-Types, the event names correspond to the instance names of the Service Activities with additional pre- and postfixes. This is necessary to distinguish between different Service-Function calls that need to be served by the FB-Type. Detailed explanations are given later in this chapter. Data variables are named according to the scheme *IN_x_y*, with y denoting the number of the data variable that is associated with the event number x, counting from the top of the interface side. The same naming conventions are used for output events and data variables.

This naming scheme cannot be applied in case of input Activity Interfaces, which do not allow any input or output Activity Connections but require *Constant Parameters* to be defined. For example, Activity Interfaces are considered which are intended to only hold constant configuration parameters. Such configuration values are needed either during the Workflow Modelling phase or for the *code generator*[1]. Mapping an Activity Interface definition of this type to a FB-Interface would result in an event input which remains unused.

After describing an approach to generate FB-Interfaces, the generation of a FB's behaviour is considered next.

5.2.3 Generation of a Basic FB-Type

The behaviour of a Basic Function Block is described by an Execution Control Chart (ECC). Basic FBs turned out to be very suitable in order to describe logical operations (see [28] or [1]), like most of the CFA types are based on, especially the introduced activity types *REND*, *MERGE*, and *IF*. The generation of an Execution Control Chart can be summarized into four generation steps, described below.

- *Declaration of Internal Variables to store intermediate processing results.*
- *Generation of EC-States for every input event.*
- *Generation of EC-Actions and encompassed FB algorithms.*
- *Generation of EC-Transitions to enable state changes.*

[1]For example, the Activity Interface *CONFIG* of the CFA *CALCULATE* (see Section 4.1.2) is used to determine the activities' output data type. Therefore the Constant Parameter which is mapped to this interface is only used during the workflow modelling phase.

Internal variables are used to store intermediate processing results of the FB's algorithms. Moreover, results of complex boolean conditions are stored in internal variables. Instead of associating a complex condition expression to an EC-Transition a single internal variable is assigned. This improves the readability of the ECC.

An EC-State is generated in general for every input event of the FB, because an input event requires an action to be performed. The defined EC-Action produces output data and triggers an output event or initiates the transition to another EC-State. In this context it is worth mentioning that in many cases, additional *intermediate* EC-States are required, which need to be reached before the result output data can be processed. For example, consider a FB that has an initialization *algorithm* (e.g., to set internal variables of the FB), which must be executed correctly in advance, before any other action can be invoked. To accomplish such a behaviour, a *Ready* EC-State is introduced which denotes that the FB has been initialized correctly when it is reached. Only if the ECC resides in the Ready state, further actions can be executed. Otherwise, incoming events are dropped.

Algorithms define the main behaviour of the FB [28]. They are intended to process the input data values, to evaluate condition expressions, and to write intermediate results or end results into the designated data variables. The output events which need to be triggered are combined with the defined algorithms, thus expressing EC-Actions in the ECC. An EC-Transition enhanced with a condition can be used to test a data variable (e.g., internal variable or input data variable) for a certain value, as well as for testing the presence of a certain input event. Based on the condition result, a state change is initiated which consequently leads to the execution of another EC-Action.

5.2.4 Generation of a Composite Function Block Network

Apart from Basic FB-Types, Composite FB-Types are applied whose behaviour is described by an internal *Composite Function Block Network*. Composite FB-Types are mainly needed to realize the Service-Component FB-Types that represent interfaces to Service-Components. The generation of Composite FB-Types is performed in four steps, which are described in the following.

- *Generation of the FB-Interface.*
- *Creation of the required internal FB instances.*
- *Generation of internal event and data connections.*
- *Association of the internal FB-Network with the Composite FB-Type.*

Firstly, the FBs Interface needs to be generated as already described. The interface elements (i.e., event connectors and data connectors) are used to route data and events into the Composite FB-Network. Secondly, the required FB instances which should be contained within the Composite FB-Network are created and added. This requires, that the corresponding FB-Types are available in a FB-Library.

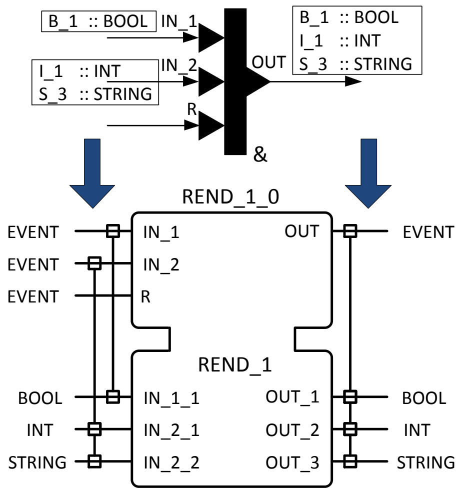
In the third step, the event and data connections, which define the behaviour of the Composite FB-Network, are established. The set of connections encompasses connections between the internal FBs, as well as connections between internal FBs and the left and right FB-Interface sides of the Composite FB-Type. Lastly, the Composite FB-Network instance is associated with the Composite FB-Type and the generated FB-Type is stored in a FB-Type library-group.

5.3 Control-Flow Activity FB-Types

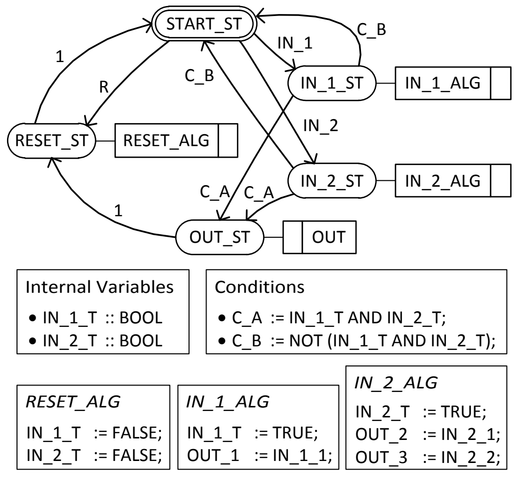
This section presents generation rules for Control-Flow Activity FB-Types that are mainly generated as Basic FB-Types. They represent instances of the eight different CFA types of the introduced Workflow Modelling Language, within a generated FB control application. Observe that the generation of the Basic FB-Type's ECC is predefined for every CFA type (i.e., parts of the behaviour "generation" are hard-coded). However, the *generation steps* explained in Section 5.2 are applied in the same sequence for the majority of CFA types.

5.3.1 REND FB-Type

The *REND* Control-Flow Activity synchronizes two incoming control-flow branches into a single branch [28]. This means, that the control-flow is passed to the outgoing branch, once both incoming branches have been enabled. Due to the synchronization functionality, also the incoming parameters of both incoming branches need to be merged into on combined control- and data-flow. Figure 5.2(a) shows the REND CFA, with examples of defined Activity Parameters. By applying the mapping rules of Section 5.2.2 to every input and output Activity Interface, the FB's interface can be generated. For the generation of the FB's behaviour as an ECC, the internal variables *IN_1_T* and *IN_2_T* are created at first. These are used as flags, to mark the occurrence of the input events *IN_1* and *IN_2*. Consequently, these variables are set to *true* if the corresponding intermediate EC-State (i.e., either *IN_1_ST* or *IN_2_ST*) is reached upon the event occurrence.



(a) Generated Function Block Interface



(b) Generated Execution Control Chart with algorithms and conditions

Figure 5.2: Generated Function-Block type for the Control-Flow Activity REND

Additionally to the flags, the values of the corresponding input data variables are assigned to the output data variables. However, these values cannot considered to be valid, before the output event *OUT* is triggered[2]. If both of the input events were active, the transition condition *C_A* evaluates to true and the EC-State *OUT_ST* is reached. Finally, the output event *OUT* is emitted and the data outputs are valid. The condition *C_B* evaluates to *true*, if only *one* of the input events has occurred. After the *OUT_ST* EC-State has been reached and finished its associated EC-Action, the *RESET_ST* is reached, via the outgoing EC-Transition, and the internal variables are set to *false*. Finally, the *Start* EC-State is reached again and the FB resides in its initial state. The resulting ECC and the generated algorithms are depicted in Figure 5.2(b).

With reference to the structure of the ECC, the conclusion can be drawn that, although internal variables are needed and corresponding algorithms in order to set those variables, the overall structure is kept simple. Therefore, the extension of the *REND* activity to more than two inputs, does not require additional generation effort. Extensions are realized by adding one state and one internal variable, corresponding to one event input including the necessary transitions. Furthermore, the algorithms as well as the conditions *C_A* and *C_B* need to be enhanced with the additional internal variables.

[2]According to Figure 2.3(b), the emission of the output event finalizes the execution of the invoked algorithm and states that the resulting data outputs are valid.

5.3.2 IF FB-Type

To enable the evaluation of a condition including *data values*, the notion of an *IF* CFA has been introduced. The values of the incoming data objects, provided by the incoming control-flows into the Activity Interfaces *IN_1* and *IN_2*, are used to express the if-condition. Activity input *COND* is used to specify the if-condition in Structured Text language notation. Figure 5.3 shows the generated FB-Interface as well as the ECC and corresponding algorithms.

With reference to the ECC drawing and the provided expressions, the condition is directly applied as label for the transitions, leading from the initial EC-State to the corresponding output states. The transition to the *ELSE_ST* requires the boolean inverted condition as a label. According to Section 4.1.2, the evaluation of the conditions *C_A* and *C_B* is only triggered on reception of the event *IN_1E*. In case the condition is satisfied upon request, the output event *THEN* is triggered, otherwise the *ELSE* output event.

5.3.3 SIGNAL FB-Type

Contrary to the other Control-Flow Activity FB-Types that have been presented in this section, the *SIGNAL* FB-Type is generated as a Composite FB-Type. According to Section 4.1.2, the *SIGNAL* CFA enables the reception of a signal (i.e., event) and

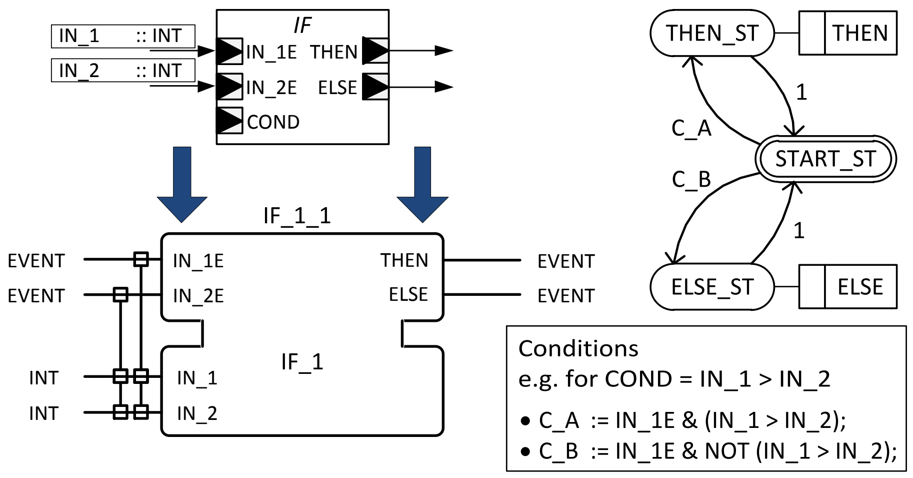


Figure 5.3: Generated Function Block type for the Control-Flow Activity *IF*

optionally a set of data values. The data types of the data values are specified using the *RCV_DATA* Activity Interface. Figure 5.4 depicts the generated FB-Interface for the concrete parametrization example, located in the uppermost part of the image. In this case, two data objects of the types *BOOL* and *INT* are received and passed to the corresponding data output variables *OUT_1* and *OUT_2*.

The internals of the generated Composite FB-Network depend on the value of the boolean *Constant Parameter UDP_TCP*. In case of the value *false*, the generated Composite FB-Network encompasses a *SUBSCRIBE* communication FB, which is the endpoint in a *UDP Multicast* communication service. Otherwise, events and data objects are received using a *CLIENT* communication FB, as endpoint in a *TCP Client/Server* connection. Additionally, some logic FBs are required to report the communication status. Using an UDP Multicast service results in the advantage that an event can be received by several subscribers. This can be useful if listeners to a single signal are needed in different sub-workflows.

For its service initialization, the corresponding FB-Type requires the *source connection endpoint ID (sourceID)* in order to register itself as communication endpoint, listening for messages sent from the defined source. Once a message is received, the output event *OUT* is activated and the data outputs are valid. In the case of a communication error the event *FAULT* is emitted.

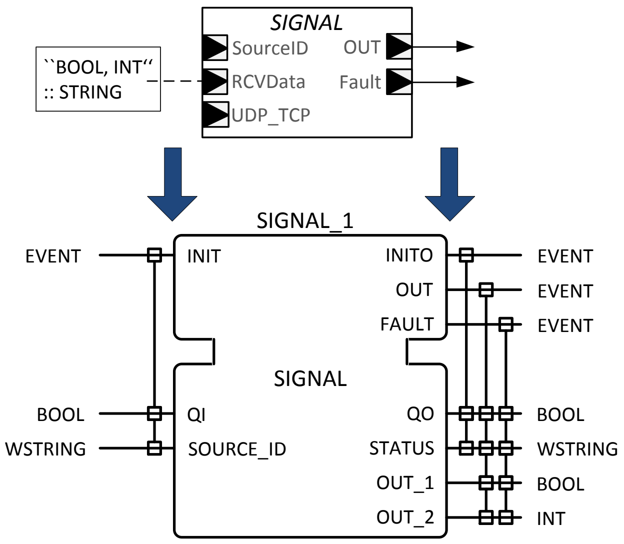


Figure 5.4: Generated Function Block type for the Control-Flow Activity *SIGNAL*

5.3.4 Remaining Control-Flow Activity Types

The generation of FB-Types for the remaining Control-Flow Activity types *MERGE* and *CALCULATE* is not presented explicitly in this section. This is because the generation procedure can be described analogous. The CFA type *WAIT* does not require a FB-Type to be generated. In [28] the FB-Type *E_DELAY* is presented as common type, usually available in the FB-Libraries of IEC 64199 compliant Engineering Environments (see e.g., [39], [17], or [21]). Assuming that this FB-Type is available, an instance can be created and added to the FB control application. The CFA type *START* represents the start-point of a Workflow Specification. With regard to the generation of the supervisory control application, a representing FB-Type has to be generated, which performs a supervised initialization of the FBs within the generated control application.

5.4 Service-Component FB-Types

As mentioned before, the main objective is the generation of a supervisory control application which coordinates the acting SCs. To make this possible, *interfaces* are needed for the control application, which enable the communication with the SCs. The communication is necessary to invoke Service-Functions (SFs), and to receive status information and processing results.

The first section deals with a general description of how interaction between SCs and the supervisory control application is arranged. Secondly, details on how the generation of Service-Component FB-Types is performed are provided. The generation and functionality of essential internal FB-Types in the SC FB's Composite Network are explained. Finally, the generation of the Composite Network is described.

5.4.1 Interaction of Service-Component and Control Application

For the following considerations it is assumed that a SC, including its implemented functionality as well as the behaviour information (i.e., declaration of SFs), is provided by the component vendor (see Section 3.1.1). The generated supervisory control application interacts with the SCs using an interface which has to provide the following functionalities:

- *Connection establishment to the SC.*

- *Permission and creation of SF invocation commands which need to be sent to the SC.*
- *Reception of processed return data of the SC and status information.*
- *Evaluation of status information.*

Regarding the required communication between SC and the control application, the assumption is made that the SC provides a *connection service* that is started once the SC has been initialized. This connection service listens to an incoming connection request from the supervisory control application. Once the connection has been established, a SF can only be invoked if the SC is ready to receive the next command. If a SC is currently executing a SF which had been invoked previously, it cannot accept an incoming command. Consequently, a SF invocation request can only be accepted and sent if the SC is not busy at the moment. In case of a SF invocation request has been granted, the associated command is constructed and submitted to the SC via the established connection. After the SC has processed the requested SF, status information and eventually processed return data is received by the interface. By evaluating the status information, the interface can report processing errors as well as received data to the control application.

The interface to a SC has been realized by generating a Composite FB-Type which is called the Service-Component FB-Type (SC FB-Type). Figure 5.5 depicts a schematic overview of the FB's Composite Network and illustrates the connection to the SC.

The required functionalities of the SC FB-Type are mainly realized by three internal FB-Types of the Composite Network. Firstly, the FB-Type located on the left constructs invocation commands and checks if the SC is ready. Service-Function invocation requests are represented by the event inputs ① and their associated parameters (i.e., input data) ②. Every event input ① represents one SF call, that is modelled in the Workflow Specification and associated to the SC. If the invocation request has been accepted, the SF parameter values as well as the SF's name are passed to the *communication FB* ③ and submitted to the SC. While the SC processes the received invocation command by calling the corresponding SF ④, it cannot accept further commands until the execution is finished. The processed data and SC status information is returned to the requesting SC FB ⑤ and forwarded to the next FB-Type which performs the status evaluation. Based on the returned status information which is defined according to Table 4.1, the FB-Type, located on the right, reports the successful or erroneous SF execution back to the supervisory control application. The corresponding events and parameter outputs ⑥, ⑦ are forwarded to the SC FB-Type's output interface. After the return data has been passed to the output, the execution of the invoked SF is considered to be done and

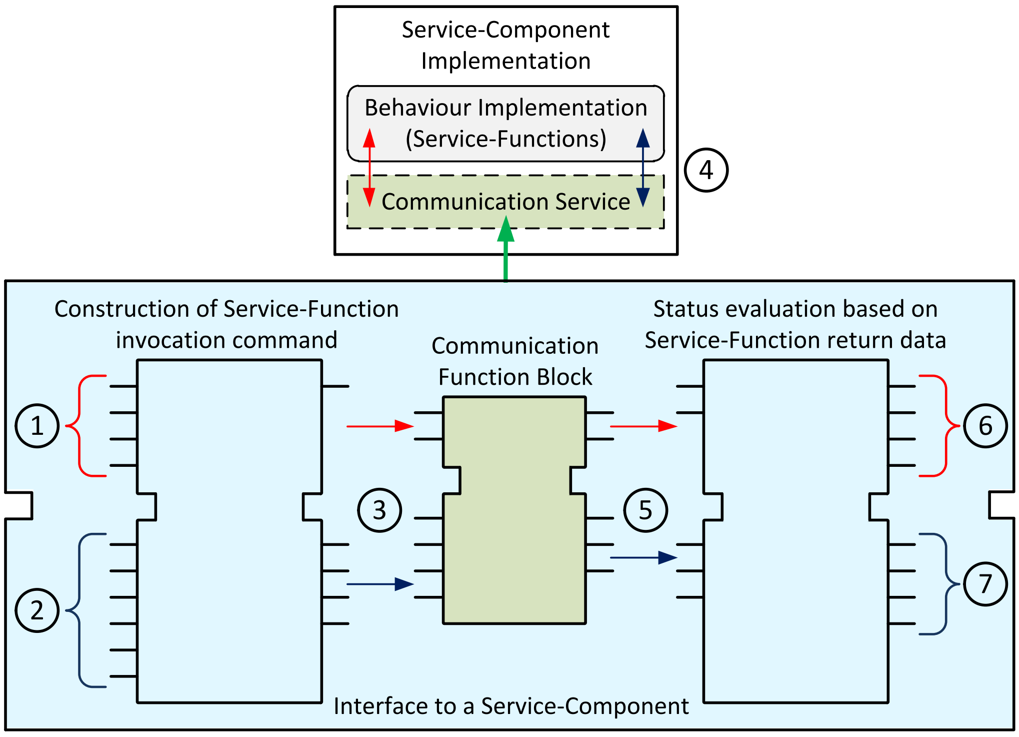

Service-Component Implementation
Behaviour Implementation (Service-Functions)
Communication Service
4
Construction of Service-Function invocation command
Communication Function Block
Status evaluation based on Service-Function return data
1
2
3
5
6
7
Interface to a Service-Component

Figure 5.5: Schematic overview of a Service-Component Function Block type and its interaction with the associated Service-Component

the SC is ready to receive the next command. Apart from the internal FB-Types introduced above, additional logic FBs are contained in the Composite Network to enable a *time-out* error, in case of a SC which does not respond within a defined time-span.

The following sections deals with the generation of the SC-FB's interface as well as functional description of the internal FB-Types which are depicted schematically in Figure 5.5 in the left and right image parts.

5.4.2 Generation of the Function Block Interface

As mentioned earlier, a Service-Component has to handle any Service-Function call that is associated with it and which are modelled as Service Activities within a Workflow Specification. Consequently, the generated SC FB-Type needs to represent an interface to the Service-Component, to enable the invocation of the SFs. To comply with these requirements, the interface of the SC FB-Type is generated in a way that every Service Activity instance which represents a SF call, is represented as an event input. This associated event is equally named as the corresponding Service Activity instance. Furthermore, since every SF call requires specific input parameters, data input variables are generated for every parameter and associated with the event input. This allows SFs of equal type to be called with different input parameter values.

Figure 5.6 depicts how the FB-Interface of a SC FB-Type is generated, based on the schematic Workflow Specification displayed in the left image half. In the following, the Service Activities *serviceA* and *serviceD* are considered to represent SF calls of a certain SC. The SF invocation events are equally named as the Service Activities, and their input parameters are generated by directly transforming the Activity Interface definition into a WITH-construct. Apart from the SF invocation events, the *INIT* event enables the initialization of the SC FB. The configuration parameter *ID* specifies the connection endpoint identification for the internal communication FB. Moreover, the data input *MAX_RESPONSE* specifies a timespan for the SC, to respond to an invocation request. A connection failure is reported to the control application via *CONN_F*, if the maximum response time is exceeded.

For the SC FB's output interface, two execution confirmation events with the associated output data variables are generated for every service invocation event. Depending on the result of the status evaluation that is performed by the FB-Type,

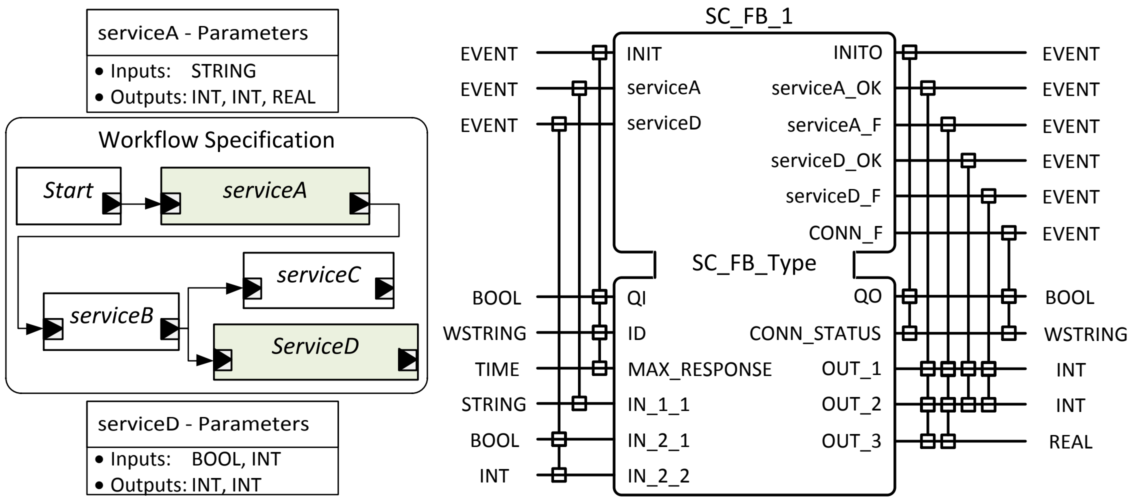


Figure 5.6: Generated interface of a Service-Component Function Block type, based on a modelled Workflow Specification

depicted in the right of Figure 5.5, the output event with either postfix *_OK* or *_F* is triggered. These postfixes can be directly compared to the "normal" output interface *Output* and to the *Fault* output of an Activity, respectively. Both confirmation outputs are associated with the same output parameters. However, in case of erroneous execution (i.e., the fault *_F* event has been triggered), the data outputs are not guaranteed to hold valid return data. The data output *CONN_STATUS* reports the connection status of the internal communication FB-Type.

With reference to the generated input data variables for the service invocation requests, the case of a number of activities referring to the same SF invocation shall be considered. In this case, input data variables, carrying the Service-Function input data values, need to be generated for every call of this Service-Function. As a reason for this, the FB input data variables do not support the *fan-in* or *multiplexing* of several incoming data connections. In contrast to the data inputs, only the minimum number of output data variable types, that is necessary to serve all Service-Function outputs, needs to be generated. This is because the IEC 61499 Standard supports the *fan-out* of data and events [1], [28]. This means that a data output, having a specific data type, can be used for several different execution confirmation events, sharing the same data type within their WITH-constructs.

Summarizing, three main design rules can be identified for the generation of SC FB-Interfaces. Firstly, every Service Activity instance results exactly in *one input event* that needs to be generated and represents the Service-Function invocation. Secondly, separate input data variables are needed for every SF call. Lastly, output data variables can be used for several SFs as data output and they are associated with both, the normal output event *_OK* and the error output event *_F*.

5.4.3 Elements of the Composite Network

This sub-section provides a detailed overview of the generated Composite Network which has been introduced earlier in this chapter. In the following, the internal FB-Types and their functionality is treated. Emphasis is put on the creation of service invocation commands, the passing of commands and parameters to the communication FB, and the interpretation of return data.

Creation of Service-Function invocation commands

Once a service invocation request is performed, the event and its associated input data is routed to the first FB-Type within the SC FB's Composite Network, which is called *CreateCommand*. The FB's behaviour is generated as an ECC, whose functionality can be summarized into four steps.

- *Check if the SC is ready to receive the next command.*
- *If the command may be sent, forward the command name and the data input values to the communication FB, and trigger the command transfer to the SC.*
- *If the command cannot be sent, the invocation request is dropped and an error is reported to the control application.*
- *After the execution of the SF has been triggered, wait for the SC to enter its idle state (ready) again.*

Referring to the first step, an intermediate state within the ECC is used which can only be reached if a corresponding *READY* event is received, at a fixed event input of the *CreateCommand* FB. If the command may be sent, the data values are forwarded to the output interface of the *CreateCommand* FB-Type. To enable the correct identification of the command through the SC, the command's name needs to be sent, additionally to the input parameter values. The relations explained above are depicted in Figure 5.7.

Data passing between the internal Function Blocks

In the next step we discuss how SF invocation commands (i.e., invocation event and associated parameters) and return values are passed between the affected FBs. At first, the interface sides of the communication FB as well as the right interface side

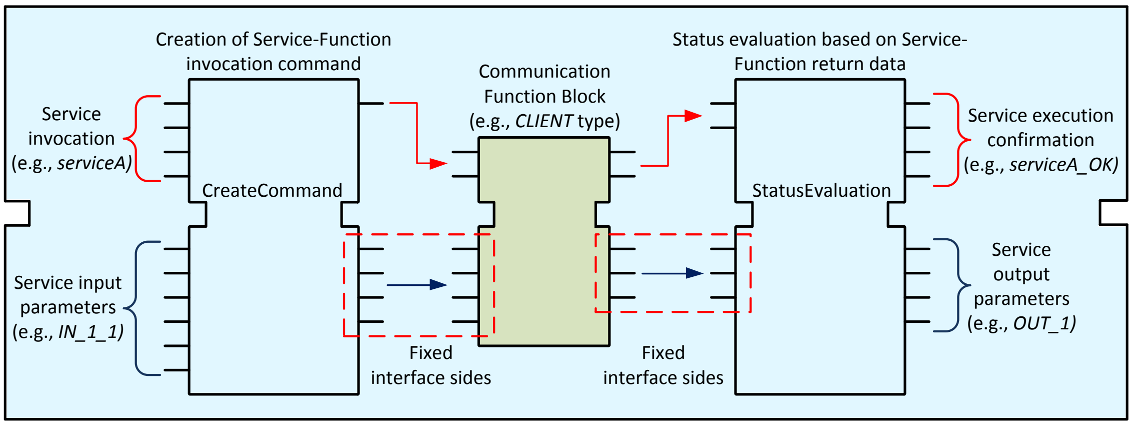

Creation of Service-Function invocation command
Communication Function Block (e.g., *CLIENT* type)
Status evaluation based on Service-Function return data
Service invocation (e.g., *serviceA)*
CreateCommand
StatusEvaluation
Service execution confirmation (e.g., *serviceA_OK)*
Service input parameters (e.g., *IN_1_1)*
Fixed interface sides
Fixed interface sides
Service output parameters (e.g., *OUT_1)*

Figure 5.7: Interaction between internal Function Blocks of the Service-Component Function Block type

of *CreateCommand* FB and the left interface side of the *StatusEvaluation* FB, are considered.

According to Figure 5.7, the listed interface sides do not vary with a changing number of SF calls that are modelled in the Workflow Specification. This means that the generation of these interface sides does *only* depend on the defined Service-Component behaviour information. Therefore, these interface sides are generated such that several defined SF invocations can be handled. Consequently the minimum number of data variables, needed to carry the associated parameter values of the defined SFs, have to be generated. The major advantage of this approach results in the fact that the interface of the communication FB is fixed already at design time. Moreover, the *communication service* of the implemented SC can be realized platform independently, but needs to be able to receive the passed data values with the predefined data types. The currently implemented modelling approach builds on a TCP connection service between SCs and the generated SC FB-Types. The SC FB's Composite-Network contains a *CLIENT* FB-Type which establishes a connection to the SC's internal *TCP server*.

Evaluation of the returned status value

After the SC has processed the invoked SF, the produced return data and status information is passed to the *StatusEvaluation* FB. The *StatusEvaluation* FB requires the command name as additional data input, in order to trigger the correct execution confirmation events (see Figure 5.6). This information is directly received from the *CreateCommand* FB.

The SC's status information, having the data type *integer*, is returned for every called SF, and is determined as the first return parameter of the SF. Depending on the returned status value, the *StatusEvaluation* FB either triggers the positive execution confirmation event with name postfix *_OK*, or the error execution confirmation event *_F*. A special case is given for connection errors, which are additionally reported by triggering the output event *CONN_F*. We distinguish between two types of connection errors. Firstly, the communication FB can report a connection fault by setting its event output qualifier *QO* to *false*. Secondly, in case of the response time of the SC exceeds the determined *MAX_RESPONSE* time span, a connection *time out* error is assumed. If no connection error has been detected a predefined output event is looped back to the *CreateCommand* FB, to signal that the SC is ready to receive the next service invocation request.

Besides the status evaluation, the received processing data is written to appropriate data outputs. With reference to the generation of the SC FB's interface, it has been stated that only the minimum number of data output variables is generated. These can be used for all modelled SF invocations of the Workflow Specification associated with the considered SC. When writing the return data to the outputs, the first appropriate data output is used for each output parameter type. For example, consider that *two* data outputs of type *BOOL* are available at the *StatusEvaluation* FB's output interface. In case of a SF which only returns one boolean value, the first data output which matches the required data type is used. The output interface of the *StatusEvaluation* FB basically corresponds to the output interface of the SC FB-Type. This means that the output events as well as the data outputs are forwarded to the output interface of the SC FB (see Figure 5.7).

Based on the generated FB-Types *CreateCommand* and *StatusEvaluation*, the SC FB's Composite FB-Network can be generated by creating the FB instances, as well as data and event connections between them.

5.4.4 Generation of the Composite Function Block Network

By using the generated FBs *CreateCommand* and *StatusEvaluation*, in combination with an appropriate communication FB-Type (here: *CLIENT* FB-Type), the Composite FB-Network of the SC FB can be generated. Apart from these FB-Types, which represent the major behaviour of the SC FB, some logic FBs are required for instance to implement the *time-out* functionality. The generation of the Composite FB-Network is summarized in the processing steps, described below.

In the first step, instances of the generated FB-Types as well as a *CLIENT* FB, an *E_DELAY* FB and an *E_REND* FB are created[3]. Afterwards, the event and data connections between the FB instances *CreateCommand*, *CLIENT* and *StatusEvaluation*, as well as the connections between the *interface sides* of the Composite FB and the FBs *CreateCommand* and *StatusEvaluation*, are established.

Especially, for the connections between the left and right hand interface sides of the Composite FB and the neighboured, generated FBs, we can make use of the equally named input and output events and their associated WITH constructs. Making use of this advantage, the connections between these FB instances and the Composite FBs interface are created straightforward. The connections for the time-out logic, the initialization event chain, and remaining connections between the internal FBs are predefined and equally to connect for every SC FB-Type. Finally, the generated

[3]The FB *E_DELAY* is used to delay the propagation of an event for a specified time-span. *E_REND* FBs allow the synchronization of two incoming events [28].

Composite FB-Network is bound to the SC FB-Type and the FB-Type is stored within a group of the FB library, of the IEC 61499 Engineering Environment.

5.5 Resulting Control Application

This section deals with the generation of a *supervisory Control Application* according to IEC 61499 that coordinates (external) Service-Components in order to perform the desired process. Once the FB-Network has been generated, it can be downloaded to a hardware resource and run afterwards. To enable the iterative improvement of a Workflow Specification, the supervisory control application can be tested against the 3D-Simulation environment.

5.5.1 Creating FB-Type Instances

The generation procedure is started by generating the required FB-Type library-group like discussed in Section 5.2. In the next step, the needed FB instances are created. The FB instance names correspond to the SC's name, in case of SC FBs, whereas FB instances that represent CFAs are named after the CFA instance in the Workflow Specification. Regarding the *START* CFA, an initialization FB-Network needs to be generated and connected to the first SF invocation event. This section of the resulting control application is executed after the target resource has been started. The initialization network enables a supervised initialization of SC FBs and other FB instances which require to get initialized. A description of the initialization concept is provided later in this section.

5.5.2 Event Connections and Data Connections

The control functionality of the FB-Network is generated by creating the necessary event and data connections between the created FB instances. Hereby, the information that is provided through the modelled Activity Connections of the Workflow Specification is used and extracted.

By considering an Activity Connection, the source and target activities are always accessible. Due to the naming conventions mentioned above, the generated source and target FB instances can be accessed via the activities' *instance names*, in the first step. Secondly, the source Activity Parameters (APs), the destination APs and the *Parameter Mapping* information of the destination activity, are used in order to find the corresponding connection endpoints of the representing FB instances. This means that the corresponding source/destination events and data variables of the affected FB instances, that shall be connected, need to be found. Again the *naming conventions* that were introduced for the generation of FB-Types are used. In case of a Service Activity, the input events as well as the output events are named after the activity.

Based on the determined source and destination events, the corresponding data outputs and data inputs are obtained by utilizing the event↔data *associations* that are given by the *WITH constructs*. If all the event and data inputs and outputs could be found, the event and data connections can be created by connecting the paired events and data ports, respectively. The remaining connections, that need to be established, are necessary to couple the initialization Network section to the FB-Network representing the designed workflow.

5.5.3 Initialization of the Function Block Network

Service-Component FBs as well as the *SIGNAL* FB-Types require to be initialized, because these FBs need to set up communication interfaces. An initialization mechanism is needed which supervises the initialization procedure. The execution of the control FB-Network can only be commenced, if all FBs have been initialized successfully.

In the following the generation of a *SC_INIT* FB-Type (Service-Component Initialization) is explained, which is realized as a Basic FB-Type. Concerning the FB-Interface definition, one output event is generated for every generated SC FB-Type. Moreover, these events are associated with a boolean output qualifier *QO_x*, which is connected to the corresponding input qualifier *QI* of the SC FB "*x*". The output qualifier *QO_x* is set to *true* if the corresponding SC FB shall be initialized. The left interface side of the *SC_INIT* type contains one boolean input qualifier *QI_x* for every SC FB-Type. These are used to receive an *initialization feedback* from the concerned SC FB-Types. All of those input data variables are associated with a single response input event *RSP*. This event is triggered for every initialization response which is emitted by the SC FB-Types.

The initialization procedure is triggered using the input event *INIT*. An initialization event is emitted for every SC FB-Type with the associated qualifier *QO_x* set to *true*. If a positive input qualifier *QI_x* could be received for every SC FB-Type, the FBs are considered to be initialized correctly and the output event *INIT_OK* is emitted. Figure 5.8 depicts a schematic overview of the initialization FB-Network, where two SC FB instances are initialized. Apart from the *SC_INIT* FB, instances of the FBs *E_RESTART* and *E_REND* are created. The *E_RESTART* FB emits an event once the corresponding *resource*, to which the FB is mapped, is *started* (i.e., *Cold* or *Warm* start event output) [28]. The *E_REND* FB synchronizes the *start* event from *E_RESTART* and the *INIT_OK* event. Moreover, the resulting output event is forwarded to the first *service invocation* event, of the first SC FB that is affected, according to the Workflow Specification. An example for a generated supervisory control application is provided in Chapter 7.

5.5.4 Execution of the Control Application

Once the control application has been generated, an IEC 61499 compliant Engineering Environment for FB-Applications is used to map and deploy the supervisory control application to distributed resources. These resources support the runtime environment needed to execute the FB network. In order to test the overall behaviour and further improve the Workflow Specification, the FB runtime environment can also be run coupled with the 3D-Simulation Environment. This requires the corresponding Service-Component implementation to include functionality to send signals and data values to the 3D-Simulation Environment. Moreover, the modelled component in the 3D-Simulation Environment, requires behaviour infor-

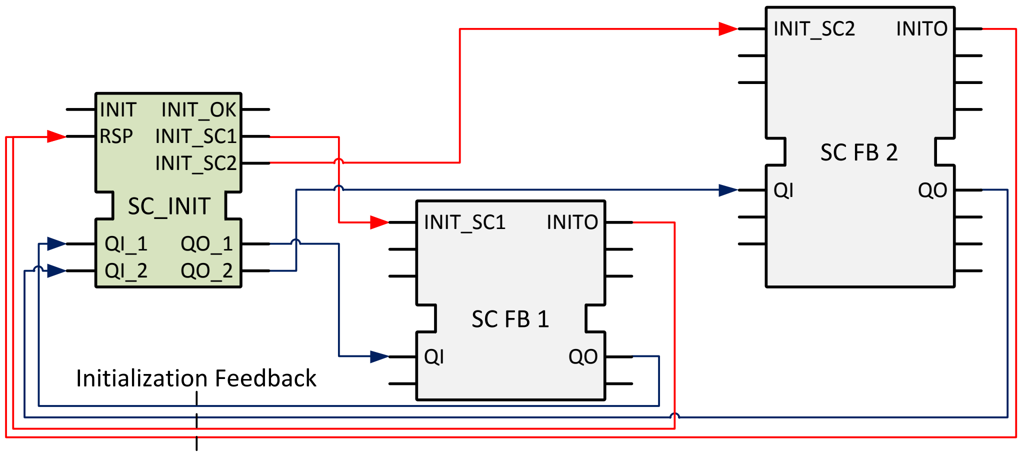


Figure 5.8: Initialization of Service-Component Function Block types using the *SC_INIT* FB-Type

mation that is directly associated with it. This kind of behaviour description shall not be mixed up with the description of Service-Functions. Examples include defined robot kinematics or defined, moving parts like a conveyor belt. Upon an appropriate signal is received by the 3D-Simulation Environment, the simulation object performs the associated action.

In the current implementation of the code generation approach, the self-implemented Service-Components need to be extended manually in order to support the connection to the 3D-Simulation Environment. The improved integration of the 3D-Simulation Environment into the workflow modelling process as well as the code generation process is planned in the further development of this engineering approach.

5.6 Concluding Remarks

A methodology for the automatic generation of an IEC 61499 compliant, supervisory control application, based on a Workflow Specification, has been presented. As explained, the generation of the resulting FB-Network, requires a set of FB-Types that need to be generated in advance and stored within a FB library-group. Section 5.2 describes how representative Function Blocks are generated for Service Activities and Control-Flow Activities.

The conclusion can be drawn that the majority of generated FBs are Basic FB-Types, whose Execution Control behaviour is determined by an Execution Control Chart (ECC). The generation steps for the automatic creation of an ECC, are very generic and can be applied to several Control-Flow Activity FB-Types. They are also applied for the generation of the internal FBs in the Composite FB-Network of a Service-Component FB.

Service-Component FBs are generated as Composite FB-Types and act as interfaces for the supervisory control application to a Service-Component. The Service-Function invocation commands are sent to the Service-Component via a CLIENT/SERVER connection. Although the method of using one FB to represent all Service Function invocations, corresponding to a certain SC, seems to be intuitive, the *input interface* sides of the SC FBs tend to grow very large with a rising number of Service-Function calls. A possible solution to this problem could be achieved by using *Adapter Function Blocks*, by shifting the complexity of the input interface into the Composite-FB.

Useful naming schemes for event and data inputs and outputs, simplifies the generation of event and data connections during the creation of FB-Networks. In that way, events of the FBs can be directly referenced from the modelled Activity Connections and their source and destination endpoints. Data inputs and outputs are then referenced using the *WITH constructs*, that associate events with data variables.

The generated FB-Network needs to be initialized in advance, before its execution can be commenced. For this purpose, a special Basic FB-Type is generated which performs the initialization of the Service Component FBs.

6 Implementation

This chapter deals with an overview of how the described engineering approach is implemented in the Eclipse-based 4DIAC Integrated Development Environment [17]. The implementation and integration of a *graphical* Workflow Modelling Editor as well as the described *code generation* approach, is treated.

The first section gives an overview description of the used frameworks and tools. In the second part, the implementation of the graphical *Workflow Modelling Editor* based on the *Graphical Editing Framework (GEF)* is treated. In the third section, the basic structures of the *code generator* are explained.

6.1 Base Frameworks and Tools

In this section, a brief overview of the frameworks that are used for the implementation is given.

4DIAC Integrated Development Environment

The open-source initiative *4DIAC* (Framework for Distributed Industrial Automation and Control) aims to develop a toolchain for the design and execution of IEC 61499 compliant control applications, which consists of the two software projects *4DIAC-IDE* and *FORTE* [17].

The *4DIAC-IDE* (4DIAC Integrated Development Environment) is an engineering environment for distributed control applications according to the IEC 61499 Standard. Based on the plug-in structure of the Eclipse Framework, the application is extensible through newly implemented plug-in features. For the implementation of custom FB-Types, the 4DIAC-IDE includes a *FB-Type Editor*, a *FB Tester* as well as a *FB-Network Editor*. A *System Configuration Editor* enables the configuration of the distributed hardware devices. Once a FB-Application has been modelled, and the FBs have been distributed to the devices, the application can be downloaded to the resources and executed. The second project of the 4DIAC initiative, the *FORTE* (4DIAC Runtime Environment), allows the execution of FBs on the target devices on which the FORTE is run. Implemented in the programming language C++, the FORTE supports different platforms including Linux, Windows and embedded systems [17]. The 4DIAC toolchain forms the basis for the developed Workflow Modelling Editor and the code generator described in this thesis.

The Eclipse Modelling Framework (EMF), described in Section 2.5.4, has been used to generate the base plug-in structures for the Workflow Modelling Editor as well as for the code generator, based on EMF *Ecore* models.

Graphical Editing Framework

The *Graphical Editing Framework (GEF)* is built on the visualization framework *Draw2D* and enables the graphical representation of model objects within an editor, that are defined in a data model [37], [27]. GEF makes no restrictions on the type of the used data model. However, the only requirement for the model is to provide mechanisms which enable the *notification* of the GEF framework, in case of model object changes. For the implementation of the graphical Workflow Modelling Editor, the concepts of GEF, combined with the given EMF Ecore model, have been applied in order to transport user requested changes to the underlying model.

Connectivity between the used Frameworks and Tools

In the following, the interconnection between the base frameworks, tools as well as the implemented Workflow Modelling Editor and code generator plug-ins is explained. Figure 6.1 shows the relations between these components. The Workflow Modelling Editor builds on the defined EMF Ecore model which basically represents the language elements of the Workflow Modelling Language. A graphical representation of the workflow model is realized by using the methodologies of the GEF framework. The GEF functionalities are used to transfer change requests from the

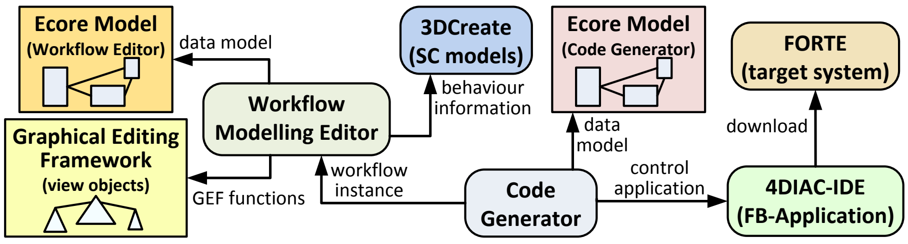


Figure 6.1: Relations between the used frameworks and tools

editor's user interface (i.e., the graphical representation through *view objects*), to the underlying data model. By accessing the Service-Component behaviour description, that is modelled in 3DCreate, the Workflow Modelling Editor creates a palette of the available Service Activities. After defining the workflow model, the code generator accesses the workflow instance and generates the supervisory control application, which is added as new project to the 4DIAC-IDE workspace. The code generator's plug-in structure builds on an Ecore model. For the execution of the generated control application, the 4DIAC-IDE is used to map the FB-Network to the desired target resource. By using the deployment functionality, the application is downloaded to the target system. The control application is executed through the FORTE, which is running on the device.

6.2 Workflow Modelling Editor

The Workflow Modelling Editor is implemented as a composition of three plug-ins which have some dependencies on the plug-in structure of the 4DIAC-IDE plug-ins. In the first part of this section, the structure of the *EMF Ecore* data model is presented. After that, the *plug-ins*, containing the source code for the Workflow Modelling Editor, which were partly generated by the EMF code generation tools, are explained. Finally, an overview description of the implemented code, including important functions and classes, is given.

6.2.1 EMF Ecore Model

An EMF *Ecore* model has been constructed that is used to automatically generate the model-representing Java classes. To make the data model suitable for the combination with *GEF* special `View` objects are used, which represent the current model state. The `View` objects are associated with an appropriate `EditPart`, which acts as a bridge element between the model object and its graphical representation, given by a figure object of type `IFigure`. Figure 6.2 shows the constructed Ecore data model which is the basis of the Workflow Modelling Editor plug-ins.

A `Workflow` consists of at least one `Activity` that can be of `ControlFlowActivity` type or of `TaskActivity` type[1]. `Activities` refer to exactly one `ServiceFunction`, which itself is referenced by a `ServiceComponent`. `ActivityConnections` are used

[1]The `TaskActivity` type corresponds to the Service Activity type which is introduced in Section 3.4.1.

to connect Activities together, using their input and output `ActivityInterface Elements`. For the generation of the control application, it is important that the `Workflow` refers to the `ActivityConnections`, to make them easier accessible. In order to allow the mapping of *ActivityParameters* of a source activity to input ActivityParameters of a target activity, an `ActivityParameterMapper` is used.

With regards to the notations used in Figure 6.2, the entities with light grey background denote model objects which are associated with a specific `EditPart`. Opened arrows describe references in the given direction. Generalized model objects, as well as Interfaces are denoted with closed arrows. A filled diamond denotes, that this model objects provides a *containment* for the model object on the opposite end of the arc. Providing containments in general allows the *serialization* of the model objects [13]. Entities having dependencies on existing 4DIAC plug-ins are separated with "::" from the parent plug-in package.

Using the *code generation* framework part of the EMF, Java class and interface definitions can be created automatically by configuring a so-called *genmodel*. The generated code includes *factory classes* which provide members to create instances of the modelled entities.

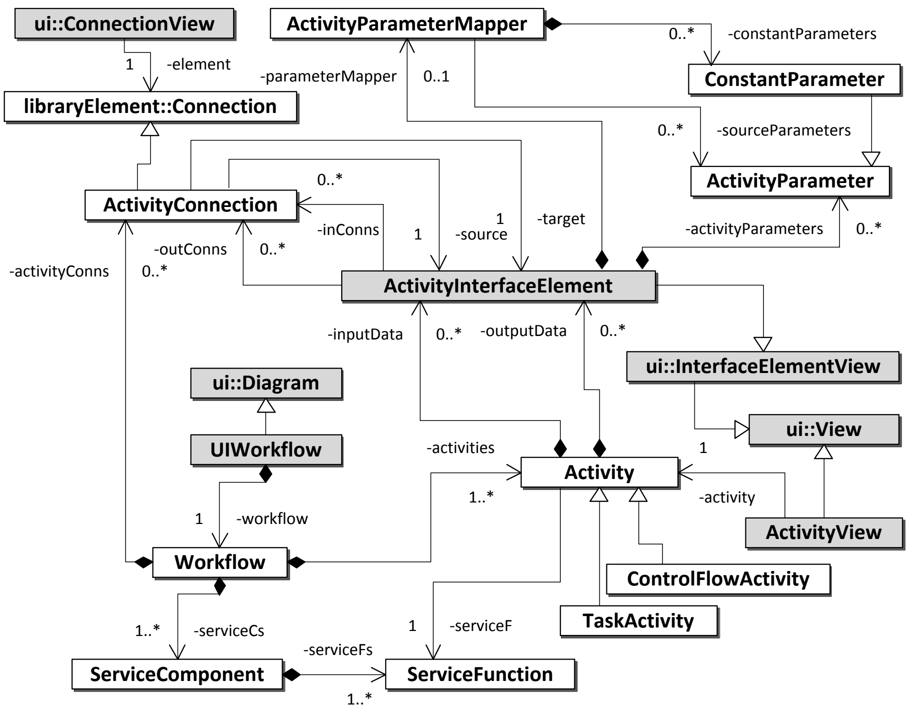


Figure 6.2: Ecore Data Model for the Workflow Modelling Editor

6.2.2 Plug-In Structure of the Workflow Modelling Editor

This section deals with a brief overview of the plug-in structure for the realization of the Workflow Modelling Editor. The basic contents, as well as the functionalities, that are provided by the plug-ins, are explained.

The plug-in `workflow.model` includes *Java classes* and *interfaces* representing the model objects of the Ecore model, as well as the `WorkflowmodelFactory` class that is used to create object instances.

The major functionality of the *Workflow Modelling Editor* is implemented within the classes of the `workflow.editor` plug-in. The root package `workflow.editor`, contains classes to enable start up and configuration of the *WorkflowEditor*. An `ActivityFactory` creates the activity instances which should be dropped into the editor diagram window. Command implementations, to *create* or *delete* activity instances or activity connections, are covered by the `workflow.editor.commands` package. The plug-in package `workflow.editor.editparts` contains implementations of the various `EditParts`. Model object representatives are *Draw2D* based figures of type `IFigure` and implemented as classes within the package `workflow.editor.figures`. Finally, the package `workflow.editor.policies` contains the implementation of *Edit Policies*, which are installed on *Edit Parts* and utilized to create `Command` instances upon a model change request.

6.2.3 Implementation Description

The Workflow Modelling Editor is implemented as a plug-in project, that is included into the 4DIAC-IDE. In this section, the implementation of important classes is described briefly.

Class Diagram of the Workflow Modelling Editor

A general overview of the involved classes and their relations together, is given in Figure 6.3, which shows a simplified class diagram of the editor.

As the Workflow Modelling Editor consists of one graphical editor, the `Workflow Editor` can be considered as the main editor. It is initialized based on its *model source*, the `UIWorkflow`. According to Figure 6.2, the `UIWorkflow` extends the `Diagram` class and refers to several child `View` objects that represent ActivityConnections and Activities. The `WorkflowEditor` specializes the `DiagramEditor` which

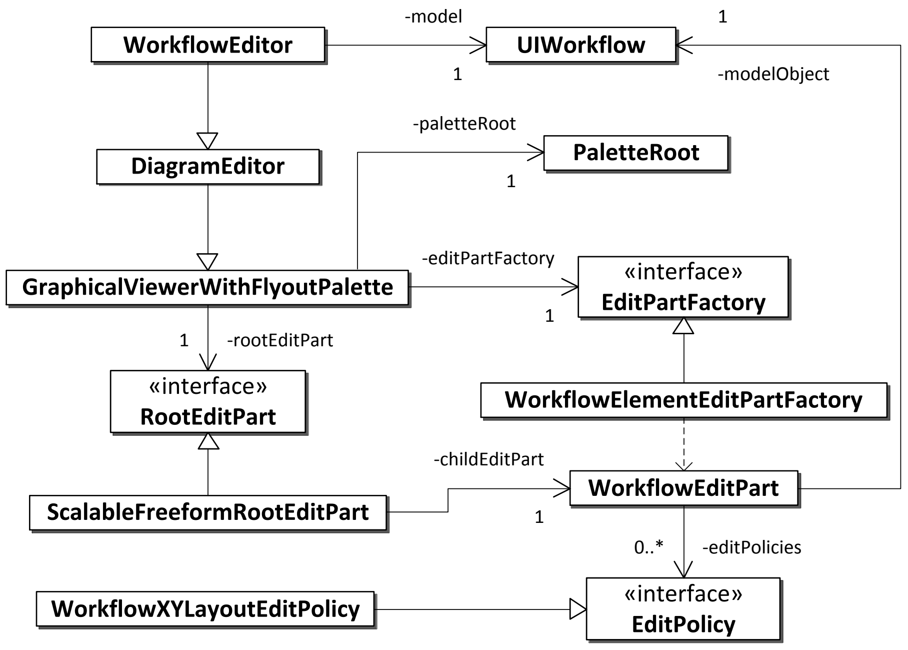


Figure 6.3: Simplified Class Diagram of the Workflow Modelling Editor

further extends the `GraphicalViewerWithFlyoutPalette`. This is an editor type, that provides a *palette window* containing `PaletteDrawer` entries. Every entry represents an activity type that can be used for modelling a workflow. The whole palette has got a root element, the `PaletteRoot` that is required for the initialization of the editor.

Every *GraphicalViewer* has got one `RootEditPart` which corresponds to the highest levelled `EditPart` in the hierarchy of *EditParts*. For example the `Scaleable FreeformRootEditPart`, implementing the `RootEditPart` interface, allows *zoom operations* to be performed on the editor window. The `WorkflowElementEditPart Factory` creates `EditPart` instances for the model objects `UIWorkflow`, `Activity View`, `ActivityInterfaceElement` and `ConnectionView`. An `EditPolicy` is required by an `EditPart` in order to create *commands* which are executed to change the model objects. Considering the case of the `WorkflowEditPart`, the `WorkflowXY LayoutEditPolicy` is installed which creates command instances of the types `Create ActivityCommand` and `SetPositionCommand`. The `CreateActivityCommand` creates an activity instance at the position specified in the editor window. Existing activities can be moved within the editor window by executing a `SetPositionCommand` which only affects the *View* object of the activity.

The class diagram in Figure 6.3 depicts the most important relations and dependencies of the `WorkflowEditor` class and does not show the entire class structure. For instance, the additionally required `EditPart` classes are not shown.

Description of Essential Functionalities

As mentioned in Section 3.4.1, the Workflow Modelling Editor has to import the defined behaviour information of the Service-Components, from the *3D-Simulation Environment*. For this workflow modelling approach, the 3D-Simulation Toolkit *3DCreate* [64] has been used in combination with the Workflow Modelling Editor plug-ins. *3DCreate* offers an *Application Programming Interface (API)* that is based on the *Microsoft Component Object Model (COM)*[2] [35] and is accessed using the *com4j* libraries [45]. These libraries contain Java-based wrapper classes, for the interaction of Java applications with COM objects.

The function `create3DCreateElementsDrawer`, uses the COM access plug-ins in order to connect to the *3DCreate* application. Within the 3DCreate COM API, the Service-Components are referred by COM objects and consequently, their behaviour information can be read by an *XML parser*.

In order to enable *parameter mapping* for the Workflow Modelling Editor, a *property view* has been realized within the `InterfaceElementEditPart` class, by implementing the `IPropertySource` interface. The property view is opened when a *ActivityInterfaceElement* is selected in the graphical editor. Further classes and functions of the plug-in `workflow.editor` are implemented in compliance with the *GEF* concepts (see [37] and [27]).

6.3 Code Generator

The generation of an IEC 61499 compliant control application is performed by a *code generator* that has been implemented as Eclipse plug-in. This section deals with a description of this plug-in, which depends on the `workflow.editor` plug-in structure as well as the 4DIAC-IDE plug-ins. In the first part, the Ecore model of the code generator is explained. The second part covers the implementation of relevant classes, including important member functions.

[2]Microsoft COM allows the development of software components which can interact, by using *COM objects*. The data of COM objects can be accessed by using predefined function members, that belong to *COM interfaces*, across process borders and even computer device borders [35].

6.3.1 EMF Ecore Model

Similar to the implementation of the Workflow Modelling Editor, an Ecore data model has been used to generate the basic plug-in structure, including class and interface definitions, for the code generator. The generated source code has been modified and further classes were added manually. Figure 6.4 depicts the base Ecore data model of the generator plug-in, which does not include manually added classes. However, it contains enough information, necessary to explain the functionality of the code generator.

The model object `AutomationSystemGenerator` can considered to be the central element of the Ecore model. It refers to an existing `lib::AutomationSystem` object and to the previously defined Workflow Specification `workflow.model::Workflow`. An `AutomationSystem` is defined within the 4DIAC plug-in structure and may encompass several independent FB-Applications, a FB-Type library, as well as a *System Configuration* file which stores the configuration of the distributed hardware devices. The `ControlFBNetworkGenerator` refers to the same objects of the AutomationSystem and the Workflow, as the AutomationSystemGenerator does. Activities need to be "transformed" trough the generation of representing FB-Types. This is performed by `WorkflowActivityTransformers` which can either be of the

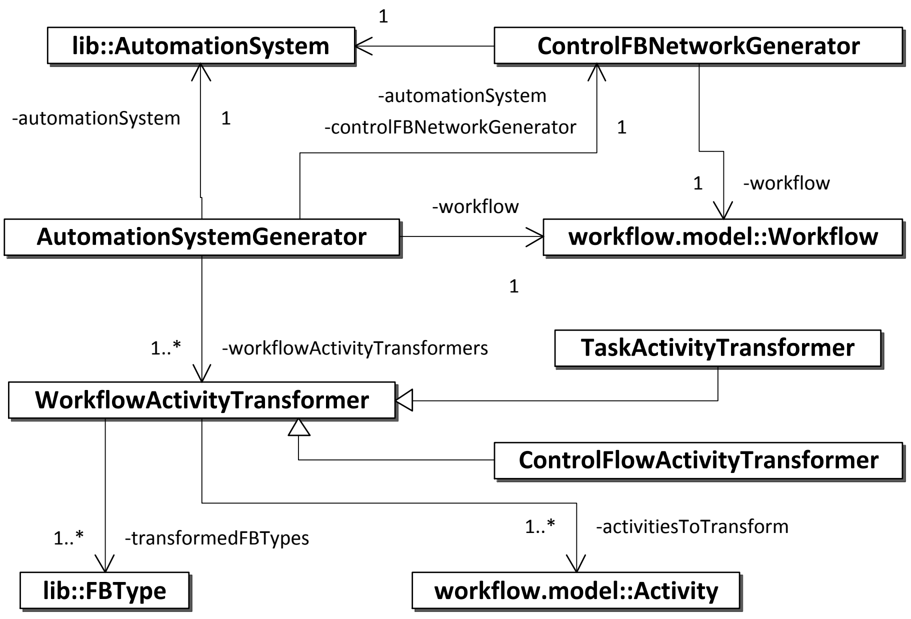


Figure 6.4: Ecore Data Model for the code generator

types `TaskActivityTransformer` or `ControlFlowActivityTransformer`[3]. The activity transformer types refer to a list of `workflow.model::Activities` and produce a list of generated `lib::FBTypes`, during the execution of the member function `transform()`.

6.3.2 Plug-In Structure of the Code Generator

The code generator is implemented as a single plug-in project, that is included into the 4DIAC-IDE plug-in structure. In the following, an overview of the plug-in packages and their contents is given.

The package `workflow.generatormodel.*` encompasses the class and interface definitions resulting from the Ecore model. Furthermore, similar to the Workflow Modelling Editor plug-ins, the `GeneratorModelFactory` enables the creation of instances of the generated classes. For the concrete implementation of the generator functionalities, some of the generated classes are modified accordingly. In order to establish data or event connections, the package `workflow.generator.commands` contains the classes `FBDataConnectionCreateCommand` and `FBEventConnectionCreateCommand`. To create a connection, an instance of the desired command needs to be created and, for its execution, the method `execute()` has to be invoked. Package `workflow.generator.eccgen` contains a set of *ECC generators*, each enabling the generation of a Basic FB-Type. The generated Basic FB-Types mainly represent Control-Flow Activity instances, which have been modelled in the Workflow Specification.

6.3.3 Implementation Description

The class *RunAutomationSystemGenerator*, providing the equally named *action*, starts the generation process by calling the function `run()` once the action is activated via the workbench window. A new `AutomationSystem` object is instanced as well as an `AutomationSystemGenerator` object. This method requires an active *WorkflowEditor* diagram to be currently opened, because the workflow instance `Workflow` needs to be referenced by this method. The method `run()` of the class `AutomationSystemGenerator` performs the code generation process, which has been described before in this thesis. For every SC which is used in the Workflow

[3]The `WorkflowActivityTransformer` is generated as *abstract* class, thus forcing the derivation of sub-classes.

Specification, as well as for every *Control-Flow Activity* instance, an appropriate `WorkflowActivityTransformer` instance is created in this context.

Assuming that a FB library-group has been generated correctly, a `ControlFBNet workGenerator` instance can be created. Once the method `run()` is called, the FB-Application is generated according to the described approach. For the generation of the event and data connections of the control application, the `ActivityConnections` are referenced via the `Workflow` object according to Figure 6.4 and passed to the function `establishEventAndDataConnections`. The generated control FB-Network is stored as *Application* of the *AutomationSystem* object and is available in the *System Manager* perspective of the 4DIAC-IDE.

6.4 Concluding Remarks

This chapter dealt with the implementation of the Workflow Modelling Editor and the code generator. The resulting plug-in structures have been integrated into the 4DIAC Integrated Engineering Environment.

In the first section, tools and frameworks have been presented which form the basis for the implementation of the Workflow Modelling Editor as well as the code generator. Moreover, the connectivity between the base frameworks and tools, as well as the implemented plug-ins, has been explained. The Eclipse Modelling Framework has been used in order to create the source code of the basic model objects, based on the designed Ecore data models. Combined with the Graphical Editing Framework, a graphical representation of certain data model objects has been implemented, which resulted in the development of a Workflow Modelling Editor plug-in.

The second part treated the implementation of the Workflow Modelling Editor, including a description of the underlying EMF Ecore model and the description of the plug-in structure. In the third part, the implementation of the code generator was presented. The essential classes and function of both plug-ins have been described briefly.

7 Verification and Test

This chapter summarizes the results of the developed approach which has been described in this thesis. An example application is presented, to show how the Workflow Modelling Editor can be used to program control behaviour. Moreover, the generated supervisory control application is described. Finally, the presented results are summarized and commented.

7.1 Human/Robot-Interaction Example

In the following, a Human-Robot-Interaction example is presented in order to show the appliance of the developed approach to a more complex process. Four Service-Components (SC) are involved in this workflow example, listed below.

- *5DOF AL5D Robotic Arm Combo Kit* from LynxMotion [32], controlled via *RS-232* serial connection.
- *Ultrasonic sensor* from the *LEGO Mindstorms* construction kit [26].
- *Touch sensor* from the *LEGO Mindstorms* construction kit [26].
- *LynxMotionHMI*, a self-defined software SC which calculates an appropriate *movement speed* value for the robotic arm.

In order to access the LEGO Mindstorms sensors, an existing *communication layer* for devices running the Robot Operating System (ROS) [49] is included in the run-time environment FORTE. The example process, that has been modelled with a Workflow Specification, can be described as follows. It is assumed that the ultrasonic sensor is mounted close to the border of the robot's working range and used to detect an approaching person. The robotic arm is moving a predefined path continuously at a given maximum speed. Once a person is located within the range of the ultrasonic sensor, the velocity of the robot's arm is reduced, depending on the measured distance. Furthermore, if the *touch sensor* is pushed, the robotic arm stops and the execution of the current movement is suspended until the touch sensor is released.

7.1.1 Important Service-Functions

This sub-section gives an overview of the services, which are provided by the Service-Components and used in this example application. The SC "UltrasonicSensor" provides the Service-Functions, listed below.

- SF `connect`: Needed to initialize the ultrasonic sensor, and requires the correct *ROS topic ID*[1], corresponding to the ultrasonic sensor, to be passed. Moreover, a multicast IP address and a port needs to be specified. This connection endpoint ID is required to initialize a publisher communication service that is used to send the sensor value.
- SF `getSample`: Used to subscribe sensor values periodically with the given *interval time*. The measured distance is returned as `LREAL`[2] value and can be received using a *SIGNAL* Control-Flow Activity instance.
- SF `stopSample`: Stops the periodic reception of sensor values, initiated by `getSample`.

The "TouchSensor" SC provides the same SFs as the ultrasonic sensor SC, but it returns a boolean value instead, denoting if the sensor is pushed (i.e., *true*) or not.

To control the robotic arm, the SC "LynxMotionRobot" provides two Service-Functions (SF) that are described below.

- SF `connect`: Initializes the serial *RS-232* connection to the robot controller, and initializes[3] the robot controller afterwards.
- SF `movePath`: Moves to the next position that is defined in a `REAL` type array of positions. Hereby, one position is provided as an aggregation of *six* `REAL` values. Every value corresponds to an absolute position value (e.g., angle) of one of the *five* robot axes and the *gripper*. A model of the robotic arm is shown in Figure 7.1. Parameter `minTime` (type `UINT`) is used to set the minimum time for the positioning command to last, in units of milliseconds. This value can be directly related to the velocity of the movement.
- SF `getStatus`: Returns the current status (e.g., "moving" or "stopped") of the robot as well as the current position values.

[1]Messages in ROS are routed based on a publish/subscribe methodology. A *ROS topic ID* identifies a certain message type which can be used by a subscriber to read the message contents [49].

[2]`LREAL` is a 64-Bit wide floating point value type, defined in the IEC 1131-3 Standard (see [28] and [1]).

[3]According to [32], the initialization of the internal absolute encoders of the robot's servo motors, is done by performing an initial movement of all axes as well as the gripper.

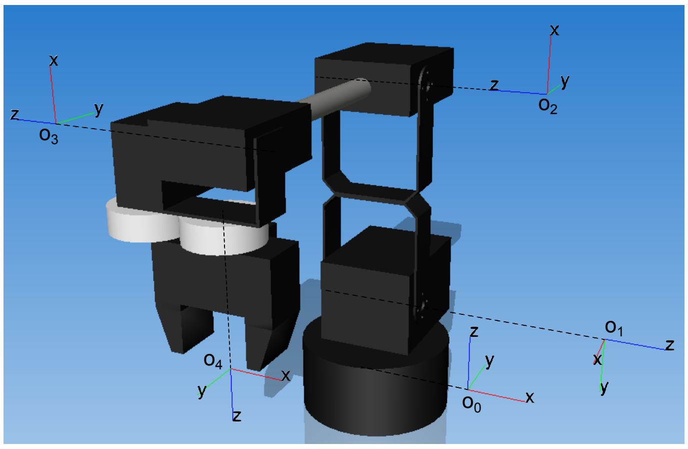


Figure 7.1: CAD model of the Robotic Arm with axis coordinate frames, shown in 3D Create [64]

- SF `stopAxes`: Cancels the current movement.

Finally, the SC "LynxMotionHMI" provides the SF `suggestedSpeed` which calculates an appropriate `minTime` value for the SF `movePath`. The calculation is based on the received distance value from the ultrasonic sensor device.

7.1.2 Design of the Workflow Specification

Based on the modelled SCs, the Workflow Specification is designed by firstly initializing the SCs "UltrasonicSensor", "TouchSensor" and "LynxMotionRobot". By applying the activity `getSample` for both sensor types, the periodic emission of sensor values is triggered. These values are received using two instances of the *SIGNAL* Control-Flow Activities. The received distance value is forwarded to the activity `suggestedSpeed` to calculate the *speed* value for the next movement. If the received boolean value of the touch sensor equals *true*, the current movement is stopped.

A movement is initiated using the activity `movePath`. In the next step a loop structure is needed to periodically check if the robot is still moving, using the activity `getStatus`. If the robot has stopped moving, the next position movement of the provided *path* is started. In case of the activity `movePath` reporting an error, the robotic arm has to be stopped. Moreover, to prevent additional invocations of `movePath`, the service `stopSample` is called for both sensor types. This procedure can be described as an exception handling branch of the Workflow Specification. Besides the mentioned activity instances a number of Control-Flow Activities is needed to synchronize or branch control-flows. The resulting Workflow Specification is depicted in Figure 7.2.

7.1.3 Resulting Control FB-Network

The generated supervisory control application, which coordinates the SCs in order to realize the desired process behaviour, is shown in Figure 7.3. With reference to the introduced code generation approach, the major complexity is hidden within the Service-Component FB-Types that are highlighted with green coloured stars. Several SC FB-Types are initialized by the *SC_INIT* FB, which is tagged with a grey coloured star. The initialization is triggered once the executing resource is started (see *E_RESTART* FB-Type). In order to execute the FB control application, the FB instances need to be mapped to a configured resource at first. Before the execution is initiated, the involved SCs have to be started in advance. Otherwise the SC FB-Types cannot establish a connection to their corresponding SCs.

7.2 Results and Comments

The presented example shows, that the designed Workflow Specification is less difficult to read than the generated FB-Application, which has a fairly complex structure. As pointed out, the main complexity is hidden within the generated FB-Types. Especially the SC FB-Types are of *Composite* type, having an internal FB-Network which contains generated *Basic* FB-Types. The complexity of the control application itself is given due to the high amount of data and event connections.

This simplified programming approach has been used in a real industry-oriented application, to program a control application in order to run an object recognition Service-Component. In the next development step, a bin-picking system composed of an object recognition system with a conventional industrial robot and a path-planning component shall be programmed. The first implemented examples show a significant difference between the engineering effort, required for the developed programming method and the effort that needs to be made when implementing an IEC 61499 control application directly.

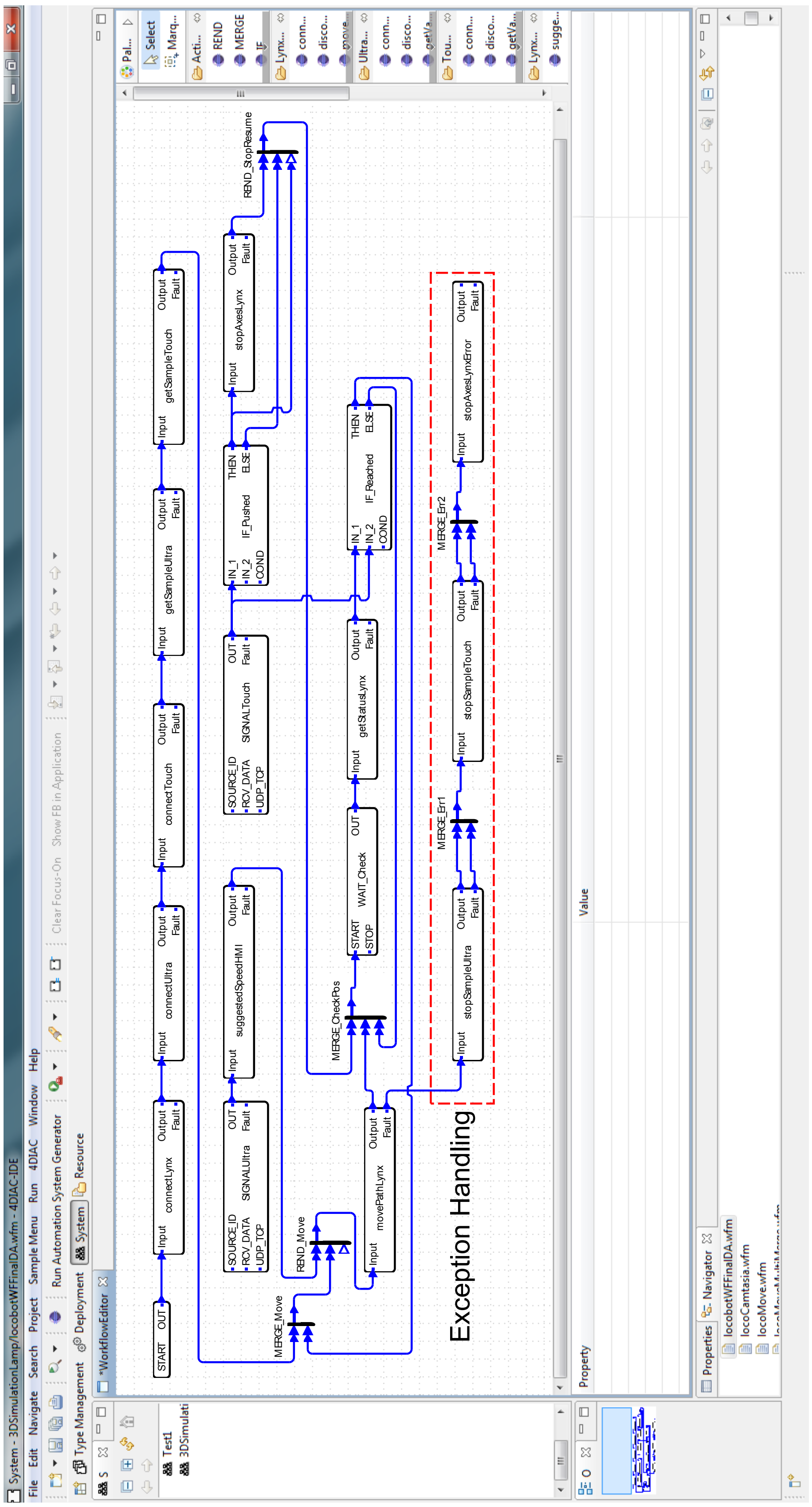


Figure 7.2: Workflow Editor window showing the Workflow Specification for the *Robot-Interaction Example*

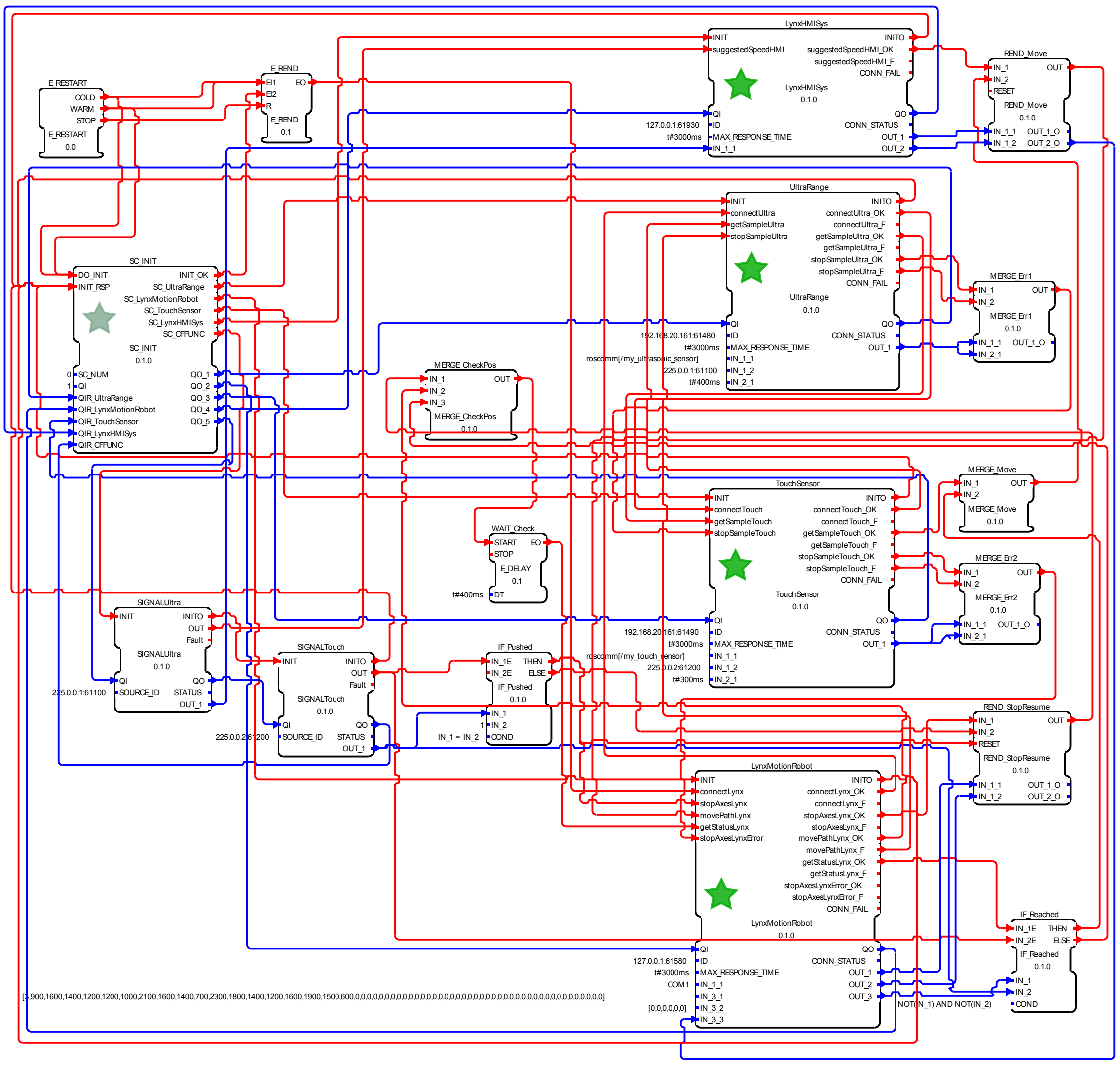


Figure 7.3: Generated control application for the *Robot-Interaction Example*. Service-Component FB-Types are highlighted with green coloured stars. The *SC_INIT* FB is marked with a grey coloured star.

Table 7.1 provides a direct comparison of the engineering effort needed for both programming methods, considering the relevant steps of the engineering process.

The functional components need to be implemented in advance, if the component implementations are not provided by component vendors. Service-Components need to be modelled to enable the model-based programming approach, and if the execution of a directly implemented control application shall be coupled with the 3D-

Engineering step	**Model-based approach**	**Direct implementation**
Implementation of components	required if not available	required if not available
Modelling SCs	required	required for simulation
Implementation of SC interfaces	automatic generation	implementation required
Control-Flow function elements	automatic generation	implementation required or use standard FBs
Programming the control application	activity instances; activity connections; parameter mappings	FB-Instances; event/data connections; constant data inputs

Table 7.1: Comparison of the engineering effort required by the model-based engineering approach and by direct implementation.

Simulation Environment. The direct implementation of interfaces to SCs as well as control-flow function elements, which represent functionalities of Control-Flow Activities, has turned out to be very time consuming. In this context the major engineering effort needs to be put into the implementation of representing FB-Types including their behaviour. The generated SC interfaces and control-flow function elements also include error handling mechanisms which do not need to be implemented separately.

Referring to the last step in Table 7.1, the engineering effort is similar for both methods. Event and data connections need to be established in a FB control application. Similarly, a Workflow Specification needs parameter mapping information between two activities. However, the graphical representation of a Workflow Specification is easier to read, compared to a complex FB-Application.

The complexity of the generated control application can be further reduced, by extending SCs with high-level Service-Functions. These extended service implementations can replace the same functionality that is expressed by aggregations of low-level SFs, by a reduced number of high-level SFs. To enable the execution of the control application combined with the 3D-Simulation Environment *3DCreate* [64], currently modified Service-Component implementations need to be used. The modifications encompass functionalities to send signals to 3DCreate, using the provided Application Programming Interface, which can be detected by the modelled components. A modelled component can react upon the reception of a signal by execution the defined simulation behaviour (e.g., using defined kinematics to move a robotic arm in the simulation).

8 Conclusion and Future Work

The application of modular and flexible robotic systems in production industry to cope with the challenges of fast changing market trends, competition, and rising costs, gains increasing popularity. In order to allow fast configuration and reconfiguration of such complex systems, efficient engineering tools are required. The aim of this thesis was to develop a model-based engineering method that enables simplified programming of a flexible robotic system.

The main idea is to use workflow modelling for the definition of the process that should be performed by the modular robotic system. A graphical Workflow Modelling Language was developed, where so-called activities are used to model invocations of services that are provided by the system components. Activity Connections are used to link activities together, thus forming an execution sequence. An Activity Connection represents a combined control- and data-flow that enables data exchange between activities. The notion of a combined control- and data-flow was chosen in order to keep the Workflow Specification easy to read.

As a main requirement, the Workflow Modelling Language supports reactive workflow execution. This characteristic is necessary to ensure that the executing workflow can react upon exceptional events. Reaction on erroneous situations is crucial for reliable operation of the robotic system. Therefore, the developed Workflow Modelling Language supports exception handling mechanisms. Custom handling strategies to resolve an execution error can be defined within the graphical Workflow Specification.

In the second step an executable, supervisory control application according to IEC 61499 is generated, based on the designed Workflow Specification. A code generator has been developed, which transforms worklow activities to representative Function Blocks according to defined transformation rules. The purpose of the control application is to coordinate the functional components of the robotic system, in order to achieve the desired process behaviour. As a special characteristic, the generated control application realizes reactive workflow execution. This means that the control application does not execute a modelled service invocation itself, but delegates it to the responsible system component. An IEC 61499 compliant runtime environment is used to execute the generated control application.

The graphical workflow editor as well as the code generator were integrated as separate plug-ins into the existing 4DIAC Framework. In order to reduce the initial programming effort, the Eclipse Modelling Framework EMF was used to automatically generate the base class structure from defined EMF data models.

Application examples proved that the model-based engineering method simplifies programming of control applications for modular robotic systems. Especially due to the automatic generation of interfaces to the system components, including status evaluation functionalities, the engineering effort is reduced significantly. Moreover, processes which are modelled as a graphical Workflow Specifications are easier to retrace compared to a large FB-Network containing complex FB-Types. Fast reconfiguration of a control application is performed by modifying the Workflow Specification and re-running the code generator.

The developed model-based engineering approach as well as the prototype implementations of the workflow editor and the code generator provide a useful basis for future development steps, as listed below.

- *Hierarchical workflow modelling*: To enable hierarchical workflow structures, a *sub-workflow* language element needs to be introduced and implemented. Sub-workflows are useful to reduce the complexity of a Workflow Specification by grouping functional connected activities.
- *Extended exception handling concepts*: Currently the Workflow Modelling Language supports embedded exception handling. However, if many exception cases need to be handled, the complexity of the workflow model increases. An autonomous exception handling approach could solve this issue, since exception handling strategies can be defined separated from the Workflow Specification.
- *Improved integration of the 3D-Simulation Environment*: In order to simplify the execution of the control application combined with the 3D-Simulation Environment, an improved integration of the simulation tool into the implemented plug-ins is required. The ability of the control application to communicate with the 3D-Simulation Environment needs to be independent of the implementation of the Service-Components.
- *Control-Flow Activity type editor*: To gain more flexibility in defining functionalities that are not bound to a certain Service-Component, it is planned to implement a Control-Flow Activity editor. In that way the group of predefined Control-Flow Activities can be easily extended.

Summarizing, the results that have been obtained from the first test applications are promising. The realization of the suggested functionalities could take the applicability of the model-based engineering method to a higher level, making it ready to be used for more complex applications.

Bibliography

[1] "Iec 61499-1: Function blocks - part 1: Architecture," International Electrotechnical Commission, Tech. Rep., 2005.

[2] N. Ando, S. Balakirsky, T. Hemker, M. Reggiani, and O. Stryk, *Simulation, Modeling, and Programming for Autonomous Robots: Second International Conference, Simpar 2010, Darmstadt, Germany, November 15-18, 2010, Proceedings*, ser. Lecture Notes in Artificial Intelligence. Springer, 2010. [Online]. Available: http://books.google.at/books?id=8USi-anN1_MC

[3] P. C. Attie, M. P. Singh, A. P. Sheth, and M. Rusinkiewicz, "Specifying and enforcing intertask dependencies," in *19th International Conference on Very Large Data Bases.* Morgan Kaufmann, 1993, pp. 134–145.

[4] S. Bangsow, *Fertigungssimulationen mit Plant Simulation und SimTalk: Anwendung und Programmierung mit Beispielen und Lösungen*, ser. Edition Cad.de. Hanser, 2008.

[5] C. Barreto, V. Bullard, T. Erl, J. Evdemon, and other, "Web services business process execution language version 2.0," OASIS Open, Burlington, USA, Primer, 2007.

[6] D. Brandl, *Design Patterns for Flexible Manufacturing.* ISA, 2006.

[7] H. Cao, H. Jin, S. Wu, and S. Ibrahim, "Petri net based grid workflow verification and optimization," Huazhong University of Science and Technology, Wuhan, China, Technical Report, 2011.

[8] J. Chomicki and D. Toman, *Temporal logic in information systems*, ser. BRICS report series. BRICS, 1997.

[9] J. Christensen, "Iec 61499:a standard for software reuse in embedded, distributed control systems," 2011, last access: 2.11.2011. [Online]. Available: http://knol.google.com/k/iec-61499

[10] M. Dahms, *Modellierung und Steuerung ereignisdiskreter Fertigungssysteme mittels farbiger, zeitbehafteter Petri-Netze.* Aachen, Germany: Shaker Verlag, 2008.

[11] Dassault-Systèmes, “Catia - virtual product,” last access: 7.11.2011. [Online]. Available: http://www.3ds.com/products/catia

[12] M. Dumas and A. ter Hofstede, “Uml activity diagrams as a workflow specification language,” Queensland University of Technology, Brisbane, Australia, Technical Report, 2001.

[13] Eclipse-Foundation, “Eclipse modelling framework,” 2011, last access: 6.11.2011. [Online]. Available: http://www.eclipse.org/modeling/emf/?project=emf#emf

[14] H. Eshuis, “Semantics and verification of uml activity diagrams for workflow modelling,” University of Twente, Enschede, Netherlands, Dissertation, 2002.

[15] R. Eshuis and J. Dehnert, “Reactive petri nets for workflow modeling,” in *Application and Theory of Petri Nets 2003.* Berlin, Germany: Springer-Verlag, 2003, pp. 296–315.

[16] B. Favre-Bulle, *Automatisierung komplexer Industrieprozesse: Systeme, Verfahren und Informationsmanagement.* Springer, 2004.

[17] FORDIAC, “4diac - framework for distributed industrial automation and control,” 2008-2011, last access: 5.11.2011. [Online]. Available: http://www.fordiac.org/

[18] W. Gruber, *Modeling and Transformation of Workflows with Temporal Constraints.* Berlin, Deutschland: Akademische Verlagsgesellschaft Aka GmbH, 2004.

[19] J. Herrington, *Code Generation in Action*, Greenwich CT, USA, 2003.

[20] D. Hollingsworth, “The workflow reference model,” Workflow Management Coalition - WfMC, Winchester, United Kingdom, Specification, 1995.

[21] Holobloc, “Resources for the new generation of automation and control software,” 2011, last access: 2.11.2011. [Online]. Available: http://www.holobloc.com/

[22] IBM and SAP, “Ws-bpel extension for sub-processes (bpel-spe),” White Paper, 2005.

[23] ISIS, "Generic modelling environment," 2008-2011, last access: 6.11.2011. [Online]. Available: http://www.isis.vanderbilt.edu/Projects/gme

[24] D. Ivanova, I. Batchkova, S. Panjaitan, F. Wagner, and G. Frey, "Combining iec 61499 and isa s88 for batch control," in *13th IFAC Symposium on Information Control Problems in Manufacturing*, 2009, pp. 187–192.

[25] S. Jablonski, M. Böhm, and W. Schulze, *Workflow-Management. Entwicklung von Anwendungen und Systemen.* Heidelberg, Deutschland: Dpunkt-Verlag, 1997, vol. 1.

[26] LEGO-Group, "Lego mindstorms products," last access: 20.11.2011. [Online]. Available: http://mindstorms.lego.com/en-us/products/default.aspx

[27] R. Lemaigre, "Gef wiki tutorial," 2011, last access: 6.11.2011. [Online]. Available: http://wiki.eclipse.org/GEF_Description

[28] R. W. Lewis, *Modelling Control Systems using IEC 61499: Applying Function Blocks to Distributed Systems*, ser. IEE Control Engineering Series, No. 59. London, United Kingdom: The Institution of Electrical Engineers, 2001.

[29] S. Ling and H. Schmidt, "Time petri nets for workflow modelling and analysis," School of Computer Science and Software Engineering, Victoria, Australia, Technical Report, 2000.

[30] J. Lunze, *Ereignisdiskrete Systeme - Modellierung und Analyse dynamischer Systeme mit Automaten, Markovketten und Petrinetzen.* München, Germany: Oldenburg Wissenschaftsverlag GmbH, 2006.

[31] Z. Luo, A. Sheth, K. Kochut, and J. Miller, "Exception handling in workflow systems," *Applied Intelligence*, vol. 13, pp. 125–147, August 2000.

[32] Lynxmotion, "Robotic arm al5d," last access: 20.11.2011. [Online]. Available: http://www.lynxmotion.com/c-130-al5d.aspx

[33] MEDEIA-Consortium, "Medeia - state-of-the-art and technology review report," Work Deliverable, 2008. [Online]. Available: http://www.medeia.eu/26.0.html

[34] MEDEIA-Consortium, "Medeia - model-driven embedded systems design environment for the industrial automation sector," White Paper, 2008-2010.

[35] Microsoft, "The component object model," last access: 20.11.2011. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/ms694363%28v=VS.85%29.aspx

[36] Microsoft-Robotics, "Visual programming language introduction," last access: 7.11.2011. [Online]. Available: http://msdn.microsoft.com/en-us/library/bb483088.aspx

[37] B. Moore, D. Dean, A. Gerber, G. Wagenknecht, and P. Vanderheyden, *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework.* International Technical Support Organization, 2004.

[38] K. Nemec, "Simulation und modellierung," last access: 7.11.2011. [Online]. Available: http://www.eurosis-project.eu/3-d-simulation.html

[39] NXTControl, "nxtstudio - engineering software for all tasks," 2011, last access: 5.11.2011. [Online]. Available: http://www.nxtcontrol.com/en/products/nxtstudio.html

[40] OASIS, "Bpel.xml.org - online community for the web services business process execution langugage oasis standard," 1993-2011, last access: 24.10.2011. [Online]. Available: http://bpel.xml.org/

[41] OASIS, "Ws-bpel extension for people (bpel4people) specification version 1.1," Burlington, USA, Specification, 2009.

[42] OASIS, "Web services - human task (ws-humantask) specification version 1.1," Burlington, USA, Specification, 2010.

[43] B. Oestereich, *Die UML-Kurzreferenz 2.0 für die Praxis: Kurz, bündig, ballastfrei.* München, Germany: Oldenbourg, 2009.

[44] OMG, "Model driven architecture (mda) - frequently asked questions," 1997-2010, last access: 6.11.2011. [Online]. Available: http://www.omg.org/mda/faq_mda.htm

[45] ORACLE, "Com4j," last access: 20.11.2011. [Online]. Available: http://java.net/projects/com4j

[46] Oracle, "Oracle bpel process manager overview," last access: 7.11.2011. [Online]. Available: http://www.oracle.com/technetwork/middleware/bpel/overview/index.html

[47] E. Oren and A. Haller, "Formal frameworks for workflow management," DERI - Digital Enterprise Research Institute, Galway, Ireland, Technical Report, 2005.

[48] PLCOpen, "Introduction into iec 61131-3 programming languages," last access: 18.11.2011. [Online]. Available: http://www.plcopen.org/pages/tc1_standards/iec_61131_3/

[49] ROS.org, "Robot operating system - documentation," last access: 10.01.2012. [Online]. Available: http://www.ros.org/

[50] J. Siegel, "Introduction to omg's unified modeling language," 1997-2011, last access: 27.10.2011. [Online]. Available: http://www.omg.org/gettingstarted/what_is_uml.htm

[51] Siemens-PLM, "Explore tecnomatix," last access: 7.11.2011. [Online]. Available: http://www.plm.automation.siemens.com/en_gb/products/tecnomatix/

[52] X. Song, M. Han, and T. Thiery, "Representing and handling workflow exceptions in highly dynamic healthcare environments," in *International Journal Of Intelligent Control and Systems*, vol. 12. Fremont, USA: Westing Publishing Co., 2007, pp. 2–14.

[53] T. Stahl, M. Völter, S. Efftinge, and A. Haase, *Modellgetriebene Softwareentwicklung - Techniken, Engineering, Management*, 2007, vol. 2.

[54] W. M. P. van der Aalst, "Three good reasons for using a petri-net-based workflow management system," *The Kluwer International Series in Engineering and Computer Science: Information and Process Integration in Enterprises: Rethinking Documents, Chapter 10*, vol. 428, pp. 161–182, 1998.

[55] W. van der Aalst, "The application of petri nets to workflow management," Eindhoven University of Technology, Eindhofen, Netherlands, Tech. Rep., 1998.

[56] W. van der Aalst and A. ter Hofstede, "The workflow patterns initiative," 1999-2010, last access: 21.10.2011. [Online]. Available: http://www.workflowpatterns.com/

[57] W. van der Aalst and A. ter Hofstede, "Yawl: Yet another workflow language," Eindhoven University of Technology; Queensland University of Technology, Eindhoven, Netherlands; Brisbane, Australia, Technical Report, 2002.

[58] W. van der Aalst, A. ter Hofstede, N. Russel, and D. Edmond, "Workflow data patterns," Eindhoven University of Technology; Queensland University of Technology, Eindhoven, Netherlands; Brisbane, Australia, Technical Report, 2004.

[59] W. van der Aalst, A. ter Hofstede, N. Russel, and D. Edmond, "Workflow resource patterns," Eindhoven University of Technology; Queensland University of Technology, Eindhoven, Netherlands; Brisbane, Australia, Technical Report, 2004.

[60] W. van der Aalst, A. ter Hofstede, N. Russel, and D. Edmond, "Exception handling patterns in process-aware information systems," Eindhoven University of Technology; Queensland University of Technology, Eindhoven, Netherlands; Brisbane, Australia, Technical Report, 2006.

[61] W. van der Aalst, A. ter Hofstede, N. Russel, and N. Mulyar, "Workflow control-flow patterns: A revised view," Eindhoven University of Technology; Queensland University of Technology, Eindhoven, Netherlands; Brisbane, Australia, Technical Report, 2006.

[62] W. van der Aalst, A. ter Hofstede, N. Russel, and P. Wohed, "On the suitability of uml 2.0 activity diagrams for business process modelling," in *Proceedings of the Third Asia-Pacific Conference on Conceptual Modelling*, vol. 53, Hobart, Australia, 2006, pp. 95–104.

[63] E. Visser and A. van Deursen, "Generative programming," 2000-2010, last access: 6.11.2011. [Online]. Available: http://www.program-transformation.org

[64] VisualComponents, "3dcreate," last access: 7.11.2011. [Online]. Available: http://www.visualcomponents.com/Products/3DCreate

[65] WfMC, "The workflow management coalition," last access: 7.11.2011. [Online]. Available: http://www.wfmc.org/

[66] WfMC, "Terminology and glossary," Workflow Management Coalition - WfMC, Winchester, United Kingdom, Specification, 1999.

[67] Wikipedia, "Business process execution language," 2011, last access: 25.10.2011. [Online]. Available: http://en.wikipedia.org/wiki/Business_Process_Execution_Language

[68] Y. Wu, "An introduction to bpel standard and its extensions," Helsinki University of Technology, Helsinki, Finland, Technical Report, 2007.

[69] YAWL-Group, "Yet another workflow language (yawl)," last access: 7.11.2011. [Online]. Available: http://www.yawlfoundation.org/

[70] A. Zoitl and H. Prähofer, "Building hierarchical automation solutions in the iec 61499 modeling language," in *9th IEEE International Conference on Industrial Informatics (INDIN)*, 2011, pp. 557–564.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.