Process Model for the Development of Semi-Autonomous Service Robots

From the Faculty of Physics and Electrical Engineering
University of Bremen

for obtaining the academical degree of

*Doktor-Ingenieur (Dr.-Ing.)*

approved dissertation

by

Dipl.-Ing. Oliver Prenzel

Primary Supervisor: Prof. Dr.-Ing. Axel Gräser
Secondary Supervisor: Prof. Dr.-Ing. Udo Frese
Submitted at: 26 May 2009
Date of Colloquium: 06 July 2009
‘The character of software development sometimes seems closer to mathematics and art than it does to most other engineering disciplines. Software is inherently an intangible, intellectual development medium. No laws of physics govern its behavior; it is both marvelously and dangerously malleable. For this reason, it is critical that mature disciplines and processes be applied when working with software.’ [Ahern et al., 2003]
I like to thank my supervisor Prof. Dr. Axel Gräser for giving me the opportunity to create this work within the fascinating area of service robotics. I really appreciate his support during my time at the Institute of Automation and I know that I learned a lot during this time. Special acknowledgments are also dedicated to my secondary supervisor, Prof. Dr. Udo Frese. His review of this thesis as well as the comments and discussions have been of outstanding quality and profound detail. Furthermore, I want to thank Prof. Dr. Georg Thiele and Prof. Dr. Walter Anheier for their interest in my work, for taking part in the examination board and for several fruitful discussions.

Somebody who definitely shaped me and my work is my friend Dr. Christian Martens. I want to take the opportunity to thank him for many important things: I had the chance to contribute as a diploma student during the time of his foundation of essential development concepts for semi-autonomous service robots at the IAT. After his time at IAT it was my pleasure to build on a sound basis to continue his work. In parallel, even though always being extremely busy, Christian never lost the contact to his original work - we had joint publications and many inspiring scientific discussions. Finally, he taught me how to track and reach milestones, especially this most important one here. I also thank Christian for the strict review of large parts of this thesis.

My congratulations go to my colleague and friend, Dr. Darko Ojdanić, for winning the race for reaching the Ph.D. degree first. I am indebted to him due to several years of fruitful joint work while we never lost the joyful sight on work and life.

I like to express my wishes for successful continuation of our projects to my younger colleagues and friends, Uwe Lange, Sorin Grigorescu and Christos Fragkopoulos. There should never be a hill too steep to climb. Of course there are further colleagues at IAT who I have to blame for a nice time, in particular I want to thank the good souls of IAT, Erwin Wendland and Lothar Renner.

I supervised many students during the last years and I want to express my gratitude for all their effort. However, two names cannot be pointed out adequately enough: Henning Kampe and Alice Boit. These tough workers joined me to dive deep into the conceptual space of my work and without their creative and excellent contributions parts of this work would not have been realized in practice. An additional thanks goes to Alice for her thorough proofreading.

Lots of thanks is devoted to my parents, Helmut and Gisela, for their support and their believe in me, which is certainly an important basis for success.

Finally, it is due to my girlfriend Janine that the finalization phase of my thesis did not mean to be entirely delved into work, but to keep the eyes open for all the wonderful moments that really matter in life besides professional success. I thank her for this and the love she is giving to me.

Bremen, July 2009

Oliver Prenzel
Abstract

The creation of a universal service robot for daily use in the home or working environment is an ambitious objective of ongoing research and engineering efforts. Considering today’s state-of-the-art, service robots cannot be expected to be able to act fully autonomously and intelligently in unstructured environments within the middle-term future. This is due to the generally high complexity of meaningful task executions, with still unsolved problems in task and motion planning as well as sufficient scene understanding. To encounter these problems, an approach has been developed that reduces the technical complexity with the help of two development paradigms: The task knowledge for a certain scenario is pre-structured, and the robotic system executes its tasks semi-autonomously, i.e. involves the user into task execution. By means of simulations in representative sample scenarios it has been demonstrated that the real time suitability of task planning based on this approach can be guaranteed.

Within this work, the fundamental concepts are now integrated into a complete process model for the development of semi-autonomous service robots. The process model gives a clear guidance for a scenario- and behavior-driven development of a service robotic system. The process contains four levels and starts with analysis steps that lead to well-defined outputs for guiding subsequent steps. Due to the paradigm of pre-structuring and offline verification of task knowledge, a lot of effort is required for the introduction of new task knowledge and the enhancement of existing one. The complexity of the specification procedure is encountered with the development of special tools that provide a user-friendly input of task knowledge and fully integrate into the overall concept. The process of task knowledge specification leads to the definition of software interfaces that encapsulate functionality of a robotic system. Besides defining the interfaces, the process model guides the development process on the functional level and also supports the final testing on component and scenario level. As it will be argued, the application of a process model cannot be successful without support of suitable tools. Besides the tools that especially target at task knowledge specification methods developed throughout this work, the complete process model makes use of a state-of-the-art CASE tool for the development of software systems based on executable UML-models.

The applicability of the proposed method is validated with the help of different scenarios from the AMaRob project. The AMaRob research project, currently running at the Institute of Automation (IAT), at the University of Bremen, targets at the development of rehabilitation robots for severely impaired persons and the robot’s evaluation in domestic and professional environments. Three representative scenarios that are realized with the help of the process model cover the preparation of a meal, the functionality control of workpieces in a rehabilitation workshop as well as various service tasks at a library service counter.
Kurzfassung


The "FRIEND::Process"

In the following an overview is given of the process model FRIEND::Process, which has been developed in this work. It serves as a graphical contents description and is referenced throughout the work. The diagram contains process steps (S), process repositories (R) and development artifacts (A). A compact summary of the FRIEND::Process can be found in Appendix A, page 205. An online version of the summary is included in the IAT Wiki: [Prenzel, 2009].
Contents

1. Introduction: Robots in Service - A Vision to Become True .......................... 1
   1.1. Service Robots in the Past and Today ...................................................... 1
   1.2. Service Robots in Near Term Future ....................................................... 2
   1.3. A Process Model to Guide the Developments of Semi-Autonomous Service Robots 4
   1.4. Outline of this Work .................................................................................. 6

2. Background and Problem Statement .................................................................. 7
   2.1. The Evolution of Rehabilitation Robotics ..................................................... 7
   2.2. The Semi-Autonomous Rehabilitation Robot FRIEND .................................... 9
   2.3. Rehabilitation Robotic Support Scenarios .................................................... 11
       2.3.1. AMaRob Scenario #1: Preparation of a Meal and Support for Eating .......... 12
       2.3.2. AMaRob Scenario #2: Working at a Library Service Desk ....................... 14
       2.3.3. AMaRob Scenario #3: Quality Check of Workpieces ................................. 14
   2.4. Task Specification and Execution With Process-Structures ............................ 16
       2.4.1. Abstract Process-Structures ................................................................. 16
       2.4.2. Elementary Process-Structures ............................................................. 18
       2.4.3. Summary on Process-Structures ............................................................ 20
   2.5. Integrated Process with Model and Tool Support Required .......................... 21

3. State-Of-The-Art ................................................................................................. 23
   3.1. Process Models ............................................................................................ 23
       3.1.1. Model Driven Engineering / Development ................................................. 24
       3.1.2. Key Issues to be Tackled via MDE ........................................................ 24
       3.1.3. The HARMONY Development Process .................................................. 30
       3.1.4. From Generic to Problem Specific Process Models ................................. 35
   3.2. Methods for Programming Service Robot Behavior ........................................ 36
       3.2.1. Task Description Language .................................................................... 37
       3.2.2. Graphical programming languages (DeVar: RoboGlyph) ......................... 38
       3.2.3. The MORPHA Style Guide for Icon-Based Programming ....................... 39
       3.2.4. Microsoft Robotic Studio ....................................................................... 39
       3.2.5. Industrial Automation Systems: Programming with IEC 61131-3/61499 ........ 41
       3.2.6. Design of IEC 61131-3 Function Blocks using SysML ............................... 44
       3.2.7. CORFU - Function Block Development Kit (FBDK) ................................. 44
       3.2.8. Universal Realtime Behavior Interface - URBI ......................................... 45
       3.2.9. Summary ............................................................................................... 46

4. Requirements Analysis for the Process Model "FRIEND::Process" ......................... 47
   4.1. The Driving Forces: Scenario, Model and Tool Driven Process ....................... 48
   4.2. Requirements for Scenario Analysis ............................................................ 50
## Contents

8.1.2. Development of Skill Tester ........................................... 155
8.1.3. Configuration of World Model Data .................................. 156
8.2. Step [4b]: Scenario Testing ............................................ 159
  8.2.1. Tools and Repositories .............................................. 159
  8.2.2. Sequencer - Architectural Large Scale Design ............... 161
  8.2.3. Sequencer - Architectural Design of Task Planner ........... 161
  8.2.4. Sequencer - Architectural Design of Skill Executer ......... 162
  8.2.5. Sequencer - Behavioral Design .................................... 165
8.3. Summary ................................................................. 165

9. Application and Evaluation of the FRIEND::Process in AMaRob Scenarios 169
  9.1. Step 1a - Scenario Specification .................................... 169
  9.2. Step 1b - Sub-Scenario Specification ................................. 171
      9.3.1. System Specification ........................................... 171
      9.3.2. Further Task Participating Objects .......................... 173
      9.3.3. Sub-Scenario #1: Fetch Meal from Fridge .................. 174
      9.3.4. Sub-Scenario #2: Cook Meal .................................. 176
      9.3.5. Sub-Scenario #3: Fetch Meal from Oven ..................... 176
      9.3.6. Sub-Scenario #4: Take Away the Lid ......................... 178
      9.3.7. Sub-Scenario #5: Eating support ............................. 178
      9.3.8. Sub-Scenario #6: Take Away the Mealtray .................. 183
  9.5. Step 3a/b - Skill Development ...................................... 187
  9.6. Step 4a - Skill Testing ............................................. 189
  9.7. Step 4b - Scenario Testing .......................................... 193
  9.8. Evaluation of Application Results .................................. 197

10. FRIEND::Process – Retrospective and Perspective 199

A. Compact Overview on the Process Model "FRIEND::Process" 205
   A.1. Process Steps ..................................................... 205
   A.2. Repositories ...................................................... 209
   A.3. Development Artifacts ............................................ 209
   A.4. Development Tools ................................................. 210

B. Composed Operators (COPs) in AMaRob Scenario 'Fetch Meal' 211
   B.1. DepartFromContainer(Manipulator, Object, Container) ....... 212
   B.2. GetObjectOutside(Manipulator, Object, Container, InsertLoc) 212
   B.3. MoveObjectToRelLoc(Manipulator, Object, Container, InsertLoc) 213
   B.4. MoveObjectFromRelLoc(Manipulator, Object, Container, InsertLoc) 213
   B.5. PlaceObjectInside(Manipulator, Object, Container, InsertLoc) 214

C. Sample Protocol of Task Execution .................................. 215

D. Situation Graph .......................................................... 221
   D.1. Extraction of Valid Situations .................................... 221
   D.2. Determination of Unique Sets of Facts ......................... 222

E. Manually Drawn Petri Net ............................................... 225
Contents

Bibliography 227
List of Figures 237
List of Tables 241
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbC</td>
<td>Automation by Configuration: Instead of programming the robotic system, the development of new system functionality is done via configuration. The configuration interface embeds all necessary specification, programming and verification methods and guides the programmer through the configuration process. Only the really relevant configuration steps according to a certain programming context are shown to the user. The AbC-principle shall be understood as system-controlled and constrained-based programming by configuration.</td>
</tr>
<tr>
<td>ADL</td>
<td>Activities of Daily Living, [Wikipedia, 2009a], p. 8</td>
</tr>
<tr>
<td>Aggregation</td>
<td>In UML an aggregation is a special form of association between classes, where the component’s lifetime is independent of the aggregating unit. In object model diagrams it is visualized with an empty diamond at the aggregating unit’s side. See also [Borland Software Corporation, Inc., 2003]., p. 89</td>
</tr>
<tr>
<td>AMaRob</td>
<td>Autonomous Manipulator Control for Rehabilitation Robots, <a href="http://www.amarob.de">http://www.amarob.de</a>, p. 9</td>
</tr>
<tr>
<td>AND/OR net</td>
<td>An AND/OR net provides a graph-based compact, distributed, domain-specific representation of geometric configurations of parts and devices in a generalized robotic work cell. The approach maintains a correspondence from geometric state information to task and motion plans and on-line discrete-event control that is not available in traditional action-based planners. [Cao and Sanderson, 1998], p. 16</td>
</tr>
<tr>
<td>AOP</td>
<td>Assembly operation, p. 17</td>
</tr>
<tr>
<td>API</td>
<td>Application programming interface, p. 105</td>
</tr>
<tr>
<td>Association</td>
<td>In UML an association is the general form of relationship between classes. In object model diagrams it is visualized with a simple link between the connected units. See also [Borland Software Corporation, Inc., 2003]., p. 89</td>
</tr>
<tr>
<td>BMBF</td>
<td>German Federal Ministry of Education and Research, Bundesministerium für Bildung und Forschung, p. 9</td>
</tr>
<tr>
<td>Bumblebee</td>
<td>Bumblebee® is a stereo-camera system with built-in calibration, synchronization as well as stereo-projective calculation features. The manufacturer is Point Grey, <a href="http://www.ptgrey.com/index.asp">http://www.ptgrey.com/index.asp</a>, p. 10</td>
</tr>
</tbody>
</table>
CASE  Computer-aided Software Engineering. In the field software engineering, CASE is the scientific application of a set of tools and methods to a software which is meant to result in high-quality, defect-free, and maintainable software products. [Wikipedia, 2009c], p. 22

CMM(I)  Capability Maturity Model (Integration) [Humphrey, 1989], p. 28

Composition  In UML a composition is a special form of association between classes, where the component’s lifetime depends on the composing unit. In object model diagrams it is visualized with a filled diamond at the composing unit’s side. See also [Borland Software Corporation, Inc., 2003], p. 89

COP  Composed operator, p. 18

CORFU  Common Object-oriented Real-time Framework for Unified development of distributed applications, p. 44

DeVar  Desktop Vocational Assistant Robot, p. 38

DOP  Disassembly operation, p. 17

EEOP  Elementary executable operation. These are the building blocks on PSE-level. During task execution, each EEOP is mapped to a skill method call in the reactive layer., p. 19

ERM  An entity-relationship model (ERM) in software engineering is an abstract and conceptual representation of data. Entity-relationship modeling is a relational schema database modeling method, used to produce a type of conceptual schema or semantic data model of a system, often a relational database, and its requirements in a top-down fashion. [Wikipedia, 2009d], p. 87

ESS  Engineering Support System [Tranoris and Thramboulidis, 2003], p. 44

FB, Function Block  A function block (FB) within the FRIEND::Process encapsulates one EEOP, which in turn is equivalent to a skill method call within the reactive layer., p. 57

FBD  Function Block Diagram, one of five languages defined in IEC 61131-3 for industrial automation systems., p. 42

FBDK  Function Block Development Kit, [Holobloc, Inc., 2008], p. 44

FBN  Function block network, p. 55


FRIEND::Architecture  Synonym for the MASSiVE control architecture for semi-autonomous service robots, p. 59

FRIEND::Process  Process Model for the Development of Semi-Autonomous Service Robots, developed within the FRIEND projects. A compact summary of the FRIEND::Process can be found in Appendix A, page 205. An online version of the summary is included in the IAT Wiki: [Prenzel, 2009], p. 48
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUI</td>
<td>Graphical user interface, p. 78</td>
</tr>
<tr>
<td>HMI</td>
<td>Human Machine Interface, p. 8</td>
</tr>
<tr>
<td>IAT</td>
<td>Institute of Automation, University of Bremen, <a href="http://www.iat.uni-bremen.de">http://www.iat.uni-bremen.de</a>, p. 9</td>
</tr>
<tr>
<td>IEC</td>
<td>International Electrotechnical Commission, p. 41</td>
</tr>
<tr>
<td>IFR</td>
<td>International Federation of Robotics, [IFR, 2007], p. 1</td>
</tr>
<tr>
<td>IL</td>
<td>Instruction List, one of five languages defined in IEC 61131-3 for industrial automation systems., p. 42</td>
</tr>
<tr>
<td>IE</td>
<td>Intelligent Environment, IE An Intelligent Environment (IE) as meant here supports a robotic system with distributed smart components. It thus lowers the technical complexity of the robotic system itself. Consequently, the IE differs from the concepts ambient intelligence or ubiquitous computing, where only human being centered environments are designed. See also [Prenzel et al., 2006]., p. 14</td>
</tr>
<tr>
<td>IPMCS</td>
<td>Industrial Process Measurement and Control Systems [Tranoris and Thramboulidis, 2003], p. 44</td>
</tr>
<tr>
<td>LD</td>
<td>Ladder Diagram, one of five languages defined in IEC 61131-3 for industrial automation systems., p. 43</td>
</tr>
<tr>
<td>MASSiVE</td>
<td>Multi-layer Architecture for Semi-Autonomous Service Robots with Verified Task Execution, also called FRIEND::Architecture, [Martens et al., 2007], p. 59</td>
</tr>
<tr>
<td>MDA</td>
<td>Model Driven Architecture, [OMG, 2008], p. 24</td>
</tr>
<tr>
<td>MDD</td>
<td>Model Driven Development, p. 29</td>
</tr>
<tr>
<td>MDE</td>
<td>Model-Driven Engineering, [Schmidt, 2006], p. 24</td>
</tr>
<tr>
<td>MML</td>
<td>Modeling Maturity Level [Kleppe et al., 2003], p. 28</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>In UML the multiplicity of associations indicates the number of involved objects. (*)-multiplicity means that theoretically the number is infinite. See also [Borland Software Corporation, Inc., 2003]., p. 90</td>
</tr>
<tr>
<td>MVR</td>
<td>Mapped Virtual Reality, p. 105</td>
</tr>
<tr>
<td>Nemo</td>
<td>Nemo® is one of the most recent developments of an electrical wheelchair from Meyra [Meyra, 2008]. This platform is equipped with various adjustabilities of the seat for moving upwards or providing a stand up position for the user., p. 9</td>
</tr>
<tr>
<td>OC</td>
<td>Object constellation, p. 17</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group, [OMG, 2008], p. 24</td>
</tr>
<tr>
<td>Ontology</td>
<td>An ontology is a formal, explicit specification of a shared conceptualization [Gruber, 1993]. An ontology provides a shared vocabulary, which can be used to model a domain, that is, the type of objects and/or concepts that exist, and their properties and relations. [Arvidsson and Flycht-Eriksson, 2008], p. 69</td>
</tr>
<tr>
<td>Glossary</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>OpenGL</td>
<td>OpenGL (Open Graphics Library, [OpenGL.org, 2009]) is a standard specification that defines a cross-platform API to realize applications with 2D or 3D computer graphics., p. 105</td>
</tr>
<tr>
<td>Petri net</td>
<td>A Petri net (also known as a place/transition net or P/T net) is one of several mathematical modeling languages for the description of discrete distributed systems. A Petri net is a directed bipartite graph, in which the nodes represent transitions (i.e. discrete events that may occur), places (i.e. conditions), and directed arcs (that describe which places are pre- and/or postconditions for which transitions). Petri nets were invented in August 1939 by Carl Adam Petri - at the age of 13 - for the purpose of describing chemical processes [Wikipedia, 2009h], p. 19</td>
</tr>
<tr>
<td>PLC</td>
<td>Programmable Logic Controller, p. 41</td>
</tr>
<tr>
<td>POU</td>
<td>Program Organisation Unit, p. 41</td>
</tr>
<tr>
<td>Process-structures</td>
<td>Process-structures pre-structure a certain task by context-related limitation of the task knowledge to a minimal subset; enabling the verification of the logical correctness and finally leading to a robust runtime system behavior. Process-structures are defined on two levels of abstraction: $PS_A =$ abstract process-structures and $PS_E =$ elementary process-structures. [Martens, 2003, Martens et al., 2007], p. 5</td>
</tr>
<tr>
<td>$PS_A$</td>
<td>Abstract process-structure, p. 16</td>
</tr>
<tr>
<td>$PS_E$</td>
<td>Elementary process-structure, p. 18</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service. QoS-requirements define how well the system shall perform, i.e. they specify quantifiable performance measures., p. 31</td>
</tr>
<tr>
<td>RFID</td>
<td>Radio Frequency Identification, p. 50</td>
</tr>
<tr>
<td>Service robot</td>
<td>A robot which operates semi or fully autonomously to perform services useful to the well-being of humans and equipment, excluding manufacturing operations, p. 1</td>
</tr>
<tr>
<td>SFC</td>
<td>Sequential Function Chart, one of five languages defined in IEC 61131-3 for industrial automation systems., p. 43</td>
</tr>
<tr>
<td>Skill</td>
<td>The terms EEOP and skill represent the same, namely the algorithmic implementation of a basic system functionality, e.g. a manipulative action or a machine vision operation. The different terms have historical origins at the IAT. The term EEOP is used within the task planning domain and the term skill is used within the reactive system layer. The historical origin for EEOPs is [Martens, 2003] and for skills it is mainly the software framework Smartsoft [Schlegel and Wörz, 1999]. Within the domain of function block networks, one function block represents one EEOP/skill., p. 18</td>
</tr>
<tr>
<td>SO</td>
<td>Abbreviation for Scene Object, p. 104</td>
</tr>
<tr>
<td>SSI</td>
<td>(Symbolically represented) Sub-symbolic item, p. 99</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ST</td>
<td>Structured Text, one of five languages defined in IEC 61131-3 for industrial automation systems.</td>
</tr>
<tr>
<td>Stereotype (UML)</td>
<td>The purpose of UML stereotypes is to provide an extension mechanism to the basic set of UML. They allow designers to extend the vocabulary of UML in order to create new model elements, derived from existing ones, but that have specific properties that are suitable for a particular problem domain or otherwise specialized usage.</td>
</tr>
<tr>
<td>Sub-symbolic</td>
<td>The terms originates from the theory of neuronal networks and describes the characteristic of distributed information within the network. In general, sub-symbolic data is understood as concrete physical information about a real world object, e.g. color, size, shape, location or weight.</td>
</tr>
<tr>
<td>Symbolic</td>
<td>Abstract symbolic (discrete) representation of real world objects, e.g. &quot;C&quot; for a cup. Symbolic task planners operate on the basis of abstract symbols. See also: Sub-symbolic.</td>
</tr>
<tr>
<td>SysML</td>
<td>Systems Modeling Language, increasingly established standard for modeling of complex systems.</td>
</tr>
<tr>
<td>TCA</td>
<td>Task Control Architecture [Simmons, 1994].</td>
</tr>
<tr>
<td>TCPR</td>
<td>Type conformal parameter replacement. Introduced in [Martens, 2003] and summarized in Definition 9 (page 80), p. 79</td>
</tr>
<tr>
<td>TDL</td>
<td>Task Description Language, [Simmons and Apfelbaum, 1998], p. 37</td>
</tr>
<tr>
<td>TPO</td>
<td>Task participating object, p. 17</td>
</tr>
<tr>
<td>TRE</td>
<td>Task representing elements, p. 74</td>
</tr>
<tr>
<td>UC</td>
<td>Use case, p. 208</td>
</tr>
<tr>
<td>UI</td>
<td>User interface. With the Qt-Designer Qt user interfaces can be set up conveniently. They are stored in an XML-based user interface (ui) file and are converted by the user interface compiler (UIC) into a C++ class that can be included into the architectural design of a graphical user interface (GUI) application., p. 90</td>
</tr>
<tr>
<td>UIC</td>
<td>User interface compiler, provided by Qt [Nokia-Corporation, 2009], to convert user interface files, as they result from the Qt designer, into C++ classes that can be included into an application., p. 90</td>
</tr>
<tr>
<td>Viewport</td>
<td>A viewport is a rectangular region in computer graphics. It has several definitions in different contexts, e.g. in 3D computer graphics it refers to the 2D rectangle used to project the 3D scene to the position of a camera., p. 105</td>
</tr>
<tr>
<td><strong>VPL</strong></td>
<td>Visual Programming Language. Survey on VPLs and definition is given in: [Boshernitsan and Downes, 1997], p. 36</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td><strong>Widget</strong></td>
<td>A widget (or control) is an element of a graphical user interface (GUI) that displays an information arrangement changeable by the user, such as a window or a text box [Wikipedia, 2009f], p. 87</td>
</tr>
<tr>
<td><strong>XML</strong></td>
<td>The Extensible Markup Language (XML) is a general-purpose specification for creating custom markup languages. [Wikipedia, 2009n], p. 90</td>
</tr>
</tbody>
</table>
• Within this work, the following special and numbered elements are used:

  – **Statement**: General statement

  – **Definition**: General definition

  – **Process Requirement - RP**: Requirement with respect to the process model FRIEND::Process itself. They are abbreviated with *RP*.

  – **Tool Requirement - RT**: Requirement with respect to the tools that have been specially developed for the process model. They are abbreviated with *RT*.

• **Colors in UML diagrams**:

```
<table>
<thead>
<tr>
<th>Color Legend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Focused Element</td>
</tr>
<tr>
<td>Helper Element in Current Context</td>
</tr>
<tr>
<td>Input Artifact of Process Step or Data Structure in General</td>
</tr>
<tr>
<td>Core Element from Other Context</td>
</tr>
<tr>
<td>Statechart</td>
</tr>
</tbody>
</table>
```
1

Introduction: Robots in Service - A Vision to Become True

1.1. Service Robots in the Past and Today

The International Federation of Robotics (IFR, [IFR, 2007]), defines a Service Robot as follows:

'A robot which operates semi or fully autonomously to perform services useful to the well being of humans and equipment, excluding manufacturing operations.'

Mankind’s fascination for human- or animal-like machines doing some service for its owner may reach back thousands of years, probably to circa 800 B.C., where, as described in Homer’s Iliad, artificial powerful and intelligent creatures supported Hephaistos, the Greek god of smiths, fire and metalworking [Graefe and Bischoff, 2003]. On the way from that ancient vision to the roots of modern industrial robotics, skilled craftsmen in Europe and Japan have succeeded to create automated dolls in the 16th respectively the 18th century. These dolls were for example able to bring a cup of tea to a guest over a distance of 2 m [Graefe and Bischoff, 2003]. But only in the 1960s, the technological progress was far enough for the hour of birth of modern industrial robots. In 1961 the first robotic system has been installed by General Motors in Ewing Township, USA, for the unloading of an aluminum die casting machine [Schraft et al., 2004]. From that time on, industrial robotics has undergone considerable developments till reaching the status quo, where industrial productions without robotic support cannot be imagined anymore.

Gradually, improvements in sensor systems and control technique evolved and in the 1980ies, the next decisive evolutionary step with respect to service robotics was reached with the realization of first robotic installations outside of production fields. These were applications like an autonomous road vehicle that could follow a road at a speed of 96 km/h and which was designated as a world record at that
1. Introduction: Robots in Service - A Vision to Become True

Figure 1.1. Statistics on service robots for professional use [IFR, 2006].

1.1. Introduction: A Vision to Become True

Time [Graefe and Bischoff, 2003]. Also, many visions arose along with various prototypic installations. Joseph Engelberger, also referred to as the "Father of (Industrial) Robotics" [IEEE, 2007] stated in his book "Robotics in Service" [Engelberger, 1989] in 1989 that the time is ripe for robotics to launch itself into an entirely new marketplace - the field of service robotics. He described the diverse technological fields that, according to his conviction, had already shaped the basis for the visions in the 1980ies. In significant contrast to the 1985 Delphi Study (see [Engelberger, 1989]), Engelberger projected robotics in service to be the largest class of applications in 1995. He justified his claims with the fact that in service activities much greater opportunities exist than in manufacturing. Further on, Engelberger qualitatively detailed his visions of various application fields like surgery, commercial cleaning, farming, military service, assistants for the elderly, household service and others. However, as to be discussed below, the realization of marketable applications in several of the application fields named by Engelberger took approximately 10 years longer and some of the visions still wait to become true.

1.2. Service Robots in Near Term Future

Looking at the nowadays state-of-the-art, different kinds of systems that can be characterized as service robots according to the IFR-definition are about to be or already have been established on the market. A good overview on the systems that are already in daily service as well as the future impact of service robots is reflected by statistical data that is regularly published by the IFR and measures the number of installations of robotic systems [IFR, 2006]. Recent data and the forecasts until 2010 are available in Figure 1.1 and Figure 1.2.

According to Figure 1.1, the IFR counted within the field of professional use the amount of 40.000 service robotic units that have been installed worldwide up to 2006. Their main application areas are defense, rescue and security with majority of unmanned aerial and ground-based vehicles for military. Further on, milking robots, underwater systems, pool cleaners, demolition robots for the construction industry, robot assisted surgery and mobile platforms for general use follow as most established application fields for service robots. As projections into the near term future, i.e. within 2007-2010, the IFR estimates another 35.000 units of service robots to be installed for professional use.
1.2. Service Robots in Near Term Future

In the second field of domestic or personal use, as Figure 1.2 shows, up to 3.5 million units were sold until 2006. Here, the main areas are domestic applications, such as household robots for vacuum cleaning, lawn-mowing, entertainment or leisure. The typical application fields within the second category and the rather low price per unit as well as the completely different target group explains the high number of sold units and also justifies the distinction between the two categories. The projections of domestic robotic units to be sold, given by the IFR for 2007-2010, is 1.34 million and an extra number of 2.2 million units as entertainment and leisure robots, which are, mostly, very low cost units.

The presented statistics reveal that, due to the technological progress, robotic systems are currently becoming mature to enter more and more application areas outside of the industrial environments.

However, as commented in [Graefe and Bischoff, 2003]

- Each of them can, to a very limited extend, perceive its environment and displays only traces of intelligence.
- Each of them is a specialist, able to deliver only one kind of service in only one kind of environment.

Thus, looking back at the above cited visions from a quarter-century ago - i.e. referring for example to Engelberger’s visions about personal robots and 24h-assistants - it has to be admitted that the realization of those systems is yet a matter of research. According to the current state-of-the-art, the consequent realization of a fully autonomous assistive service robot still has to be stated as unrealistic at the moment. The reasons are manifold, but mainly originate from the different conditions between industrial robotic applications and the requirements in unstructured home or equivalent environments. In the industrial fields, there are well defined working cells with for example constant and known illumination. These conditions do not hold for service robots which operate in domestic fields. Here, this a priori knowledge is not available in most cases. A system, which is designed for fully autonomous operation under these conditions, if realized at all, will be cost-intensive and will perform tasks with poor efficiency and reliability due to its inherent high technical complexity [Dario et al., 2004]. This contrasts with the desire for a flexible and manageable system with predictable behavior.

The following statement summarizes the contemplations so far:

**Statement 1** *Today’s service robots either are specialists themselves or due to system complexity, specialists are required for programming of service robots.*
1.3. A Process Model to Guide the Developments of Semi-Autonomous Service Robots

Despite the rather pessimistic projection for a fast realization of fully autonomous systems that are able to operate in arbitrary unstructured environments, there are promising research approaches which take various design constraints into account to achieve service robot systems of real practical use much faster. One of these design paradigms is the application of the concept of shared-mode control [Didi et al., 1999, Dario et al., 2001, Martens et al., 2002, Colle et al., 2002, Prenzel et al., 2007].

Within Amazon’s Mechanical Turk project [Amazon.com, 2008] a good explanation of the basic idea is given:

> When we think of interfaces between human beings and computers, we usually assume that the human being is the one requesting that a task be completed, and the computer is completing the task and providing the results. What if these processes were reversed and a computer program could ask a human being to perform a task and return the results?

With respect to a service robot this means that the user’s cognitive capabilities are taken into account when an alternative autonomous solution is not available, is too complex or is not robust enough. Also, in the case of uncertainties or conflicts according to the current sensor signals, a user interaction may be helpful. Target platforms that can benefit from this principle are all systems that are, at least temporarily, operated by a human being. One kind of service robot that is especially suitable for the semi-autonomous approach is the category of rehabilitation robotic systems for severely impaired people. The shared-mode control principle is summarized in the first essential development paradigm as given the following:

**Definition 1**

**Semi-Autonomous Task Execution (Essential Paradigm 1):** The shared-mode control as introduced in Section 2.1 builds the basis of the control concept for task planning and execution. A system-controlled integration of the user’s cognitive capabilities into autonomous task execution takes place.

The second essential paradigm is derived from the requirements for an appropriate software architecture that incorporates a task planning mechanism which is suitable to cope with planning problems as they arise in typical everyday environments. The development of such architectures is still an area of active research [Nesnas et al., 2006]. Hybrid multi-layer architectures like TCA [Coste-Maniere and Simmons, 2000] or 3T [Bonasso et al., 1998] predominate in the field of fully autonomous systems. They are usually based upon a deliberative task planner.

The problem with classical deliberative planning approaches can be argued as follows: Basically, a deliberative planning mechanism includes a graph search problem [Russel and Norvig, 2003]. The world is modeled in the system with the help of symbolic facts, where each node of a graph represents a state (snapshot) of the world. The planner has to find a sequence of operations that transforms a given initial state into a desired target state. The complexity of breadth-first graph search algorithms is illustrated in Table 1.1. The search depth \( d \) represents the length of the operation sequence. It
1.3. A Process Model to Guide the Developments of Semi-Autonomous Service Robots

<table>
<thead>
<tr>
<th>Depth (d)</th>
<th>Nodes (N)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>100</td>
<td>10 milliseconds</td>
</tr>
<tr>
<td>4</td>
<td>11.100</td>
<td>1.11 seconds</td>
</tr>
<tr>
<td>6</td>
<td>$10^6$</td>
<td>111 seconds</td>
</tr>
<tr>
<td>8</td>
<td>$10^8$</td>
<td>3 hours</td>
</tr>
<tr>
<td>10</td>
<td>$10^{10}$</td>
<td>12 days</td>
</tr>
<tr>
<td>12</td>
<td>$10^{12}$</td>
<td>3.5 years</td>
</tr>
<tr>
<td>14</td>
<td>$10^{14}$</td>
<td>352 years</td>
</tr>
<tr>
<td>16</td>
<td>$10^{16}$</td>
<td>35.233 years</td>
</tr>
</tbody>
</table>

Table 1.1. Number of expanded nodes and calculation time in breadth-first graph search. Branching factor $b = 10$ and a calculation time of 10,000 nodes/s are the assumed values. $N = b^d \cdot (b^{(d-1)} - 1)/(b - 1)$.

It can be seen that already for trivial problems, e.g. just above $d = 4$, unacceptable search times arise. From the qualitative viewpoint, it does not even matter whether the number of calculated nodes per time is changed with factor 100. It can be seen that the complexity of the problem is $O(b^d)^1$ per node and therefore belongs to the category of NP-complete\(^2\) problems. The mean search time can be improved in comparison to breadth-first search, with e.g. heuristic approaches like $A^*$, with hierarchical planning, search in the space of plans or successive reduction of abstraction [Weld, 1999], but in worst cases a similar complexity as shown has to be faced. Even though the performance of deliberative task planners has increased [Weld, 1999, Russel and Norvig, 2003], it is still questionable whether they are efficient and robust enough for the application in real world domains [Russel and Norvig, 2003, Cao and Sanderson, 1996, Dario et al., 2004].

An alternative to deliberative systems are assembly planning systems, as introduced in [Cao and Sanderson, 1998]. Based on this idea, Martens developed a software-technical framework [Martens, 2003] that operates on pre-structured task knowledge, leading to the second essential development paradigm:

**Definition 2**

**Pre-structured Verifiable Task Knowledge (Essential Paradigm 2):** The task planner operates on the basis of so-called *process-structures*. They enable the pre-structuring of a certain task by context-related limitation of the task knowledge to a minimal subset. Based on this limitation, the logical correctness can be verified offline and a robust runtime system behavior can be guaranteed. Also, the process-structure-based approach forms the basis for task planning in real-time, which cannot be guaranteed by deliberative approaches under all circumstances. Despite pre-structuring, the process-structures are flexible enough to adapt to diverse objects. Thus, re-usability of the task knowledge in different scenarios is achieved.

Martens has shown how to realize meaningful daily life tasks within his framework; guaranteeing a robust and real-time suitable task execution based on verified task knowledge at the same time. On the one hand the novel idea of offline pre-structuring of tasks with process-structures paves the way for the creation of robotic systems with decisively reduced technical complexity. On the other hand, the task knowledge specification process is a complex procedure with many sub-steps of specification. Throughout the process, several tools are used and complex background knowledge about the underlying principles is required for successful completion of all programming steps.

---

\(^1\)The Landau symbol $O$ represents the maximum growth rate of an algorithm [Bachmann, 1894, Russel and Norvig, 2003]

\(^2\)No algorithm is known for NP-complete problems which is able to solve the problem with the help of a deterministic algorithm in polynomial time[Cook, 1971, Karp, 1972, Russel and Norvig, 2003]
Subject of this work is the development of a special process model for the development of semi-autonomous service robots. This has the impact to structure the complete development process, meanwhile taking the basic framework and the essential development paradigms into account. The process model will integrate new tools into the development procedure to facilitate each development step. The programmer is supported with the help of intelligent mechanisms that hide as many details as possible from him. Verification routines will be integral part of the process and will actively prohibit erroneous programming input. This avoids time-consuming iterative programming-verification-programming cycles and will lead to a much more user-friendly overall procedure. Improving the process of developing semi-autonomous service robots with a clearly structuring and guiding process model plays a vital role for the applicability of the promising fundamental concepts in daily practice.

1.4. Outline of this Work

- The current Chapter 1 reviewed the path of the vision of robots in service of mankind, summa-
  rizing its evolution from historical roots to today’s market products and future projections. On
  the one hand, nowadays, certain kinds of service robots are becoming more and more mature
  and enter the market. On the other hand, the reliable and successful operation of complex and
  universal systems is still a matter of research. A promising approach is to develop systems that
  operate semi-autonomously and with the help of pre-structured task knowledge.

- The subsequent Chapter 2 will supply further background information as well as discuss and
  summarize the open problems with respect to this approach. This leads to the derivation of
  the requirement of a process model that structures the development of semi-autonomous service
  robots.

- Chapter 3 covers the review of the state-of-the-art with respect to the previous problem state-
  ments. This means, the investigation results on existing programming methods for service robots
  and on process models are given and evaluated.

- Chapter 4 presents the general idea of the process model FRIEND::Process developed here and
  discusses its requirements. The subsequent chapters present the four main development steps of
  the FRIEND::Process:

  - Chapter 5 covers the scenario analysis;

  - Subject of Chapter 6 is the task knowledge specification and verification;

  - Chapter 7 discusses the development of the elementary functional units (skills) of a system; and

  - Chapter 8 explains the methods for final testing of skills and scenarios.

  - In Chapter 9 the process model is validated and evaluated with the help of its application for the
devolution of sample scenarios from the AMaRob project.

  - Finally, Chapter 10 summarizes the benefits that come along with the FRIEND::Process and
    outlines technical challenges for future work.
2

Background and Problem Statement

As shown in the last chapter, a typical representative of a service robot that is ideally suitable to be operated via shared-control concept is a rehabilitation robot - i.e. a robotic system where the human operator is accessible all the time and thus can be involved into task execution. The first part of this chapter is intended to provide more background information about rehabilitation robots. Initially, the technical evolution of this kind of systems is discussed and subsequently, the semi-autonomous rehabilitation robot FRIEND\(^1\) is presented as sample system of this category. Further on, typical application domains are exemplified with the help of the three representative support scenarios of the AMaRob project. These scenarios will also serve for evaluation purposes of the process model in Chapter 9. The discussion of backgrounds of this work is completed by summarizing the principle of task specification with process-structures. Eventually, the open problems that found the motivation for this work will be worked out.

2.1. The Evolution of Rehabilitation Robotics

The driving force for the development of rehabilitation robots from the 1980ies on is the intention to support disabled as well as elderly people during daily life activities. On the one hand there are clear demographical trends that strongly support this idea. Nowadays the more and more aging society already leads to an alarming imbalance between growing number of elderly and other persons not able to care for themselves and decreasing number of gainfully employed and thus potentially available people for caring; and in the future, this trend is forecast to worsen [Eisenmenger et al., 2006, Dorbritz et al., 2008]): The 11th coordinated population projection for Germany predicts twice the amount of elderly

\(^1\)Functional robot arm with user-friendly interface for disabled people, [IAT, 2009b, Martens et al., 2007]
people than younger people in 2050 compared to 2006, or to be more concrete, a quarter less people gainfully employed and a tripling of people older than 79 years. Based on these facts there are statements that in 2050 one care person will be responsible for twice as much patients as currently. The sophisticated integration of robotic assistants into the care processes could alleviate the imbalance, by supporting the care personnel with routine tasks and thus freeing resources for the original care task [Büsch er, 2008]. On the other hand, a robotic device is able to restore certain aspects of autonomy for a disabled person, who would not be dependent on the assistant’s help 24 hours a day then. The study in [Römer et al., 2005] analyzed the amount of time an impaired user is utilizing the manually controlled wheelchair-mounted robot manipulator MANUS during activities of daily living (ADL). It was found out that the system was used 1.5 hours in average. Based on the collected data potential cost-savings and economic benefits have been derived and have been extrapolated to savings of 14M - 37M EUR annually on the European level.

Historically, the first kind of rehabilitation robotic systems that have been realized were fixed workstation systems, being capable of executing preprogrammed tasks, like picking up paper from a printer or taking a book from a bookshelf. Mainly, two drawbacks could be identified for this first generation of rehabilitation robots: The robotic manipulators that had been mounted on the workstations were off-the-shelf industrial robots and the human machine interfaces (HMI) had been specially adapted to a certain user’s needs according to his disability.

The next generation of rehabilitation robots targeted at more flexible to use systems as special purpose wheelchair mounted manipulators, like MANUS [Mokhtari and Amni, 2001], RAPTOR [Mahoney, 2001] or Wessex/Weston robot arm [Hillman et al., 2002]. Different kinds of HMIs (as e.g. joystick, keyboard or space-mouse) allowed the user to control the manipulator’s end-effector either directly in Cartesian space or to directly control the joints of the arm. This control on low interaction level offers a great flexibility in use on the one hand but puts a decisively high cognitive load on the user on the other hand, leading to tiresome interaction efforts [Kawamura et al., 1995].

To improve the systems, the development of fully autonomous and mobile assistants was pursued at this point, e.g. by the projects MOVAR [der Loos, 1995], MOVAID [Dallaway et al., 1995] or Care-O-bot [May and Schäffer, 1999, Graf et al., 2002, Hans et al., 2002]. Even though impressive task demonstrations have been accomplished by these or similar systems, they come along with the problems for fully autonomous systems as stated in Chapter 1.

The first systems that took the concept of shared-mode control into consideration are e.g. ISAC [Kawamura et al., 1995], MUSIIC [Kazi et al., 1995, Kazi et al., 1997], KARES [Bien et al., 2001] and FRIEND I [Martens et al., 2001]. They provide preprogrammed autonomous skills that can be activated by the user as desired, like the visually controlled grasping of objects [Lang et al., 2000], weight controlled pouring of a drink [She et al., 2003a] or force-torque controlled drink serving [She et al., 2003b]. The different tasks have been realized as sensor-based closed control loop processes and are therefore robust against dynamic environmental changes. The large disadvantage that still persists is that the user has to guarantee that various pre-conditions for a certain skill hold, like for example the fact the distinctive objects features are within the current camera image before activating the camera-based object grasping skill.

As a result it turned out that a special software framework is necessary that is able to realize a sophisticated integration of the user’s cognitive capabilities into task execution. On the one hand the action sequences that are necessary to accomplish a certain task are complex but also the interaction effort for the user has to be minimized within the overall task execution process. The development and
application of an adequate framework has been conducted alongside with the realization of the second and third generation of the FRIEND system.

2.2. The Semi-Autonomous Rehabilitation Robot FRIEND

During the development of the first two systems belonging to the group of intelligent wheelchair mounted manipulators, FRIEND I & II, the basis of essential hardware and software concepts has been established [Martens et al., 2007]. Within the currently ongoing BMBF²-funded research project AMaRob (Autonomous Manipulator Control for Rehabilitation Robots, see also [IAT, 2009a]), lead by the Institute of Automation (IAT) at the University of Bremen, the third-generation of the rehabilitation systems, system FRIEND III, is being developing. Within the project, FRIEND shall leave the status as laboratory prototype and the upcoming step to a real product will be prepared (see also [IAT, 2009b]).

The AMaRob project consortium is an interdisciplinary team, consisting of technicians, designers and therapists and further representatives of various interest groups. They influence the development of the new generation system according to their specific background. Besides covering the various technical aspects, it is of great relevance to include design aspects as well as requirements from daily practice given by therapists to develop a rehabilitation robot that is suitable for daily living tasks.

The result, as depicted in Figure 2.1, is based on the platform Nemo®, which is the most recent development of an electrical wheelchair from Meyra [Meyra, 2008]. This platform is equipped with various adjustabilities of the seat for moving upwards or providing a stand up position for the user.

Figure 2.1. FRIEND III rehabilitation robot.

²German Federal Ministry of Education and Research, Bundesministerium für Bildung und Fororschung
2. Background and Problem Statement

The basic electrical wheelchair platform has been equipped with additional elements as depicted and detailed in the following:

**Robotic Manipulator:** The Light Weight Robotic Arm 3 (LWA3) is a serial product from Schunk built up from modular entities. The setup of the 7 degrees of freedom manipulator for the new FRIEND system has been selected to fulfill the weight and torque constraints given by the objects to be manipulated in the support scenarios. Therefore the arm provides the force of 60 N vertically downwards at the end-effector in the horizontally outstretched and thus worst case configuration, has a weight of 15.3 kg as well as a diameter of 120mm of the largest module at the robot-arm’s base. In order to guarantee for the user a comfortable navigation of the wheelchair in narrow passages, the robot-arm is mounted on a panning arm which can park the arm behind the seat as depicted in Figure 2.1. Further on the robot arm is equipped with an intelligent gripper with built-in anti-slipping mechanism that is able to detect the slipping of the gripped object and adapt the force accordingly. At the robot’s wrist a force-torque sensor is installed for the realization of force-torque-based reactive manipulative operations.

**TFT-Display on Panning Arm:** Similar to the robot arm the TFT-Display for visual interaction with the user is also mounted on a panning arm to avoid exceeding the original size of the wheelchair during navigation. A touch-sensitive surface on the display can be used by the care-personnel for direct interactions with the system.

**Intelligent Tray:** The central area of manipulator actions is the tray in front of the user of the wheelchair system. Similar to the predecessor system FRIEND II, the new system is equipped with an intelligent tray to acquire precise information about objects to be manipulated. The former weight-dependent tactile surface is replaced by a more precise system based on infra-red (IR) devices.

**Infra-Red Communication with Appliances:** An infra-red control unit for communication with various appliances in the robot’s environment is integrated underneath the pan-tilt-head unit of the camera system. Thus, an automatic door opening mechanisms in the refrigerator and the microwave oven, the configuration and control of the microwave oven itself, IR-controllable switches and power sockets or various consumer electronic components can be operated wireless.

**Computer System:** The computer system is mounted at the backside of the wheelchair. The mounting as depicted is still in a prototype state. In a future step, a cover will be developed to protect the system from splash water and to obtain a better protection of the cables from being ripped off during navigation in narrow environments. A computer with two Intel XEON® QuadCores with 2.33 GHz and 8 GB RAM has been chosen to satisfy the high computing power for robot vision as well as task and motion planning.

**Stereo-Camera System:** As depicted, the camera system is mounted at the top of the system on a pan-tilt-head, which itself is installed on a special rack. This rack is inclined so that the head of the user is not in the line of sight when observing the area of the user’s tray. A Bumblebee® 2 stereo-camera system with built-in calibration, synchronization as well as stereo-projective calculation features is used.

**Further Input Devices:** Besides the TFT-display as input device for the able companion, it is possible to integrate any input device into the system. An input device abstraction mechanism within the software control architecture decouples the input hardware from all other software modules. Thus, the input medium is chosen according to the kind of disability of the user or according to his preferences. The basic setup of the new generation FRIEND system provides for the following devices:
2.3. Rehabilitation Robotic Support Scenarios

- **Chin joystick**: A joystick as depicted in Figure 2.1 that can be controlled with the chin is an off-the-shelf device from Meyra that can optionally be provided for their wheelchairs. The user’s ability to move the head with a certain degree of freedom is a pre-requisite, which is fulfilled by the target patients group. With the joystick, the user is able to navigate the wheelchair, but also to use it for all other command inputs like task activation, user interactions, etc.

- **Mini joystick**: A mini joystick is integrated into the intelligent wheelchair tray on the right side (see Figure 2.1). This joystick is intended to be used by patients with remaining motor capabilities in the hand. It provides the same input functionality as the chin joystick.

- **Speech in- and output**: Additionally, the system is equipped with a commercial speech synthesizer and speech recognition. If desired, all system interactions that are displayed on the user monitor can be enhanced with voice outputs. According to the current interaction context, the speech recognition accepts spoken input command by the user.

- **Joystick for accompanying person**: Another input device for the accompanying person is the joystick at the backside of the wheelchair. For the sake of handling comfort it can be taken from the holding position and is connected to the system with a flexible helix cable.

The third generation FRIEND system will be tested in three different support scenarios in a stepwise procedure. First, therapists from the co-operation partner NRZ Friedehorst will evaluate it. Secondly, certain patients will be selected for extensive tests in training environments that are set up according to the three scenarios. The methods developed in this work will be evaluated with the help of these representative application scenarios and, without loss of generality, with reference to the robotic system FRIEND.

2.3. Rehabilitation Robotic Support Scenarios

The AMaRob project focuses on the development of three representative application scenarios that support disabled people in domestic as well as in working environments. The main goal is to prove suitability of the described kind of a rehabilitation robot for daily use. The beforehand mentioned Dutch study [Römer et al., 2005] claimed an average usage time of their kind of rehabilitation robot of about 1.5 hours. Based on this fact, AMaRob targets at the realization of 1.5 hours of autonomy within the three different application scenarios. However, as the system in the Dutch study has to be controlled on a very low level of abstraction, it can be expected that a system that provides own intelligence and that can be controlled on a higher level of abstraction, will be useful for much longer time spans than 1.5 hours in the future.

To provide the independence of 1.5 hours or more for the user, complete and complex chains of robot supported activities have to be realized. Their safe and successful execution has to be guaranteed even under varying environmental conditions like different lighting conditions or new obstacles in the workspace. These complex chains of actions are included in the AMaRob support scenarios that will be presented in this section.

A further important criterion for an economically successful rehabilitation robotic system is its potential to re-integrate disabled persons into the working environment. Though it is difficult to get the funding from public agencies in Germany for the private living fields, the funding for re-integrating disabled people...
into professional life are available (via Integrationsamt / Integration Agency [BIH, 2009]). Therefore, the developments within AMaRob are based upon three representative application scenarios with emphasis on the latter field - one from the domestic and two from the professional area.

2.3.1. AMaRob Scenario #1: Preparation of a Meal and Support for Eating

This support scenario enables the user to prepare and eat a meal. It is divided into five sub-scenarios as depicted with the help of a use case diagram in Figure 2.2. It represents the application spectrum of activities of daily living (ADL) within the AMaRob project and contains operations that are feasible to be supported by a service robot but still remains challenging from the perspective of intelligent robot vision and object manipulations.

The scenario can be seen as an extension of a simpler Pour in and serve drink scenario and the concepts can be transferred to other tasks in the domestic environment due to similarities within the contained object manipulations. However, several desirable support scenarios (e.g. related to personal hygiene) have been postponed for future system improvements, as they are considered to be too complex from the current technical viewpoint.

A special meal tray as depicted in Figure 2.3 has been designed. It should be noted that similar devices, e.g. a special spoon or plate, are also available for patients who are less disabled than the patients who need FRIEND. This meal tray here specially satisfies several requirements for being manageable by the robotic system:

- The handles (base handle, lid handle and the handle of the spoon) can be recognized robustly by the vision system
- The handles are designed to be graspable by the robot gripper
- The items that can be separated either by the manipulator (lid, spoon) or by the care personnel (plate) are located at unique positions with the help of latches.
- The meal tray is as light as possible
Further specialized tools that are included in the ADL scenario are a remote controllable microwave oven and a refrigerator that is equipped with an automatic door opener. The control of these devices takes place with the help of the IR-control unit as described in Section 2.2. A typical sample ADL scenario is illustrated in Figure 2.4, which shows the manipulator fetching an object (here a bottle) from the refrigerator. In the given example, the robotic system is controlled via brain computer interface (BCI), see e.g. [Lüth et al., 2007b, Lüth et al., 2007a] for more details about BCI control of the FRIEND system.

Figure 2.3. Special mealtray for handling a meal with a robot.

Figure 2.4. ADL sample scenario: The manipulator fetches an object from the refrigerator. FRIEND is controlled with brain computer interface (BCI).
2. Background and Problem Statement

2.3.2. AMaRob Scenario #2: Working at a Library Service Desk

The first AMaRob workplace scenario is situated at a library service desk. As the use case diagram in Figure 2.5 shows, it consists of a large number of sub-scenarios. However, the nature of these sub-scenarios makes a fast realization possible: Most of the sub-tasks deal with the handling of books within the limited workspace of the library service desk and are therefore very similar in their structure.

![Use case diagram with library sub-scenarios.](UDD_AMaRob_Library)

Figure 2.5. Use case diagram with library sub-scenarios.

A positive property of the library scenario is the fact that many communication interactions between the disabled user and the library clients are necessary. Therefore, this scenario is valuable from the viewpoint of re-integration of disabled people into daily life activities.

First tests within a library environment have been conducted with the FRIEND II system. According to the *Intelligent Environment*\(^3\) principle, the library desk is equipped with a scanner that reads standard bar-code from the book, so that the manipulator just has to move the book above it.

To meet the concern that this professional life scenario will not stand the usual stressful atmosphere at service desks in large libraries, it is intended to build up a test workplace in a small library; in parallel to the training environment that is set up in the Neurological Rehabilitation Center Bremen Friedehorst.

2.3.3. AMaRob Scenario #3: Quality Check of Workpieces

The second workplace scenario takes place in a rehabilitation workshop. The training environment will also be set up in the Friedehorst Center. Here, the less disabled patients usually conduct various assembly and disassembly tasks or do functional checks of workpieces. This occupation has positive influence on the patient rehabilitation process. The scenario is representative for many quality control tasks in industry. A typical category of handled items here are keypads of public phones. The patients examine their functionality so that correctly working keypads can be built into the phones again, whereas malfunctioning ones are disassembled and repaired.

---

\(^3\)An Intelligent Environment (IE) as meant here supports a robotic system with distributed smart components. It thus lowers the technical complexity of the robotic system itself. Consequently, the IE differs from the concepts *ambient intelligence* or *ubiquitous computing*, where only human being centered environments are designed. See also [Prenzel et al., 2006].
The main sub-scenarios that have been identified according to the use case diagram in Figure 2.6 are the **Visual check** of the keypads and the subsequent **Functionality check**. To enable a quadriplegic person to examine telephone keypads with the FRIEND system, the workplace environment is equipped with two smart tools, a special keypad magazine and test adapter for telephone keypads, as depicted in Figure 2.7(a) and Figure 2.7(b) respectively. The keypad magazine ejects one keypad with the help of a pneumatic mechanism when a button is pushed by the robot. The user is then able to grasp the pad with the robot and to conduct the visual inspection of the keypad. This step cannot be fully automated since each pad could principally require to be inspected differently. After successful completion of the visual check the user inserts the pad into the tester as depicted in Figure 2.7(b). The robot has to be commanded to push the green button to establish electrical contacts between the pad and the electronic test unit to conduct the functional test. During this test, the user subsequently pushes all keys of the pad and checks the functioning of each by inspecting a diode light. The third sub-scenario according to Figure 2.6, **Open tester**, serves as failure handling, since there is no extra opening procedure needed within the normal work flow.

![Use case diagram with sub-scenarios in the workshop environment.](image)

**Figure 2.6.** Use case diagram with sub-scenarios in the workshop environment.

![Magazine for telephone keypads, Test adapter for telephone keypads.](image)

**Figure 2.7.** (a) Magazine for telephone keypads, (b) Test adapter for telephone keypads.

In order to obtain a system with manageable complexity and in accordance with the essential development paradigm in Definition 2 (page 5), the task knowledge required for the scenario realization is pre-structured with the help of process-structures. As already anticipated in Section 1.3 the setup of process-structures is a rather complex procedure and requires a certain deeper understanding of them. The complexity of this specification process is hard to quantify without an extensive general discussion. In order to provide a simpler insight into the complexity, the specification process is illustrated with the help of examples in the following.
2.4. Task Specification and Execution With Process-Structures

Process-structures come in two levels of abstraction: Abstract process-structures (\(PS_A\)) are associated with the abstract (symbolic) layer of task-knowledge representation and describe tasks in a user-oriented and non-technical way. This level of abstraction is suitable for task-oriented programming purposes. Elementary process-structures (\(PS_E\)) represent the part of the task-knowledge that is required within the geometric and system-related (sub-symbolic) level. The information on this level of abstraction subsumes necessary hardware resources, like sensors and actuators, as well as elementary system operations (EEOPs).

The activity diagram in Figure 2.8 gives a general conceptual overview on task processing based on process-structures.

![Activity Diagram](figure2.8.png)

**Figure 2.8.** Overview on task processing based on process-structures. The process-structures are the driving force throughout all steps like initial monitoring, task planning and task execution with autonomous and shared-mode controlled actions.

### 2.4.1. Abstract Process-Structures

Each high-level task that can be executed by the system is associated with an abstract process-structure (\(PS_A\)). Abstract process-structures are based on AND/OR nets, as they are typically used in assembly planning [Cao and Sanderson, 1998]. They have been enhanced with first-order predicate logic facts [Russel and Norvig, 2003], so that a semantic interpretation of the information contained in the nets becomes possible. This is necessary for their integration within an overall control-architecture.
**Table 2.1.** Composed operators of the Fetch-cup process-structure in Figure 2.9.

<table>
<thead>
<tr>
<th>No.</th>
<th>Composed Operators</th>
</tr>
</thead>
</table>
| 1   | AOP: GraspObjectInContainer(G, C, B)  
     DOP: DepartFromContainer(G, C, B) |
| 2   | AOP: PlaceObjectInside(G, C, B)  
     DOP: GetObjectOutside(G, C, B) |
| 3   | AOP: PutDownObject(G, C, T, FreePlacePos)  
     DOP: LiftObject(G, C, T, FreePlacePos) |
| 4   | AOP: GraspObject(G, C, T)  
     DOP: Depart(G, C, T) |

Similar to the STRIPS-planner [Fikes and Nilsson, 1971] the facts assign pre- and post conditions to the operations included in the process-structure.

Figure 2.9 depicts a simple version of a process-structure that is assigned to the task request *Fetch cup*. A $PS_A$ contains a symbolic representation of all objects that are involved in the task - also denoted as *task participating objects* (TPO). TPOs are combined to object constellations (OCs) which describe the different physical contact states of the objects throughout the task execution process. Due to the $PS_A$'s origin in assembly planning, the contained OCs are connected via assembly or disassembly operations (AOP, DOP) or internal state transitions (IST). The semantics of the first two operation types is self-explaining and the latter version is used to represent internal changes within a single OC without changing the components being in physical contact, for instance moving a book on a table without raising it. TPOs in this process-structure are the gripper of the robotic system (G), a box-like container (B), which is an abstract representation of a fridge, cupboard, shelf or similar, a cup (C) and a table (T). To illustrate the meaning of the specified object constellations some examples are given: $OC_1 = "the cup is inside the box", OC_3 = "the cup is grasped by the gripper inside the box" or $OC_8 = "the cup stands on the table". Besides the mere enumeration of contained objects a set of first order predicate logic facts is assigned to an OC to represent the objects’ states as well as their relationships.

![Diagram of process-structure related to the task FetchCup. The encircled numbers denote pairs of applicable composed operators (COPs), as given in Table 2.1.](image)
2. Background and Problem Statement

In total the TPOs are combined to 8 different object constellations which are connected via 4 pairs of composed operators (COP) as specified in Table 2.1. As listed there, this simple sample scenario only consists of AOPs and DOPs. Naming the operations composed operators has its reason in the fact that they constitute a high abstraction operation specification and will be decomposed into elementary process-structures as described further on. Besides the TPOs, COPs require symbolically represented sub-symbolic information as parameter. An example in this case is the last parameter in the COP pair 3 (FreePlacePos).

Certain sets of OCs in Figure 2.9 are marked as initial situation respectively target situation. A situation in a $PS_A$ contains OCs that uniquely include all TPOs and are part of the situation graph as defined in [Prenzel, 2005] (see also Appendix D). This means they have to define a valid intermediate state that is transformable via the COPs as specified for a certain process-structure. The initial situation $S_I$ is the result of the initial monitoring process and the target situation $S_T$ has a fixed association with the user’s task request. In the example given, the cup is located in the box. The gripper and the table are single isolated objects in the initial situation. In the target situation, the cup has been placed on the table, the box is left empty and the gripper is returned to a free location in the workspace. In this setup the following finite sequence of operations that transforms the initial situation into the target situation will be the result of the task planning process on the abstract level:

1. GraspObjectInContainer(G, C, B)
2. GetObjectOutside(G, C, B)
3. PutDownObject(G, C, T, FreePlacePos)
4. Depart(G, C, T)

2.4.2. Elementary Process-Structures

To illustrate task knowledge specification on the elementary level the assembly operation GraspObjectInContainer from the previously discussion $PS_A$ “Fetch Cup” is decomposed into an elementary process-structure ($PS_E$). A $PS_E$ gets the input parameters as specified in the COP’s parameter list (i.e. G, C and B in this case).

The system programmer has the following tasks to accomplish during $PS_E$ specification:

- Specify the resources of the system to be used,
- specify the elementary operations (skills) to be executed,
- determine the required flow of control and sub-symbolic data and
- establish the connection to the abstract task knowledge layer via setting the COP’s post-facts.

---

4The terms EEOP and skill represent the same, namely the algorithmic implementation of a basic system functionality, e.g. a manipulative action or a machine vision operation. The different terms have historical origins at the IAT. The term EEOP is used within the task planning domain and the term skill is used within the reactive system layer. The historical origin for EEOPs is [Martens, 2003] and for skills it is mainly the software framework Smartsoft [Schlegel and Wörz, 1999]. Within the domain of function block networks, one function block represents one EEOP/skill.
2.4. Task Specification and Execution With Process-Structures

To be able to model parallel execution of system operations and to verify $PSE$ formally on a well-defined mathematical basis, Petri nets are used. The subject of verification is the correctness of the afore-mentioned items as well as to detect possible resource conflicts, deadlock situations, unreachable targets or further modeling errors.

For illustration purposes, a decisively simplified version of the resulting Petri net for the $PSE$ GraspObjectInContainer is depicted in Figure 2.10. As indicated by the dashed lines the complete upper part of the $PSE$ has been left out. However, the given Petri net still obviously reveals the complexity of the specification on the level of elementary process-structures.

The transitions in the Petri net, which model the control flow, are arranged in pairs (or tuples) according to the set of return values that are assigned to a certain elementary executable operation (EEOP). According to their functionality EEOPs are grouped as different types, which are indicated with the prefix in the transition name. $DC$ stands for "direct control" of an actuator, $USER$ for "user interaction" and $REAC$ for "reactive operation", i.e. an EEOP which is implemented as closed control loop that couples sensors and actuators. Furthermore, there is a fourth category of EEOPs, the "monitoring operations" with the prefix $MON$, but they are not part of the depicted segment of the $PSE$. The modeling of resource usage is done with the help of fact places like $IsGlobalAvailable(SCam)$, which indicates whether the stereo-camera system is currently in use or is available. The $DataAvailable()$ place indicates the availability of sub-symbolic data and thus models the data flow. Finally, the interconnection to the abstract process-structure is established with facts (e.g. $HoldsNothing(G)$). With the help of these post- and pre-facts, the connection and consistency between the sub-symbolic control level ($PSE$) and the abstract level ($PSA$) are guaranteed.

The stringent application of the presented application specific syntax within the $PSE$–Petri nets enables offline-verification of the above listed issues on a well defined mathematical basis.

From the algorithmic viewpoint task planning on $PSE$ level is equivalent to $PSA$ level. There is an isomorphic transformation of a $PSA$ into a Petri net and thus task planning takes place on Petri net level in both cases. For this purpose, the Dijkstra algorithm is applied to find a sequence of operations (fired transitions) between the initial situation and the target situation (which both correspond to certain markings in the Petri net). This approach allows to assign execution costs to the operations to influence the selection of different operations between start and target configuration. The algorithms searches for the path with least costs. In the case of equal costs for all operations, the shortest sequences of operations is chosen.

Based on the planned operation sequence, the task execution takes place. During this procedure, a single transition from out a pair (or tuple) is fired according to the actual EEOP return value. If the return value matches the pre-planned one, the next EEOP of the plan is chosen. Otherwise, re-planning has to be performed.

Here, according to the given state of the Petri net, the manipulative operations $DC.OpenGripper$, $REAC.MoveToObjectInContainer$ and $DC.CloseGripper$ will be executed. Within the reactive manipulative EEOP $MoveToObjectInContainer$, a motion planning algorithm is started, which operates on the previously (during monitoring) generated environmental information. Due to the structure of the task-knowledge contained within the $PSE$ and the usage of the Dijkstra algorithm with weighted operations, the result of the planning process is that autonomous EEOPs are executed first and user interaction EEOPs follow in case the system fails (this general idea is also illustrated in Figure 2.8). Furthermore, manipulative EEOPs can be interrupted by the user, e.g. in case he can already foresee a possible collision. This is indicated by the ":[user interrupt]" transition in Figure 2.8 or the
2. Background and Problem Statement

Figure 2.10. Elementary process-structure of COP GraspObjectInContainer as Petri net.

The concept of process-structures can be summarized as follows:

2.4.3. Summary on Process-Structures
2.5. Integrated Process with Model and Tool Support Required

Definition 3

The \( \text{PS}_A \) level (abstract/scenario level)

- defines what happens, e.g. to fetch an object,
- is configured by non-technical personal or the user himself.

The \( \text{PS}_E \) level (elementary/system level)

- defines how something happens, i.e. the data and control flow is defined and the resources to be used,
- is configured by a system-programmer.

Informally, the process model FRIEND::Process, developed in this work, and the embedded concept of process-structures, are put into context as follows:

Statement 2 Process-structures can be considered as a "process model" for the robot to act, whereas the FRIEND::Process is a process model for developers of service robotic systems.

2.5. Integrated Process with Model and Tool Support Required

Within the framework developed by Martens, the different steps of task knowledge specification with process-structures are not or rather loosely coupled with each other. The usage of a set of different tools as well as manual programming are required for the input of new task knowledge, i.e. the enhancement or new introduction of scenarios. The potential impact of automation and integrated verification is high for this process. To give an example, the verification routines for elementary process-structures that have to be executed manually so far, can be integrated into a special tool for \( \text{PS}_E \) programming and thus can take effect immediately during the task knowledge specification process. Consequently, the necessity of time consuming and cumbersome iterative correction of mistakes becomes obsolete.

Also, as illustrated in the summary about process-structure based programming in Section 2.4, the nature of the specification processes is currently not only laborious and error prone, but takes place on a very high level of abstraction in the case of \( \text{PS}_A \) specification, and on a very detailed level in the case of \( \text{PS}_E \) specification. As the programming of abstract process-structures deals with physical contact states of involved objects, a pictorial representation of these objects will lead to a much more intuitive specification process, where currently the programmer has to maintain an imaginary embodiment of the modeled elements in his mind. On the level of elementary process-structures the many details of Petri nets can be hidden from the user by introducing a more user-friendly specification method without losing the opportunity of mathematical verification on the basis of Petri nets. The elimination of all existing drawbacks of the process-structure based programming approach is crucial for the successful applicability of this method in practice and is therefore the central motivation for the development of an ergonomically to use integrated programming system.

Moreover, the framework, as initially developed until 2003, solely has been used in combination with a simulation system to evaluate its effectiveness. In the subsequent development steps the enhancement
2. Background and Problem Statement

of this framework and its application on a real robotic system (FRIEND) have taken place. It turned out that the pure task knowledge engineering steps are one of four categories of actions that are required to develop a complete semi-autonomous rehabilitation robot. Together with a scenario-driven development approach the complete process consists of:

- Scenario Analysis
- Task Knowledge Specification and Verification
- Skill Development
- Skill and Scenario Testing

Altogether, a complex development process evolves, including numerous single steps. Many of these steps are coupled to each other, via the artifacts and implementations they produce or enhance. To manage the overall complexity and to improve the complete procedure chain, this work targets at the development of an integrated process model. This process model will still include a certain set of development tools; however, they are no longer loosely coupled, but directly interact with each other. Furthermore, as to be justified, the complete process will be supported by a model repository, which is managed consistently by a state-of-the-art CASE tool. In order to take the essential development paradigms into account (Definitions 1 and 2) and to achieve a tight integration of involved tools and development artifacts, a problem specific instead of a general purpose process model is required. Nevertheless, as to be argued in the following chapters, the specialized process model can be derived from a generic one.

**Process Requirement 1** A special integrated process model is required, tailored to manage the complexity of the development of semi-autonomous service robots, to unify and guide this process, as well as to achieve a tight integration of involved tools and development artifacts.

---

5Computer Aided Software Engineering
State-Of-The-Art

‘Model-driven engineering (MDE) technologies offer a promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively.’ (Douglas C. Schmidt)

The state-of-the-art review with relevance to this work is performed under two aspects: On the one hand issues are considered that are related to process models in general, and on the other hand methodological issues that will be embedded into the process model are inspected. First, general background information about process models is gathered. Special emphasis is given to model-driven development techniques and a related recently developed general process model. This will form the basis for the process model elaborated in this work, which is specially tailored to the requirements for the development of semi-autonomous service robots. Objective of this special process model is to organize and improve the programming of a service robot by using process-structures as central development artifacts. Therefore, the second part of the state-of-the-art discussion covers alternative and related programming methods, in order to be able to compare the methods used in this work to existing approaches as well as to extract existing concepts or parts of them for the approach pursued here.

3.1. Process Models

Process models are applied in various fields where complex procedures have to be managed. Concerning software development, the process model can be seen as a plan how to create a complex software system. This plan divides the development process in manageable phases, which are limited with respect to time and extent. There are various types of process models, with for example sequential or iterative nature. Among experts there is no common agreement about the optimal process model. In the following the various models will be discussed along with the most recent trend in software development processes: The model driven engineering. Finally, the requirements for the process model that guides the development of semi-autonomous service robots are worked out.
3. State-Of-The-Art

3.1.1. Model Driven Engineering / Development

In November 2000 the Object Management Group (OMG, [OMG, 2008]) released the Model Driven Architecture (MDA) initiative coming along with the most recent global trend in the field of systems and software engineering: The Model Driven Engineering (MDE). Basic ideas of MDA subsume recent approaches such as generative programming, domain specific languages, model-integrated computing, generic model management, software factories, etc.

The supporters of this new trend see many benefits in a move from code-centric to model-based approaches. The main arguments that shall be detailed in the following are:

- There is a tremendous lack of integrated view
- Defects are found late
- Effective communication technique is required
- Complex development process models require model support

3.1.2. Key Issues to be Tackled via MDE

Lack of Integrated View (“Big Picture is Hard to Grasp”)

In [Schmidt, 2006] the author analyzes why “Model-driven engineering (MDE) technologies offer a promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively”.

He writes that even though third-generation programming languages (e.g. C++, C#, Java) and reusable application framework platforms (which provide middleware services like event notification or distributed resource management) matured over time and nowadays reduce the complexity of the software development process decisively, various severe problems still remain. It is argued that the core reason is the complexity of the mostly used middleware platforms. Developers have, amongst other things, to cope with

- Numerous intricate dependencies and subtle side effects,
- Adaptation of application code to different or new versions of the same platform,
- System deployment, configuration and quality assurance.

Consequently, software developers are only familiar with a small subset of the platform they are using and they often cannot focus on strategic architectural issues like system-wide correctness and performance. It easily becomes impossible to trace all possible side effects due to changed user requirements or changes with respect to the language or platform. To summarize, it can be stated that the "big picture" of a software system or a part of it is hard to grasp (as it is called "lack of integrated view" in [Schmidt, 2006]). Therefore, often suboptimal solutions are implemented that unnecessarily violate
key architectural principles, duplicate code and complicate system evolution and quality assurance.

Defects are Found Late

Another motivation, which directly has an economical effect on a software project, is the fact that defects are usually introduced early but detected in a very late development phase. Figure 3.1 qualitatively shows that the number of defects introduced is highest during specification phase, whereas the most defects to be detected is highest in the stage of system testing.

![Figure 3.1](image)

This problem is not a recently discovered one, as the graphic in Figure 3.1 appeared within [Ramamoorthy et al., 1984] in 1984. Rather, the reason lies in the way of traditional software development as shown in Figure 3.2: Here the so-called waterfall model [Wikipedia, 2009] is shown, which assumes that the different stages of software development are serialized like the flow downwards a waterfall. As depicted the problem is that only at the end of the development the software can be tested and errors can be detected. This is visualized in the form of a wall that has to be conquered during the process of software creation. After finding mistakes, there is a new product iteration cycle necessary in many cases and in the worst case, the prototypes have to be re-created completely, since decisive design mistakes may turn out to lead to hardly solvable problems. Therefore, the prototypes created within the traditional way of development are often "throw-away" prototypes. In other cases the initially unforeseen problems can be fixed with a lot of extra manpower, lead to postponed product delivery or even cause a canceling of a complete project as the worst-case consequence. Reference [McConnell, 1996] estimates that "A requirements defect that is left undetected until construction or maintenance will cost 50 to 200 times as much to fix as it would have cost to fix at requirements time". Further clear figures for the stated scenarios are to be found in [Krasner, 2007].

Efficient communication technique is required

In the previous sections the term Model has been used without further specifying its meaning. The modeling takes place either with SysML on the system level or with UML on software level. SysML means
3. State-Of-The-Art

**Systems Modeling Language**\(^1\). SysML is an extension of UML, which is the abbreviation for *Unified Modeling Language*\(^2\). SysML as well as UML have been approved as a standard by OMG [OMG, 2008] and are widely used modeling techniques for either system or software design. Both consist of an integrated set of diagrams. Figure 3.3 shows the SysML diagram taxonomy, including the relations to the parent diagram sets from UML. A good online summary of the UML taxonomy is available at [Borland Software Corporation, Inc., 2003].

![SysML Diagram Taxonomy](http://www.sysml.org)

**Figure 3.3.** SysML diagram taxonomy, from http://www.sysml.org.

The diagram sets in SysML as well as in UML have been developed with the purpose to help the system and software developer to accomplish the tasks like

- Specification
- Visualization

---

\(^1\)http://www.sysml.org
\(^2\)http://www.uml.org
3.1. Process Models

- Architecture Design
- Construction
- Simulation and Testing

Carrying out the listed tasks in a visual manner via diagrams helps the developer to concentrate on The Big Picture. Properly constructed diagrams and models are an efficient communication technique as the following example (taken from: [Telelogic, 2007]) shows:

**Scenario 1 - Conceptual collaboration in text:**
Developer 1:
"Ok. That’s what we need to do: Thread A will pass event X to thread B and that will change B’s state to Running from what it was before which was Idle. When B changes to Running it will send back an event Y to A and then wait for 2 seconds and then go back to Idle. Thread A will have started in Idle also and will go to Run after B sends back event Z which happens after the 2 seconds before going to Idle. All this should happen in less then 5 seconds."

Developer 2:
"Huh?"

**Scenario 2 - Conceptual collaboration via model:**
Developer 1:
"Here, look at this sequence diagram!" (Figure 3.4)

![Sequence Diagram](image)

Figure 3.4. Example how to express a concept via model diagram.
Software Development Process Models with Model Support are Required

The waterfall process model for software development has its attractiveness in its simplicity. The assumption that the development of software goes straight forward in a sequential manner allows a serialized and thus easier scheduling from the management perspective. However, this is also the main pitfall. A strongly serialized processing of the different phases never takes place within larger and more complex software projects. As e.g. [Douglass, 1999] states from daily development practice: "[...] the artifacts of any phase cannot be complete until they are elaborated in subsequent phases and their problems identified. As a practical matter, experience has clearly shown that analysis cannot be complete until at least some design has been done, and design cannot be complete until some coding has been done, and coding cannot be complete until testing has been done."

Thus it is obvious that more complex and more sophisticated approaches of software development are required. Theoretically, many process models have been proposed for a longer time, e.g. the V-Model [Wikipedia, 2009k], Agile software development [Wikipedia, 2009b], Iterative and incremental development [Wikipedia, 2009g], Extreme Programming (XP) [Wikipedia, 2009e] and many more. However, only within the recent years suitable CASE-tools have been elaborated so far that they thoroughly support complex development approaches. Consequently, model-driven software development processes are becoming mature and suitable for practical application. To evaluate a certain process model and the level of model support, evaluation schemes have been developed, e.g. the Modeling Maturity Level [Kleppe et al., 2003].

Modeling Maturity Levels is a classification system whose levels characterize the role of modeling in a software project. The concept shows resemblance to the way software processes are rated with the Capability Maturity Model (CMM, [Humphrey, 1989]).

There are 6 levels:

- **Level 0: No Specification:** the specification of software is not written down. It is kept in the minds of the developers.

- **Level 1: Textual Specification:** the software is specified by a natural language text (be it English or Chinese or something else), written down in one or more documents.

- **Level 2: Text with Models:** a textual specification is enhanced with several models to show some of the main structures of the system.

- **Level 3: Models with Text:** the specification of software is written down in one or more models. In addition to these models, natural language text is used to explain details, the background, and the motivation of the models, but the core of the specifications lies in the models.

- **Level 4: Precise Models:** the specification of the software is written down in one or more models. Natural language can still be used to explain the background and motivation of the models, but it takes on the same role as comments in source code.
• Level 5: Models only: the models are precise and detailed enough to allow complete code generation. The code generators at this level have become as trustworthy as compilers; therefore no developer needs to even look at the generated code.

With the currently available tools it is nowadays feasible to work on the level 4 or 5. Developers have thus a real chance to benefit from the omnipresent, integrated and consistent model support and to have an integrated view into complex system, detect defects early through model execution, use effective diagram-based communication techniques and finally to manage complex processes with the help of the model-driven techniques. The current state-of-technique is visualized in Figure 3.5.

One of the currently leading solutions for system and software development is Telelogic Rhapsody®, which will be used throughout this work. A software development process that is based (but not limited) to the usage of Rhapsody is the HARMONY process proposed by Hoffmann and Douglass. It is subdivided into the systems engineering part (HARMONY-SE, [Hoffmann, 2006, Hoffmann, 2008]) and the software engineering part (HARMONY-SWE, [Douglass, 2007a, Douglass, 2007b]). Figure 3.6 gives an overview on the HARMONY process.

Core concept of HARMONY is the model support through the whole life cycle of a software product. This idea leads to the term of Model Driven Development (MDD), as the model drives the process in all stages. With development support via an all-time consistent model it becomes feasible to

- Iteratively enhance prototypes (evolutionary improved ones instead of throw-away prototypes),
- Build executable models in early stages of development,
- Test in early stages of development (e.g. design-level debugging) and thus
- Detect errors early or prevent their introduction and propagation, finally to
- Maintain an all-time consistent documentation, since the model is the (core of) documentation.
3. State-Of-The-Art

Figure 3.6 uses the classic "V"-diagram [Wikipedia, 2009k] in combination with the statechart notion to visualize the development process. The iterative characteristic of this process is shown via "high-level interrupt" due to system changes. This means, each change will restart the process in the requirements level.

### 3.1.3. The HARMONY Development Process

The tight support of a model throughout the entire process of system development according to HARMONY makes this process ideal to derive the specialized process model for the development of semi-autonomous service robots from it. Especially the Systems-Engineering part of HARMONY (HARMONY-SE) is an exemplary representative of a process model, as it clearly gives a structured guidance through development steps and their in- and outputs. For this reason a summarizing overview of HARMONY-SE is given within this section. More details about HARMONY (SE+SW) can be found in the references [Hoffmann, 2006, Hoffmann, 2008, Douglass, 2007a, Douglass, 2007b].

Figure 3.7 depicts the overview on HARMONY-SE, which can be sub-divided into the three phases Requirements Analysis, System Functional Analysis and Architectural Design. As depicted, each phase is fully model supported. Within the requirements analysis, the grouping of requirements into use cases takes place. The system functional analysis is a use-case-based identification and verification/validation of operational contracts (OpCons), also known as set of functional requirements. During the system architectural design, the previously identified operational contracts are allocated to functional or physical subsystems, including the definition of subsystem interfaces. The subsystem architectural design is carried out analogously to that one on system level. This process is based on the specification of operational contracts, it has to be characterized as function driven concept. Core elements that drive the whole process through the complete design path are the requirements related test scenarios. The result
of HARMONY-SE is the identification and derivation of required system functionality, identification of system states and modes as well as subsystems identification and requirements allocation to them.

Figure 3.7. HARMONY-SE overview [Hoffmann, 2006].

Requirements Analysis

This phase starts with the capturing of the requirements and their grouping (clustering) into use-cases. There are two types of requirements that have to be analyzed: Functional requirements, which define what the system shall do and Quality of Service (QoS) requirements, often called non-functional requirements, which define how well the system shall perform, i.e. they specify quantifiable performance measures. Each use-case describes a certain operational aspect of the system under development, i.e. the system’s behavior as perceived by the users is specified, as well as the message flow between users and the use case. The final step in the requirements analysis is the linking of the use cases to requirements and their checking for complete coverage.

System Functional Analysis

The previously identified functional requirements are now translated into a coherent description of system functions via operational contracts and validated through model execution. Figure 3.8 describes the complete workflow.

The so called black-box use case analysis starts with the definition of the static system structure with the help of the SysML artifact structure diagram (i.e. a block definition diagram is used here). The blocks to be set up are the actors that have been identified previously, as well as the system itself.

As Figure 3.9 points out, depending on the available information, there are several alternative approaches as how to build the use case model. The output of all three alternatives is the same, as the scenarios are described via sequence diagrams. The essential scenarios are merged into one common use case.
3. State-Of-The-Art

Figure 3.8. Functional analysis workflow [Hoffmann, 2006].

description via activity diagram, where each block describes one operational contract from a sequence diagram. Then, the blocks are populated with ports and interfaces and finally, statechart diagrams are specified for the blocks to describe their dynamical behavior. Starting with the scenarios (i.e. using alternative 1) is preferred if the requirements are available in written form. Alternative 2, i.e. starting with specification of the functional flow via activity diagram, is common if the systems engineers have to elaborate the requirements on their own. The third approach is suitable for strongly state-based systems. Also, if no underlying functionality can be identified in this case, the set up of an activity diagram is omitted.

After completion of building one black-box use case model, it is validated and verified together with its underlying requirements via model execution. Here, the identified scenarios, i.e. sequence diagrams, serve as stimuli. The described steps are performed for each use case, but the initially created black-box activity diagram and statechart is extended instead of creating new ones.

Subsequent to building the models for all use cases, the validation and verification takes place in a two-fold manner: First the extended black-box system model is validated and verified via model execution
using the new use case scenarios as stimuli. Afterwards, the collaboration of the implemented use cases is verified through regression testing.

The final result of the system functional analysis is a black-box system model with verified and validated operational contracts, which represent the underlying functional requirements. Therefore, the operational contracts are linked to the high-level system requirements as a final step of the system functional analysis phase.

The essential results of HARMONY-SE on functional level are described in Figure 3.10 and in Table 3.1.

![Figure 3.10. SysML artifacts utilized in functional analysis [Hoffmann, 2008].](image)

<table>
<thead>
<tr>
<th>Use Case Diagram</th>
<th>Defines the &quot;Table of Contents&quot; by packaging requirements.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Defines the system context.</td>
</tr>
<tr>
<td>Block Definition Diagram</td>
<td>Defines at the functional analysis level the context of the use case models.</td>
</tr>
<tr>
<td></td>
<td>Defines at the system design level the context and the system hierarchy.</td>
</tr>
<tr>
<td>Use Case</td>
<td>Packages’ requirements and gives them a context.</td>
</tr>
<tr>
<td>Activity Diagram</td>
<td>Shows at the functional analysis level the functional flows through the use case.</td>
</tr>
<tr>
<td></td>
<td>Shows at the system design level the allocation of decisions and operations across the system/subsystem architecture.</td>
</tr>
<tr>
<td>Sequence Diagram</td>
<td>Shows at the functional analysis level the allocation of operations and the message interactions between use cases and associated actors. At the system design level, it shows the allocation of operations and the message interactions between systems, subsystems, and actors.</td>
</tr>
<tr>
<td>Internal Block Diagram</td>
<td>Shows at the functional analysis level the functional flows through the use case.</td>
</tr>
<tr>
<td></td>
<td>Shows at the system design level the allocation of decisions and operations across the system/subsystem architecture.</td>
</tr>
<tr>
<td>Block</td>
<td>Defines the structural elements of the model. At the functional analysis level, a block is defined for each use case.</td>
</tr>
<tr>
<td></td>
<td>At the system design level, blocks are used to define the system and subsystem structure.</td>
</tr>
<tr>
<td>Statechart Diagram</td>
<td>Defines the state-based behavior of blocks.</td>
</tr>
</tbody>
</table>

Table 3.1. SysML artifacts in functional analysis [Hoffmann, 2006].
State-Of-The-Art

Architectural Design

This design phase is divided into the system architectural design and the sub-system architectural design. The applied procedures are analogous. As Figure 3.11 shows, the system design starts with the definition of the contained physical sub-systems via structure diagrams, containing actors and the system as main elements as well as the system’s sub-systems constituting the parts. Next, the black-box operational contracts from the system functional analysis are allocated to the subsystems with the help of so-called white-box activity diagrams. Basically, it is a copy of the black-box activity diagram, but with additional equipment of swim-lanes according to the different subsystem parts. Additionally, white-box sequence diagrams are specified as a decomposition of the black-box counterparts. Their purpose is to identify the interfaces of the subsystems. After adding statechart definitions to each subsystem, the validation and verification with model execution can be done. Besides this functional verification, to complete the architectural system design, an analysis of the performance and safety requirements has to take place. This typically includes failure modes effects analysis (FMEA, [Haapanen and Helminen, 2002]) and mission critically analysis.

![Figure 3.11. System architecture design work flow [Hoffmann, 2006].](image)

At the end of the system architectural design phase the HW/SW assigned operational contracts are linked to the original requirements. For each physical subsystem the following documents are generated from the deployment model as hand offs to the subsequent hardware and software design:

- **HW/SW requirements specification**
- **Logical interface control document (N2 chart, see [Hoffmann, 2006])**
- **Subsystem/subsystem component test vectors derived from system-level use case scenarios**
3.1.4. From Generic to Problem Specific Process Models

HARMONY is an attractive process model, since it provides a clear and very detailed stepwise guidance through system development. Exemplarily it defines the process steps, the in- and output artifacts of each step and concrete instructions how to use and structure the model entities of the system - and thus satisfies all requirements of a process model for real practical application. However, HARMONY is a generic process model, intended to be applied to the development of generally complex systems. The level of details in all prescribed steps has their justification for large systems and will also pay-off in these cases. The requirements for a process model for the development of semi-autonomous service robots indicate the necessity of a problem-specific process model. Thus, additionally to Process Requirement 1 in Section 2.5 the following is claimed:

**Process Requirement 2** The development is scenario-driven. A scenario analysis and its effects on the complete process have to be an integral part of the process model.

**Process Requirement 3** Process-structures are central elements of the development approach. Their structured development as well as their influence on subsequent development steps have to be respected.

**Process Requirement 4** Even though complex, most of the procedures, data structures and software infrastructures that are designed and implemented throughout the development of semi-autonomous service robots within the specialized framework are uniform. This uniformity and structured characteristic has to be formalized and will form the basis of the process model, especially with the focus onto the skill layer and its interfaces.

Consequently, a process model is developed in the following that is tailored according to these requirements, and is thus problem specific. The basic concepts of HARMONY are adapted, so that the resulting process model can be realized fully model-driven throughout the whole development process.
3. State-Of-The-Art

3.2. Methods for Programming Service Robot Behavior

One representative of a service robotic system for domestic use that is already available on the market is the Roomba vacuum cleaning robot sold by iRobot [iRobot, 2009]. Since this system has been designed for a single application, it is no great effort to equip it with a simple and easy to use control panel. The first generation of Roomba systems had to be "programmed" in the sense that the size of the room to be cleaned had to be selected via three room size buttons (Small, Medium, Large) [Wikipedia, 2009i]. Meanwhile, the next generations of this system calculated the room size on its own and the user principally is not bothered with any "system programming" at all.

It is obvious that future more versatile service robotic systems to be used for various different applications require more complex programming procedures. Within this section, the state-of-the-art in programming of service robots shall be reviewed. On the one hand, this review serves as background information, but on the other hand the goal of this chapter is also to evaluate, whether existing approaches or parts of them are applicable for the development of the programming system within this work.

Nowadays, most service robotic systems that are able to perform complex tasks are used to be programmed in a text-based manner and only a very few approaches use high-level abstractions [Biggs and MacDonald, 2003, Ekvall et al., 2006]. However, as argued in [Ekvall et al., 2006], a graphical mid-level language is a suitable form to instruct a service robotic system. It is an intermediate approach between high-level instruction in spoken languages (which require complex translation into low-level languages that are understood by a robot) and tedious manual programming in low-level instruction approaches like programming in C++.

The authors of [Boshernitsan and Downes, 1997] give an overview of visual programming languages (VPL), starting from historical facts and eventually discussing the state-of-the-art in that field in 1997. Even if their report is about one decade old, they compile some interesting facts about visual programming languages and namely the primary motivations for most research into VPLs:

> ‘From cave paintings to hieroglyphics to paintings of Campbell’s soup cans, humans have long communicated with each other using images. The field of visual programming languages asks: why, then, do we persist in trying to communicate with our computers using textual programming languages?’

The general motivation for using visual programming languages can thus be extracted to the following facts: *Many people think and remember things in terms of pictures. They relate to the world in an inherently graphical way and use imagery as a primary component of creative thought* [Smith, 1975].

Additionally, it is argued that textual programming languages have proven to be rather difficult for many creative and intelligent people to learn to use effectively. Reducing or removing entirely the necessity of translating visual ideas into somewhat artificial textual representations can help to mitigate this steep learning curve problem.

The programming environment Pygmalion developed by Smith [Smith, 1975] is rather far away from being suitable for the programming of robot behavior, however it constitutes the first major milestone in the genesis of VPLs. Pygmalion embodied an icon-based programming paradigm in which the user created, modified, and linked together small pictorial objects, called icons, with defined properties to perform computations.
3.2. Methods for Programming Service Robot Behavior

In the following the state-of-the-art of robot behavior programming will be reviewed. According to the contemplations and citations so far, the focus will essentially be on graphical programming methods.

3.2.1. Task Description Language

The Task Description Language (TDL) [Simmons and Apfelbaum, 1998] is an extension of the programming language C++ and thus a text-based approach of robot programming. It has been proposed by Simmons and Apfelbaum to implement task level control, including task decomposition, concurrent activities, environment monitoring and exception handling. TDL has been developed based on the Task Control Architecture [Simmons, 1994].

The basic task representations used in conjunction with TDL are task trees which encode parent/child relationships and synchronization constraints between nodes, which, in turn, are parameterized pieces of code. A sample task tree is shown in Figure 3.12. Task trees can modify dynamically during runtime, e.g. to use current perception data to parameterize a certain action or to enable a specific exception handling procedure. The synchronization constraints between different task tree nodes is realized by assigning states to the individual nodes, like disabled, enabled, active or completed.

![Sample Task Tree](image)

**Figure 3.12.** Sample Task Tree [Simmons and Apfelbaum, 1998].

The C++ extension TDL has been created with the purpose to facilitate the creation, synchronization and manipulation of task trees. With a special compiler, TDL constructs are translated into C++ code.

To support the design and debugging of task-level control programs, several graphical tools are provided, e.g. the Visual Design Tool (VDT, see Figure 3.13). With the help of VDT a programmer is able to specify a task in a mixed textual/graphical way.

Despite the available support with graphical tools like VDT, the TDL approach is, without denying the impact of this language extension, still a representative method of textual specification of robot behavior and thus rather suitable for experts.
3. State-Of-The-Art

3.2.2. Graphical programming languages (DeVar: RoboGlyph)

An early realization of graphical robot programming approach has been proposed in [Lees and Leifer, 1993] for the service robotic system DeVar (Desktop Vocational Assistant Robot, [Lees et al., 1988]). The motivations for the development of this programming system were similar to the ones discussed in the previous chapter: It was recognized that the difficulty of creating task programs for robots is one of the main stumbling blocks to their widespread use for service tasks. Furthermore, the objective was to provide non-technical personnel the ability for modifying the robot programming on task level.

The programming framework proposed in [Lees and Leifer, 1993], called RoboGlyph, explicitly represents, in opposite to existing data-flow languages for describing the motion of machines, the sequential nature of mechanical tasks. This concretely means that symbols are dragged on a so-called story board and a 3D model of the robot is used for creating new position icons dynamically.

The basic principle of the RoboGlyph approach is in agreement with this work’s objectives. However, the criticism in the RoboGlyph system is that it does not work in an offline situation because it is directly coupled to the real robotic system during the programming phase. Another drawback of this approach is the limitation of its applicability, because pictograms are intuitive for users only to a certain degree of complexity. Decomposing a high-level goal into a series of RoboGlyph storyboard elements is not trivial for non-technical users, because the correct sequence of sub-tasks for the robot is not always reflected in the intuitive human conception of the appropriate storyline dragged onto the drawing board. This means the proposed approach exclusively uses a too low level of abstraction for the programming task.
3.2. Methods for Programming Service Robot Behavior

3.2.3. The MORPHA Style Guide for Icon-Based Programming

The MORPHA style guide for icon-based programming of modern industrial robots proposes a flowchart-like representation of the program structure based on a well-defined set of icons and symbols, based on ISO 15187 [Bischoff et al., 2002]. The style guide has been embedded in the KUKA Icon Editor which first was available in 2000 as commercial product. The objectives of development had been as follows:

- Improved program overview of the program structure via flowchart-like representation.
- Simplified and more intuitive programming without detailed knowledge of a syntax.
- Preclusion of syntax errors, since programs are no longer typed.
- Visualization of parameters and their boundaries.

The proposed application has been evaluated within trials with beginners, service staff, application as well as expert programmers. Besides the positive confirmation of the above listed objective the trials revealed that changes are easier applied based on the flowchart-like representation.

3.2.4. Microsoft Robotic Studio

The Microsoft Robotic Studio is a universal graphical development environment for robot control and simulation. Its main purpose is on the academic and educational level, but also commercial robot manufacturers are using and/or supporting it. Due to unique interfaces there are many robot manufacturers and hardware component vendors supporting Microsoft Robotic Studio, some well-known supporters are:\n
- Lego Mindstorms NXT [Lego, 2008]
- iRobot Create [Wikipedia, 2009m]

3. State-Of-The-Art

- KUKA Robotics Educational Framework [KUKA, 2008]
- Robosoft’s robots [Robosoft, 2008]
- fischertechnik [fischertechnik, 2008]
- Aldebaran Robotics [Aldebaran Robotics, 2009]

Besides Microsoft’s .NET programming languages the programming within the Robotic Studio is also possible via data flow schemes. Namely, the Microsoft Visual Programming Language (MVPL) is provided. The main window of the MVPL-Editor and a sample MVPL diagram for simple bump-turn-go wander behavior is shown in Figure 3.15.

![Sample Microsoft Visual Programming Language (VPL) diagram for simple bump-turn-go wander behavior](image)

**Figure 3.15.** Sample Microsoft Visual Programming Language (VPL) diagram for simple bump-turn-go wander behavior [Microsoft, 2008].

MVPL mainly focuses on data flow oriented than control flow oriented programming, as the latter one is applied in conventional programming. Control flow based programs execute commands sequentially; however, a data flow oriented program may be rather compared to a series of workers on an assembly line, who do their assigned tasks as the materials arrive. Consequently, VPL is specially targeted at the concurrent or even distributed processing scenarios.
As shown in the sample diagram, the basic building blocks in MVPL are so-called activities. These blocks have inputs and outputs and can be connected to other activity blocks. The re-usage principle is realized in MVPL via hierarchical composition of sub-activities. The connections between the blocks are established via pins, where pins on the left side receive incoming messages and pins on the right side send outgoing/output messages.

As soon as an activity block receives a valid incoming message, the block becomes active and processes the message. Since all data arriving at an activity block is consumed by it, the data is replicated by the block on the output pin in case of message forwarding to subsequent blocks. Multiple input and output connection pins are allowed. The specialty of output pins is that they come in two different kinds: Result outputs with rectangular pin shape as well as notification outputs (also referred to as event or publication output) with round pin shape. Result outputs consider messages that are direct responses to incoming messages (data), whereas notification outputs have the purpose of reporting about certain internal states of the activity block which therefore does not have to be polled.

### 3.2.5. Industrial Automation Systems: Programming with IEC 61131-3/61499

Important state-of-the-art approaches for robot programming in the field of industrial automation systems are the standards IEC 61131-3/61499. These standards have been developed by the International Electrotechnical Commission (IEC) for using industrial controllers based on the experiences with existing PLC languages. In contrast to the concepts presented so far they are not closed and specific concepts for a particular application, but for more general use cases and various application fields, but mainly targeted at automation systems. Due to their flexible character and impact in automation, these standards contribute important aspects within the scope of this state-of-the-art review.

In IEC 61131-3 five programming languages were defined, together with a data concept using latest software development methods [John and Tiegelkamp, 2001].

These five language concepts of IEC 61131-3 that shall be summarized in the following are:

- Instruction List (IL)
- Structured Text (ST)
- Function Block Diagram (FBD)
- Ladder Diagram (LD)
- Sequential Function Chart (SFC)

With the help of these languages, the so-called Program Organisation Units (POUs) are implemented. One POU corresponds to a Block in conventional programming systems. POUs are the smallest independent software unit of a user program and come in three different kinds: Function, Function block and Program. Function blocks have, in difference to simple functions, memory, and are therefore able to remember status information (instantiation), whereas functions always return with the same value.

---

4PLC = Programmable Logic Controller
when being called with certain parameters. Programs provide the access to the inputs and outputs of a PLC or other POUs and are thus the main scope of any PLC user program.

**Instruction List (IL)**

The *Instruction List* (IL) is a textual language, very similar to assembler. One instruction line contains one executable command for the PLC. IL is universally usable and it is often employed as a common intermediate language to which the other textual or graphical languages are translated.

The problem of IL with respect to the IEC standard is that the influence and evaluation of an accumulation result (called *Current Result, CR*) is not precisely defined in the standard. For example the CR value and data types after an unconditional jump can be different according to a certain implementation of a programming system.

**Structured Text (ST)**

*Structured Text* (ST) is, like IL, a programming language in textual representation, comparable to PASCAL and C in the PC domain. In opposite to IL, ST is considered to be a high-level language, since it does not use low-level, machine-oriented operators, but contains a large range of abstract statements describing complex functionality in a very compressed manner.

Compared to IL, ST comes along with several advantages on the one hand, which, on the other hand, also introduce problems. The advantages are:

- Compressed formulation of programming task
- Statement blocks lead to clear program construction
- Contains powerful constructs to control the command flow

These advantages are obtained at the expense of potential losses of efficiency, since the translation of high level code to machine code cannot be influenced directly by the programmer.

Like other high level programming language, ST contains *statements* (e.g. assignments, selections and iterations), *expressions* (partial statements that produce a value necessary for the processing of statements) and *functions*.

**Function Block Diagram (FBD)**

The programming language *Function Block Diagram* (FBD) originates from the signal processing field, but now has become more and more important as universally usable language for programming of industrial controllers. FBDs provide, like textual programming languages, also a well defined starting and
3.2. Methods for Programming Service Robot Behavior

ending point of a POU, and also a variable declaration part. The specialty of FBDs is the representation of the code part with the help of networks. This enables a good structuring of the POU’s control flow.

The network of a FBD consists of block elements, connections and connection lines. The information exchange between blocks takes place via output parameters of a certain block, which are subsequently sent to the next processing unit. Figure 3.16 depicts one function block element, containing in- and output variable connectors, intern variables as well as an IEC 61131-3-compliant specification of internal behavior. In general, the graphical representation elements of FBDs are very similar to logical gate circuits.

![Figure 3.16. Function block according to IEC 61131-3 (from [Chiron and Kouiss, 2007]).](image)

The conceptual idea of programming with function blocks is used in this work. As first discussed in Section 4.3.2, elementary process-structure programming is decisively simplified with the help of a function-block-based approach.

### Ladder Diagram (LD)

The Ladder Diagram (LD) originates from electromechanical relay systems and is able to describe the flow through the network of a POU from the left to the right side. Basically, this language is capable of processing Boolean signals only. Similar to the FBDs, LDs makes use of network structures for flow specification. An LD consists of connections, contacts and coils, elements for execution control (e.g. jumps of function calls) as well as connectors. The elements can be arranged sequentially or in parallel.

### Sequential Function Chart (SFC)

The graphical programming language Sequential Function Chart (SFC) is very similar to Petri nets. It works time- and event-driven and consists of

- **Steps with associated actions**, depicted by rectangular boxes. Each steps represents a certain state of the system being controlled. At least one step is marked as initial step by vertical bars. The action is performed while the step is active. Actions can be described by one of the five IEC 61131-3 languages.

- **Transitions with associated logic condition**, depicted as directed line between steps.
3. State-Of-The-Art

A step in an SFC diagrams is either active or inactive. A connecting transition is passed when all steps above it are active and the condition associated with the transition is true\textsuperscript{5}. In this case, all the connected active steps are deactivated at once and the steps below the transition are activated at once. Like Petri nets, SFC diagrams are able to model parallel control flows.

Similar to Petri nets, SFC diagrams are verified for reachability of states, deadlock situations and safe networks. One of the open issues in the ongoing work of the IEC committee is the fact that currently no foolproof algorithm is available that forbids unsafe or unreachable networks. Furthermore, it is possible to define safe and completely reachable networks that are forbidden by the IEC verification algorithm.

3.2.6. Design of IEC 61131-3 Function Blocks using SysML

In [Chiron and Kouiss, 2007] it is discussed how to use SysML to represent IEC 61131-3 Function Blocks. The main focus of this contribution is to evaluate the modeling abilities of the rather recently released SysML [OMG, 2006] according to PLC-specific programming rules. Unlike many other papers, which dealt with the usage of the UML port element ([Heverhagen et al., 2003], [Thramboulidis, 2004]) it is discussed here how the SysML flow port introduces a standard way to express the required data interfaces.

Flow ports, as one important extension of SysML to basic UML 2.0, specify the input and output items that may flow between a block and its environment. A block is another extension and is a new stereotype of an UML-class that in particular represents a subpart of a system or functions that belong to a process like the IEC 61131-3 Function Blocks. It is possible that multiple items flow between blocks, in this case the port is called non-atomic and a flow specification lists all the flowing items. In contrast, atomic flow ports allow only one item to flow and thus they are in accordance with the IEC 61131-3 Function Block.

3.2.7. CORFU - Function Block Development Kit (FBDK)

\textit{`New generation, function-block-oriented Engineering Support Systems (ESSs), are highly required to support the whole life cycle of IPMCS (Industrial Process Measurement and Control Systems) applications.'}

The CORFU\textsuperscript{6} Function Block Development Kit (CORFU-FBDK) [Holobloc, Inc., 2008] is used in conjunction with an existing CASE tool, namely Rational’s Rose [Tranoris and Thramboulidis, 2003]. The final goal is to offer a unified design methodology in the field of IPMCS that is in accordance with most recent standards such as IEC 61499.

The proposed design process starts as follows within the UML-based CASE tool environment:

- Requirements capturing: Define use cases of the system, i.e. the responses of the system to external events that originate either from devices or humans.

\textsuperscript{5}In this case the transition is also called superable.

\textsuperscript{6}Common Object-oriented Real-time Framework for Unified development of distributed applications
3.2. Methods for Programming Service Robot Behavior

- **Behavior capturing**: Examination of the system’s dynamic behavior, e.g. with object interaction diagrams (UML sequence diagram).

- **Static view capturing**: Design of the static view of the system in terms of class diagrams.

- **Iteration**: Iterate over the last steps to achieve complete consistency between them.

At this point the process switches from the UML view to a function-block-oriented view that is better understood by control engineers. The switching is supported by a set of transformation rules. Eventually, the CORFU-FBDK allows processing the steps of

- **Refinement and evaluation**

- **Model verification**

- **Function block distribution**.

For the verification of the FB network the authors in [Tranoris and Thramboulidis, 2003] claim to still examine the possibility to export the FB network diagram in IEC61499 compliant XML specifications that can be imported by available tools for testing and validation, such as VEDA\(^7\).

### 3.2.8. Universal Realtime Behavior Interface - URBI

URBI, developed by the young French start-up company Gostai [Gostai, 2009] (founded in 2006), is one of the most recent upcoming software frameworks for robotic research. Its characteristics are that it is a parallel and event driven framework and mainly aims at

- **Orchestrating components and behaviors**

- **Providing a client/server control architecture**

URBI is fully object-oriented, provides several interfaces (e.g. C++, Matlab), a scripting language (Urbiscript), the URBI Studio IDE and URBI live, which is a simulation environment, where e.g. finite state machines can be used. Currently, URBI claims to be compatible with 15 robot systems, including prominent platforms like Aibo, Webots, Khepera, etc.

URBI is a very recent development in the field of robotics software platforms and provides several interesting features, while others are yet not included. Positive aspects with respect to this work are the capability of behavioral simulation with e.g. state machines and the behavior- and parallel-executing approach within a distributed system. However, the issues of intuitive graphical programming, system verification as well as model-driven development are obviously open problems.

\(^7\)VEDA = Verification Environment for Distributed Applications [Vyatkin and Hanisch, 2001]
3. State-Of-The-Art

3.2.9. Summary

The representative contributions in the field of robot behavior programming discussed in this section clearly point out the main direction of development and research. Visual programming languages offer the impact of intuitive and ergonomic specification of robot behavior, together with or based on such advantages as improved program overview, hierarchical diagram structures, hiding of details, etc. Most of the existing visual programming approaches are based on a non-standardized or weakly standardized form of program visualization as they make use of a particular set of building blocks and construction rules and rather loosely refer to be based on e.g. data flow or control flow programming paradigms. The prevalence of proprietary solutions is most likely to be explained with the fact that the primary motivations of the visual programming approach - the simplification of the programming process - shall be exhausted to the maximum extend. Thus, the basic building blocks and construction rules are adapted to certain requirements from the developer’s viewpoint.

On the other hand, the standards IEC 61131-3 and IEC 61499 intend to unify the programming on visual level in the general field of automation systems. To satisfy all requirements in this large application field, the standards have to offer a high degree of flexibility. Consequently, it is hard to find a compromise between fully respecting the most recent standards on the one hand and to realize a visual programming approach which is as simple and easy-to-use as possible from the viewpoint of a certain project.

To conclude, the work presented here will be in line with the mainstream in this field and will offer solutions that are as much as possibly adapted to the underlying process-structure based framework. For the sake of prior goals of this work, i.e. to offer a user-friendly and intuitive programming process, the existing standards cannot be taken into account to a large extend. However, for the purpose of the evaluation of the effectiveness of the proposed methods, comparisons to the standards will be drawn. Definitely, the formal standards (like IEC) as well as informal standards (programming with flowcharts, data flow or control flow - oriented programming, etc.) are taken into account for the development of the new methods to attain a maximum degree of conformity.
Requirements Analysis for the Process Model "FRIEND::Process"

“A requirements defect that is left undetected until construction or maintenance will cost 50 to 200 times as much to fix as it would have cost to fix at requirements time.” [McConnell, 1996]

In accordance to the discussions in the last chapter, [Ahern et al., 2003] lists the following motivations for advanced process models:

- Eliminating inconsistencies
- Reducing duplication
- Increasing clarity and understanding
- Providing common terminology
- Providing consistent style
- Establishing uniform construction rules
- Maintaining common components
- Being sensitive to the implications for legacy efforts

Within the scope of this work, the FRIEND::Process - a Process Model for the Development of Semi-Autonomous Service Robots - is elaborated. It will take the listed motivations into account. The effectiveness of the FRIEND::Process and its level of maturity will be evaluated in Chapter 9.
4. Requirements Analysis for the Process Model “FRIEND::Process”

per application for practical realization of the AMaRob scenarios. In the following section, the core elements a process model has to deal with are identified, and subsequently, the driving forces of the FRIEND::Process are identified.

4.1. The Driving Forces: Scenario, Model and Tool Driven Process

A process model for the development of software systems gives a structured specification of the following elements (the abbreviations of process elements as used from now on are given in brackets):

- **Process steps** of development (S)
- **Tools**, tool features, specification methods involved (T)
- **Development artifacts**, in-/output artifacts (A) or products of steps
- **Repositories** (R), to store the artifacts
- **Flow** of development actions and interconnection of steps/tools/products and repositories.

Each section of the FRIEND::Process description is structured according these items: Steps, tools and products, including the related action flow.

A general overview of the FRIEND::Process is given in Figure 4.1. It is shown that the process model structures the development into four main development steps (S), which contain eight single development phases in total. As already mentioned before, a decisive characteristic that also holds for the generic process model HARMONY (see Section 3.1.3) is the fact that the entire process is supported by a model repository (R1). Thus, the FRIEND::Process belongs to the category of fully model-driven processes. However, as the FRIEND::Process is specialized to the development of service robotic behavior, it furthermore interacts with a task knowledge database (R2) and a process-structure repository (R3). Finally, a skill test data repository (R4) is used, to organize a well defined test of skills, which are core building blocks to realize service robotic behavior, as to be explained in more detail later. The interaction with the four repository types requires a certain set of tools, which have been chosen throughout this work as follows:

The software model repository is accessed with the CASE-tool Telelogic Rhapsody® in C++, using UML [OMG, 2008] as modeling language. The other repositories are managed with the help of specially developed and interacting tools that guide through the development process in a user-friendly way.

Altogether, the FRIEND::Process is characterized as follows:

**Definition 4** The process model FRIEND::Process is a special process model for the development of semi-autonomous service robots and it is **scenario + model + tool driven**.

In the following, the purposes of the FRIEND::Process’ four main development steps will be presented. Alongside this summarizing description, the main requirements for each step will be derived, building the basis for the detailed elaboration of process steps in the subsequent chapters. For a compact overview
4.1. The Driving Forces: Scenario, Model and Tool Driven Process

Figure 4.1. Scheme of process model "FRIEND::Process". For a compact overview of the complete process it is referred to Appendix A on page 205 or to the online version of the summary in the IAT Wiki: [Prenzel, 2009].
4. Requirements Analysis for the Process Model “FRIEND::Process”

of the complete process as well as of all involved steps, tools and artifacts, it is referred to Appendix A on page 205 or to the online version of the summary in the IAT Wiki: [Prenzel, 2009].

4.2. Requirements for Scenario Analysis

Figure 4.2. Process model steps of scenario analysis.

According to Process Requirement 2 (page 35) the scenario driven development process starts with a Scenario Analysis. All artifacts of the scenario analysis are specified in UML syntax and are stored in the model-repository. This is done with the objective to gather as many specification artifacts as possible in one model. This makes it easier to browse back to these initial specifications during subsequent development steps. Concerning the sub-scenario specification, a linking of later evolving model elements to the initially determined sub-scenario use cases takes place, so that they serve as navigation entry point at top level of the model.

Figure 4.2 shows the development Steps 1a and 1b, belonging to the scenario analysis. First, the scenario main specification determines the so-called Task Participating Objects, including the system and its hardware components to use, the required smart components and the target application environment.

Environmental items are either objects to be manipulated or obstacles. Items of the first category are in the focus of a manipulative task. Candidates are a book, a bottle, a glass, a meal, a switch, a handle or similar objects which can be handled by a robot. Obstacles have to be taken into account during manipulator motion planning with respect to collision avoidance. In general, to perform actions that do not harm the user, the system and also not the surrounding elements, the system has to be aware of all obstacles within the workspace of the manipulator. The recognition of obstacles and the maintenance of this information within the system’s internal world representation lie in the responsibility of a global safety surveillance system, which is not discussed within this work. Here, the focus is on all information that is directly related to the task to be pursued.

The system items can be part of the robotic system itself, like a manipulator, gripper, camera system, and so on. Or they are smart components with own sensors and/or actuators. These smart components are special devices mounted at the robotic system, e.g. a sensitive surface on a wheelchair tray, or they are distributed in the environment the robot is acting in. Distributed smart elements are for example remote controllable door openers, remote controllable devices in general, or sensor systems, like again sensitive surfaces, objects that are equipped with RFID\(^1\) tags or small camera systems, distance sensors, etc. A schematic of such an environment with FRIEND III acting in it is shown in Figure 4.3.

\(^1\)RFID = radio frequency identification
4.3. Requirements for Task Knowledge Specification and Verification

The sub-scenario specification in development Step 1b appears to be, compared to all other steps, a very simple procedure. Nevertheless it cannot be omitted. Within this step, each scenario is divided into a set of manageable sub-scenarios. The criteria for dividing are:

- Involved objects
- Their physical location
- Re-usability of sub-scenarios
- Manageable complexity of sub-scenarios

Especially the two latter criteria are hard to respect correctly in the very first analysis approach. Therefore, a rough consideration is accepted here and the criteria will be re-evaluated during subsequent development steps.

**Process Requirement 5** The scenario analysis is an informal step, leading to a first structured view on a scenario. It has to collect task participating objects (system components, smart distributed components, environmental items) as well as sub-scenario use cases.

4.3. Requirements for Task Knowledge Specification and Verification

As illustrated in Figure 4.4, the development Steps 2a and 2b, subsumed as Task Knowledge Specification and Verification, deal with process-structures on the abstract and elementary level, abbreviated with $PS_A$ (process-structure, abstract) as well as $PS_E$ (process-structure, elementary). The Process Requirement 3
4. Requirements Analysis for the Process Model “FRIEND::Process”

Figure 4.4. Process model steps of task knowledge specification and verification.

(page 35), i.e. the fact that process-structures play a central role in the complete development process, takes effect at this point.

4.3.1. Requirements for Abstract Process-Structure Programming

The programming of a $PS_A$ requires as input the informally specified task participating objects as well as one of the beforehand defined sub-scenario use cases. The output of this development step are formal descriptions of the robotic system components that are involved into task execution as well as an abstract process-structure which entirely defines the necessary actions on the abstract level to perform a certain desired task within the given sub-scenario.

Task Representing Elements

The elements a $PS_A$ consists of have firstly been introduced in Section 2.4.1 (page 16). Figure 4.5 summarizes their formal specification and their relationships with the help of a class diagram. This diagram is an abstract representation of all involved entities and will not correspond directly to the actual implementation of these data structures. This abstract form of representation has been chosen in order to provide a more compact description of the elements and to maintain a consistency with the naming conventions that are used in this text here. In this diagram and all following ones that describe the definition of development artifacts, the same color scheme is used: Gray entities are input elements from a previous development stage, the elements that are produced in the current step are marked in blue, and white elements are helper elements, which are involved in different steps or serve as grouping elements only.

Figure 4.5 shows that a $PS_A$, as already discussed, consists of Object Constellations (OC) that denote one or more task participating Items being in certain physical contact states. These are either SysItems belonging to the system or to the set of distributed smart components or are environmental items (EnvItems). A composed operator (COP) describes the transition of object constellations and the effect of an operator is specified with pre- and post Facts that are assigned to the affected object constellations. A complete description of a valid state within a $PS_A$ is specified with the help of a Situation. Each $PS_A$ has a default initial situation as well as a default target situation. Additional elements that are necessary, are symbolically represented SubSymbolicData items (see again Section 2.4.1) as well as
4.3. Requirements for Task Knowledge Specification and Verification

**SymbolicParameters** as parameter placeholders of COPs and Facts for the type-conform parameter replacement\(^2\) with item instances according to the parameter's and item's Characteristics.

Additionally to the specification of a PSA according to the known principle as described in Section 2.4.1, an activity diagram is composed for each PSA that describes a sample operation sequence within the sub-scenario that is modeled by the process-structure (more extensive discussion of the PSA activity diagram is done in Section 6.1). This activity diagram is linked to the sub-scenario use case that has been defined in the previous development step.

As visible in Figure 4.5 there is strong interconnection through relationships between the task-representing elements. Due to this fact, the instantiations of these data structures are stored in a relational database. To be able to manage this database of task representing element prototypes in a user-friendly way, a special programming interface is required. This user interface has to guide through the specification process and has to guarantee a logically consistent maintenance of the database's contents.

**Process Requirement 6** A user-friendly programming interface for logically consistent management of task representing element is required (TRE Database Management).

More specific, a requirement for this tool can be formulated:

\(^2\)Type conformal replacement of parameters requires that characteristics of an item to be inserted are a superset of the parameter's characteristics. This principle is formally introduced in [Martens, 2003]
Tool Requirement 1 A user-friendly programming interface for logically consistent management of the database containing task representing element prototypes of the abstract programming level (Items, COPs, Facts) is required.

Programming of Abstract Process-Structures

So far, as introduced in Section 2.4.1, the programming of abstract process-structures took place in a semi-graphical way (see example in Figure 2.9 on page 17). Object constellations are displayed with the help of rectangular boxes containing the abbreviation of the involved items. The boxes are connected with arcs, representing the COPs. To get insight into the fact status of a certain object constellation, a separate dialog had to be opened in the graphical programming interface, containing the node’s facts and their states. This approach is not user-friendly due to the following facts:

- The physical embodiment of the items of a certain object constellation has to take place in the user’s imagination.
- Different versions of an object constellation with the same items but with different object states (e.g. different filling states of objects or the location of an object that is moved on a table, etc.) are denoted with different instance numbers. The real interpretation of the concrete physical state can only be accessed by reading through the node’s facts and their values. It is not possible to access the physical state of an object constellation with one glance.

This specification interface has to be improved to achieve a user-friendly way of abstract process-structure programming, especially with respect to usability of the programming interface for non-expert persons, e.g. in the case of the FRIEND system the patient’s caregivers or the wheelchair user himself. A form of $PS_A$ representation that is much more user-friendly, is depicted in Figure 4.6. Here the same $PS_A$ as given in the semi-graphical form in Figure 2.9 is shown with a pictographic representation of object constellations.

![Figure 4.6](image.png)  
**Figure 4.6.** Scheme of a pictographic representation of an abstract process-structure. The $PS_A$ describes the scenario *Fetch cup* and the encircled numbers represent the same COPs as given in Table 2.1 on page 17.
This leads to the following requirement for the improvement of abstract process-structure specification:

**Process Requirement 7** A new programming interface for user-friendly and intuitive specification of abstract process-structures has to be developed. This tool will be named PSA-Designer.

**Tool Requirement 2** The PSA programming interface shall represent object constellations in a pictographical manner.

Furthermore, it must be possible to connect the object constellations with COPs. These COPs will induce their pre- and post facts onto the affected object constellations in a logical consistent manner, i.e. no new fact must be in logical conflict with the set of already existing facts of an object constellation. Therefore, a mechanism is required which is able to check for logical consistency in a set of facts. In the same way, the logical correctness of an isolated object constellation itself has to be checked (e.g., floating objects are not allowed). Finally, the programming interface has to offer the possibility to specify default initial and target situations for an abstract process-structure.

**Tool Requirement 3** The PSA programming interface shall offer the possibility to connect object constellations via composed operators (COPs) as well as to specify a default initial and default target situation for an abstract process-structure.

**Tool Requirement 4** A mechanism is required, which is able to check the logical consistency of a set of facts in object constellations as they are induced by the object constellation itself or via pre- and post facts of composed operators.

### 4.3.2. Requirements for Elementary Process-Structure Programming

The programming of an elementary process-structure is also considered as programming on system level. From the viewpoint of behavioral system specification, the elementary process-structures are the direct interface to system operations that are executed on the robotic system. Elementary executable operations (EEOP) access the sensors and actuators of the system, and therefore for example, a $PSE$ contains a strategy for prevention of resource conflicts, as discussed in Section 2.4.2.

**Task Representing Elements**

Due to this boundedness to a certain system configuration, the programming of a $PSE$ takes the formal definition of SysItems, i.e. items that are assigned to a certain system, from the preceding Step 2a as input, together with one of the COPs from $PS_A$ level, its parameter items as well as its pre- and post-facts. This is again formally summarized by the gray marked elements in Figure 4.7.

As introduced in Section 2.4, a COP describes an operator from the level of abstract process-structures and it is decomposed into a special Petri net on the level of elementary process-structures. To facilitate the programming of $PSE$s to a large extend and to achieve a more user-friendly and less error-prone specification process, a new programming level is proposed in this work here, namely the programming based on function block networks (FBN).
4. Requirements Analysis for the Process Model “FRIEND::Process”

As specified by the diagram in Figure 4.7, an FBN is related to one COP. Furthermore, it aggregates system facts (SysFacts) and a set of EEOPs. Each EEOP in turn has parameters (SubSymbolicParameters), a set of possible ReturnValues, as well as resource items, which represent the resources that are required for the execution of a certain EEOP. The effect of all these task representing elements from this elementary level and also from the abstract level on the overall development procedure within the FRIEND::Process will be discussed in full detail later. The presentation of their summarized definition with the help of a class diagram is done at this point for the reason of better understanding of their relationship and the overall complexity of the process.

Now, the Tool Requirement 1 has to be refined:

**Tool Requirement 5** The tool for the management of the database containing the task representing elements as stated in the Tool Requirement 1 has to include the management capabilities of EEOPs from the elementary programming level.

**Programming of Elementary Process-Structures**

In the past, it turned out that $PS_E$ specification via Petri nets is a costly process in terms of time and error-correction. In practice, several iterative specification-verification cycles are necessary to obtain an error-free Petri net that is ready to be used as input for task planning and execution. The function block based approach is proposed with respect to the following objectives:
4.3. Requirements for Task Knowledge Specification and Verification

**Process Requirement 8** The function block networks shall hide the complexity of a PSE. The tool to be developed will be named **PSE-Designer**.

**Tool Requirement 6** The FBN-based programming interface shall offer integrated intelligent programming support: Include intelligent automation of the programming process and inherently avoid semantic mistakes.

**Tool Requirement 7** The FBN-based programming interface shall integrate the verification routines.

**Tool Requirement 8** The FBN-based programming interface shall achieve a fully integrated programming approach tailored to the requirements of the complete software framework.

In contrast to other function block programming approaches like [Tranoris and Thramboulidis, 2003, Chiron and Kouiss, 2007], the approach to be pursued in this work will only implement a part of the function block specifications as defined by the IEC 61131-3/61499 standards discussed in Section 3.2.5 (page 42). The IEC function block specification also has to satisfy further objectives, i.e. to model internal variables as well as behavior. The function blocks as considered here, merely represent the execution of a single EEOP along with some additional information related to EEOP execution, as detailed later on.

Figure 4.8 depicts a sample function block network structure that represents the same PSE **GraspObjectInContainer** modeled with a Petri net in Figure 2.10 on page 20. To get a clear understanding of the improving impact of function block networks in the specification process, it has to be pointed out that the Petri net in the cited figure is manually drawn to improve the quality of illustration. The Petri nets as they are set up in the former approach with the Petri net tool **HPSim** [Anschuetz, 2009] are already hardly readable in the case of nets with low complexity. See for example Figure E.1 in the Appendix E.

![Figure 4.8. PSE GraspObjectInContainer as function block network.](image)

With the exception of the **Start Block**, each block has so-called ports on the left as well as on the right hand side. On the left side of the block, the inputs are represented by **required** ports, whereas the outputs are modeled as **provided** ports on the right side of the block. The term ports has been taken from the UML 2.0 specification. The authors in [Chiron and Kouiss, 2007] show how to apply the UML port specification to create function block networks fully compliant to IEC 61131-3.
**Tool Requirement 9** The function blocks shall be equipped with provided and required ports, following the notation as introduced in the UML 2.0 specification.

In this approach, some special agreements are considered with respect to the requirements for \( PS_E \) programming: Like the other Control Blocks, the Start Block is unique within each network. A Start Block defines the entry point for the process-structure. Two links leave the Start Block in the example given in Figure 4.8. This implies that a parallel execution of EEOPs may be initiated right at the beginning, depending on the subsequent elements. Here, the EEOP Block AcquireObjectInContainerBySCam follows, which represents the execution of a machine vision skill that acquires an object’s location and size via stereo camera. In parallel, depending on the state of the Fact Block GripperIsOpen(Manipulator), the EEOP OpenGripper may be executed at the same time. If links between blocks shall be connected via OR or AND conditions, the appropriate Logical Blocks are used. Fact Blocks do not contain more internals as the predicate logic fact’s name and its parameter(s). A Fact Block always has two required and two provided ports, the former to set and the latter to read the fact’s state. However, it is not mandatory to connect all these ports, since fact values are used and set optionally within the network. The specialty of the Target Block in this function block network approach is that it is equipped with a required as well as with a provided port. The reason is that after successful completion of one \( PS_E \), the post-condition facts of the COP are set, i.e. the linking to the \( PS_A \) level is established here.

**Tool Requirement 10** The function block networks shall contain the following kinds of blocks: Control blocks, logical blocks, EEOP blocks and Fact blocks.

**Tool Requirement 11** Parallel EEOP execution shall be modeled by parallel leaving branches at provided ports.

**Tool Requirement 12** The provided port at the target block shall allow only the connection to fact blocks, to set all post-facts of the modeled COP.

To summarize, the ergonomic system-level programming with function block networks shall build the basis for an approach named Automation by Configuration:

**Definition 5**

**Automation by Configuration (AbC):** Instead of programming the robotic system, the development of new system functionality is done via configuration. The configuration interface embeds all necessary specification, programming and verification methods and guides the programmer through the configuration process. Only the really relevant configuration steps according to a certain programming context are shown to the user. The AbC-principle shall be understood as system-controlled and constrained-based programming by configuration.

In the context of \( PS_E \)s, instead of programming on system level, the development of elementary process-structures is done via configuration.
4.4. Requirements for Skill Development

The task execution based on process-structures as well as on the shared-control paradigm as introduced in Chapter 2 (see also essential paradigms in Definition 1 and Definition 2 on page 4) is realized within the software control architecture MASSiVE\textsuperscript{3}, also called FRIEND::Architecture. This control architecture, tailored to the mentioned paradigms, is the enhancement of the work by Martens which is discussed in [Martens, 2003]. These further contributions concern all parts of the FRIEND::Architecture, which is illustrated in Figure 4.9 - namely the Human-Machine-Interface, the Sequencer, the Reactive Layer as well as the World Model.

The control architecture and its enhancements are documented in several publications, like [Martens et al., 2006, Prenzel et al., 2007, Martens et al., 2007], and are summarized in the following as a basis for the derivation of the requirements in this section.

4.4.1. Background: FRIEND::Architecture (MASSiVE) - Infrastructure for Semi-Autonomous Task Execution within Distributed Systems

The Sequencer, as explained in Section 2.4, plans and executes sequences of operations that are considered as elementary executable operations (EEOP) from its perspective. The execution of basic operations takes place either in the Reactive Layer or the Human-Machine-Interface layer of the system. Due to uniform software interfaces, the treatment of operations by the Sequencer is uniform, too.

\textsuperscript{3}Multi-layer Architecture for Semi-Autonomous Service Robots with Verified Task Execution
4. Requirements Analysis for the Process Model “FRIEND::Process”

As depicted in Figure 4.10, the Sequencer consists of two modules that are designed as active objects: The Task Planner and the Task Executor. Active objects are a software design pattern which separates the execution of a method from its calling context via using threads. Thereby, the method’s implementation is independent of any threading details [Gamma et al., 1994]. Thus, planner and executor are able to act independently, which enables the control of skill execution in the reactive layer as well as reactions in the planner like interruption of ongoing task execution or re-planning of operation sequences.

![Figure 4.10. Skill execution from Sequencer.](image)

To be able to execute several operations simultaneously, asynchronous calls of skill-methods are necessary. Furthermore, the operations may run on different processors, e.g. because of system-hardware that is distributed physically as it is the case for remote smart devices. The distribution of skill execution capabilities should be adaptable in a flexible manner, i.e. without changing the system structure or extensive re-implementations, to have the opportunity at hand to scale the computing power available for a single skill. All these demands are fulfilled entirely with the help of standardized and platform-independent communication infrastructures based on CORBA (Common Object Request Broker Architecture, [Henne and Vinoski, 1999]) which is used to build up the skill execution infrastructure.

**Reactive Layer:** The name reactive layer resides from its purpose to provide reactive behavior. This means to directly couple sensorial input with the control of an actuator (i.e. to design a control loop) to establish autonomous behavior that is robust against dynamic environmental changes. As depicted in Figure 4.9 the reactive layer is furthermore responsible for offering monitoring operations (based on input from the sensors) as well as direct control of the actuator (manipulative skills). The latter aspect is important for example when user interaction in the form of direct actuator-control becomes necessary. Due to this, several skill servers provide the necessary basic operations, i.e. skills, of the robotic system by accessing the sensors and actuators of the system or remote smart devices. This means, a skill layer has access to a hardware layer, whereas different hardware servers encapsulate basic hardware functionalities.

Skills have to operate on the already mentioned sub-symbolic environmental information. As shown in Figure 4.9, the Sequencer including the symbolic planning engine accesses the symbolic layer of the world model. Thus, the Sequencer (on the basis of high-level process-structures and symbolic descriptions) is responsible for the correct abstract modeling of that segment of the environment that is relevant to the current task-execution. To administrate all sub-symbolic information in a structured manner, a sub-symbolic world model server is introduced within the reactive layer. Here, sub-symbolic information is stored with reference to symbolic information from the upper layer of the world model and consequently a connection between both layers of the world model (and therefore also between these both information layers) is established.
**Skill Server:** The criterion for separation into several skill servers is derived from the functional entities of the system. That means, one skill server offers all the system operations that have to be assigned basically to one certain entity. In the case of scenarios that take place in an intelligent environment, such as the one depicted in Figure 4.3, the necessary skill-servers are the Manipulator-, Tray-, MachineVision-, Cooking- and SmartFridge-skill-server, with their assigned hardware-servers as depicted in Figure 4.11. Here, for instance, the Cooking-Skill-Server provides skills to access the hardware-server MicrowaveOven and thus to control this remote device or to extract data from it.

![Figure 4.11. Reactive layer for FRIEND with intelligent environment.](image)

From the software-technical point of view, skills are methods of a skill server that are executed asynchronously. This means skill-methods are non-blocking and will return immediately after their activation. The problem of asynchronous execution is that no values or parameters can be returned. Therefore, sub-symbolic data that is generated during skill execution is stored within the world model. The information on the status of skill execution (e.g. successful execution) has to be transmitted via another communication way. For this issue callback objects are introduced, which can be accessed by the skill caller and the skill method itself. Callbacks are also used for sending information from the skill caller to the skill while it is executing. This could be for example the command to stop the skill or to re-parameterize it. Figure 4.10 already showed how the task executor within the Sequencer maps callback objects to asynchronously invoked methods.

### 4.4.2. Problem Analysis with Respect to Skill Development

At this point it is referred back to Process Requirement 4 (page 35): The infrastructure of the FRIEND::Architecture and the first development steps of the FRIEND::Process impose uniformity and many requirements onto the process of developing the “basic” skill functionality of the robotic system. In particular, this becomes manifest in the formal specification of an EEOP, as given by Figure 4.7 (page 56):

- Skill-Server which executes the EEOP/skill
4. Requirements Analysis for the Process Model “FRIEND::Process”

- Parameters
- Return Values
- Resource items

However, if a skill is implemented in a conventional code-centric approach, several problems arise:

- How to assure the correct usage of parameters within the skill implementation as well as the right handling of the data that is extracted from the world model according to these parameters?

- How to assure the correct resource usage as externally specified? It is of central importance for the system’s robust execution performance that the prevention of resource conflicts as verified on $PS_E$ level is not bypassed on the implementation level.

- Are correct return values sent back to the Sequencer layer?

- Is it possible to stop a skill?

It is obvious that a wrong implementation of a skill could foil the impact of preceding task knowledge verification. Therefore, within the skill development stage of the FRIEND::Process, methods will be introduced based on model-driven development techniques, which enforce consistency to task knowledge specifications.

A further category of problems is that a pure code-centric approach easily tends to produce skill implementations that are hard to maintain. Practical code-based development experiences show that skill methods often grow to e.g. 600 code lines or more, and may include 4 to 5 or more nesting levels. Also, a known common development problem is that the proper commenting of implementations relies on a certain programmer’s discipline. Here, the methods of the FRIEND::Process will guide the implementation process and eventually lead to maintainable skills as core building blocks of a system’s overall functionality.

**Process Requirement 9** MDD-based methods have to be introduced within the skill development step of the FRIEND::Process to enforce consistency to the previously verified task knowledge specifications as well as to achieve skill implementations that are maintainable in the long term. A uniform organization/structuring, commenting, modularization and documentation of skills has to be enforced.

Figure 4.12 gives a first overview on how to structure the skill development process within the FRIEND::Process. A fully detailed discussion is dedicated to the elaborations in Chapters 7 and 8. However, the overall approach is pointed out here: Skill development and subsequent testing on skill- and scenario-level are carried out in a tightly interacting and iterative process. From first skill designs on, the testing of a skill skeleton takes place, in accordance with and based on the model-driven development paradigm. Executable models are created in early design steps and are tested and refined subsequently. Also, Figure 4.12 shows that the complete lineup of process driving forces comes into play - i.e. the development artifacts from the task knowledge database, the process-structure repository, test data repository as well as model repository.
4.5. Requirements for Skill and Scenario Testing

The nature of the Skill Development stage in the FRIEND::Process is basically similar to the procedures that take place during Skill Design. Therefore, this step, which completes the skill to the full required functionality, has to be seen as a successive enhancement of the initial model elements with repetitive intermediate testing cycles.

4.5.1. Skill Testing

As shown in Figure 4.12, the skill test retrieves Skill Test Sequences, World Model Test Data and EEOP data as input. In the simplest case, the skill test sequence only consists of the skill that is currently under test. However, if the skill needs other skills to be executed before or after its own execution, the skill test sequence is set up accordingly. As shown, the test sequences are already defined during skill design, together with the determination of the skill’s specification, including skill use cases and skill requirements.

In Section 4.4.1 it has been discussed that skills, due to their asynchronous nature, exchange sub-symbolic information solely via the world model. In the case that other skills are not available at skill testing time to produce the required sub-symbolic input data for the skill under test, the world model test data is used. This has also been set up during skill design, together with the skill test sequences and according to the skill specification.

As to be explained in more detail in Chapter 8, the skill execution and thus also the skill test can take place in different execution levels:

![Figure 4.12. Process model steps for skill development and testing.](image)

Figure 4.12. Process model steps for skill development and testing.
4. Requirements Analysis for the Process Model “FRIEND::Process”

- In probabilistic test mode, a skill returns one value out of its finite set of return values according to a certain probability distribution function that is specified beforehand. This test is used to check the success of a skill call from the skill execution module in general, but also to execute complete or partial scenarios in a pure simulative mode and to observe the behavior of the task planning module.

- The skill simulation mode calls the main execution part of a skill but does not access hardware. In this mode, the motion planning of manipulative skills is in the loop, but is not visualized.

- The motion simulation mode introduces a delay while sending motion planning information to the internal 3D world representation, so that the effect of motion planning is observable.

- In hardware simulation level, the simulation of hardware takes place.

- The skill execution mode conducts a full execution of the skill’s complete functionality.

To conduct and evaluate the skill testing, a software tool is required:

**Tool Requirement 13** For conducting skill tests, a software tool is required with the following characteristics:

- Configure or load a skill test sequence with sets of parameters according to different skill use cases.
- Set the skill execution level.
- If required, load world model test data into the world model.
- Start and stop the skill execution and protocol the test execution.

Output of the skill test is on the one hand the observable skill behavior with respect to the execution level, e.g. analysis of the motion simulation in the 3D model visualization, or the actual effect of the skill, like a manipulator movement. On the other hand, as also shown in Figure 4.12, skill test sequence diagrams are generated in parallel and are stored in the model repository, for documentation purposes of wanted or unwanted skill behavior.

4.5.2. Scenario Testing

Scenario testing is at first sight technically similar to skill testing, as skills are executed in sequences, too. However, in this stage, the task planner in the Sequencer is in the testing loop. Thus, the skills are tested in the scope of complete and complex tasks, as they will be later activated by the user of the robotic system from the human machine interface.

Figure 4.12 shows that the skill test data repository is not used anymore. Instead, process-structures and the task representing elements from the database drive this step.

To benefit from a synergetic effect, the skill execution in both testing stages - skill and scenario testing - will use the same skill execution module. If done so, the integration of new skills into the Sequencer
has to take place only once and the skill test can be considered as a higher level test from the overall system perspective.

**Tool Requirement 14** *During skill test and scenario test the same skill execution module is used to prevent the development and existence of functionally redundant modules.*

### 4.6. Summary

In this chapter a summarized overview of the main requirements for the FRIEND::Process and the involved tools to be developed was given. Background information has been clarified that had not been mentioned so far, but that is essential for the overall understanding and for the derivation of the requirements of the FRIEND::Process. These requirements have been gathered throughout the discussion of the four main development stages of the process and are summarized and structured in Table 4.1 (Process Requirements) and Table 4.2 (Tool Requirements). Refinements of the main requirements listed here will be undertaken throughout the subsequent chapters.

#### General Requirements

| RP1 | An integrated process model is required to manage the complexity of the development process, i.e. to guide and unify the development process and to achieve a tight integration of involved tools and development artifacts. |
| RP3 | The process is decisively influenced by process-structures. A structured and ergonomic development of process-structures has to be enabled by the process. Moreover, process-structures have to be integrated into all steps of the process. |
| RP4 | Uniformity and structured characteristic of skill interfaces and procedures as well as skill layer data structures have to be formalized as basis for the process. |

#### Requirements for Scenario Analysis

| RP2+5 | Scenario analysis is one of the driving forces for the process. The informal scenario analysis leads to a first structured view on a scenario by determining task participating objects and sub-scenario use cases. |

#### Requirement for TRE Database Management

| RP6 | User-friendly programming interface for logically consistent management of task representing element prototypes within a task knowledge database is required. |

#### Requirement for PSA-Designer

| RP7 | A programming interface for user-friendly specification of abstract process-structures with pictographical representation of object constellations is required. |

#### Requirement for PSE-Designer

| RP8 | The function block networks will hide the complexity of a PSE. |

#### Requirement for Skill Development

| RP9 | MDD-based methods shall enforce the consistency of skill development to the previously verified task knowledge specifications and shall enforce uniform organization/structuring, commenting, modularization and documentation of a skill to yield maintainable implementations. |

| Table 4.1. Summarized requirements for the FRIEND::Process (Process Requirements, RP). |

In the subsequent chapters, Chapter 5-8, the FRIEND::Process is presented, including the description of all important development steps, the produced development artifacts, as well as the applied and specially developed tools and methods that are tailored to the requirements of the pursued core development paradigms and that guarantee a seamless overall development procedure. Besides the detailed description of the process model as it is given in the following chapters, a UML model has been developed which describes the FRIEND::Process and which can be found in the IAT model repository (*ProcessModel.rpy*). For a compact overview of the complete process as well as of all involved steps, tools and artifacts,
4. Requirements Analysis for the Process Model “FRIEND::Process”

**Table 4.2.** Summarized requirements for the tools that are developed for and used in the FRIEND::Process (Tool Requirements, RT).

![Table showing requirements for TRE Database Management, PSA-Designer, PSE-Designer, and Skill and Scenario Testing.]

It is referred to Appendix A on page 205. An online version of the summary is included in the IAT Wiki: [Prenzel, 2009].
The FRIEND::Process: Scenario Analysis

According to Process Requirements 2+5, the FRIEND::Process starts with a Scenario Analysis. The first step, the Scenario Specification, is the first informal step with the purpose to obtain a first structured view on a scenario and to identify the Task Participating Objects. In the second step during scenario analysis, the scenario is split up into sub-scenarios and a Sub-Scenario Use Case is assigned to each sub-scenario.

5.1. Step [1a]: Scenario Specification

Figure 5.1. Process Step 1a: Scenario Specification.
5.1.1. Definition of Products

The artifacts that are produced within the scenario specification, as well as their relationships, are depicted in Figure 5.2. Central elements are Items that build the TaskParticipatingObjects (TPO). Items can consist of sub-items, which makes a complete item hierarchy possible. As shown in the diagram, the items can be grouped into three categories: Environmental items, system items and additional smart components.

![Diagram of TaskParticipatingObjects (TPO)](image)

Even though the TPOs are specified in a rather informal way here, their specification at this point of development serves as input for a later formal specification step, namely the programming of task participating objects as parts of the abstract process-structures, as described in development Step 2a in Section 6.1. This relationship via output and input products is also given in the main process overview in Figure 4.1.

5.1.2. Tools and Repositories

Without loss of generality, the tool that is used for scenario specification in practice is Rhapsody. As illustrated in the next section, the specification is carried out in UML. Obviously, any other tool that is capable of ”drawing” UML diagrams, is suitable for this procedure, too. However, as already argued above, the intention is to store as many artifacts as possible in the central model repository, to have them at hand throughout the whole development process for easy interlinking and navigation to earlier...
design steps. As Rhapsody is used for the development of further elements, it is also used for this first simpler design procedure.

5.1.3. Ontology of Task Participating Items

The scenario analysis is a preparatory step for the formal task knowledge specification based on process-structures. As discussed before, the process-structures serve for pre-structuring of a high-level task. The flexibility of this approach is that pre-structuring means to specify task knowledge that is applicable to complete classes of objects that are involved in a certain task. To formalize the specification of object classes in an object-oriented manner, a hierarchical ontology is introduced. The term ontology can be described as follows:

An ontology is a formal, explicit specification of a shared conceptualization.  
[Gruber, 1993]

An ontology provides a shared vocabulary, which can be used to model a domain - that is, the type of objects and/or concepts that exist, and their properties and relations.  
[Arvidsson and Flycht-Eriksson, 2008]

The resulting complete ontology for the support scenarios in AMaRob is depicted in Figure 5.3, pointing out the task participating objects of the three scenarios with the help of different colors.

This strictly object-oriented ontology defines the relationship between the objects and determines compositions and derivations. Thus, it specifies common properties of objects within the hierarchy. This in turn influences further development steps that are based on this hierarchical task knowledge. For example, a manipulative operation is designed to work for certain classes of objects and is then extensible for derived object classes in an evolutionary manner.

5.2. Step [1b]: Sub-Scenario Specification

| Input artifacts: | - |
| Output artifacts: | Sub-scenario use case diagram A1.2 |
| Tools: | Rhapsody T1 |
| Repositories: | UML Model R1 |

In this development step, the main scenario is split up into sub-scenarios, according to the criteria as given in Section 4.2. This means in the kitchen scenario for example, the refrigerator and a microwave oven that are included in the overall scenario, will be handled via individual sub-scenarios, due the change of workspace that has to take place when switching from one to the other container. These sub-scenarios can then be started, stopped, repeated, re-initialized, and re-used independently. Special attention has to be paid to the fact that different sub-scenarios can be combined flexibly to be executed.
5. The FRIEND::Process: Scenario Analysis

Figure 5.3. Hierarchical ontology of task participating objects in AMaRob scenarios.
5.3. Summary

During the main step one of the FRIEND::Process - the Scenario Analysis - the two sub-steps Scenario Specification (Step 1a) as well as Sub-Scenario Specification (Step 1b) take place.

**Step 1a** identifies the task participating objects (TPOs), which include system components, distributed smart components of the environment where the system is operating in, and finally the environmental items to be manipulated. The rather informal gathering of TPOs in this step serves as first structured view on a certain task/scenario\(^1\). Due to the fact that behavior and system operations to be developed subsequently have to be re-usable, they have to operate on the basis of object classes in general. Only in the case of very specialized objects this generalization approach is allowed to be violated. To structure the following development steps, the TPOs are categorized with the help of an object ontology, which contains a hierarchical organization of object classes and defines the relations, i.e. the derivations and compositions, between them. A sample ontology of environmental objects for the three AMaRob scenarios has been defined in this chapter.

**Step 1b** splits up a scenario/task into sub-scenarios respectively sub-tasks. This is visualized with the help of use case diagrams which contain the sub-tasks as use cases. The use case diagrams serve as top level entry point for navigation within all subsequently developed artifacts with respect to a certain

\(^1\)The terms scenario and task are used equivalently.
scenario. The sub-tasks identified within Step 1b separate the main task according to maximum degree of re-usability in other scenarios.

The development artifacts of both development steps are stored in the model repository and are used in the following development steps. A development according to the Steps 1a and 1b as described here fulfills the Process Requirements 2 and 5.
6

The FRIEND::Process: Task Knowledge Specification and Verification

The FRIEND::Process is process-structure-driven. Process-structures are a compact prescription of task relevant knowledge which is verified offline.

This chapter describes, how to formally specify and verify task knowledge within the FRIEND::Process. The essential underlying development paradigm as introduced in Definition 2 (page 5) and reflected in the Process Requirement 3 anticipates that process-structures form the basis of all further developments. In process model Step 2a all required procedures with respect to abstract process-structures are conducted and in Step 2b the elementary level process-structures are handled. As in all the development steps, the products, used tools, methods and repositories as well as sample development artifacts are presented.

Figure 6.1. Process Step 2a: Abstract Process-Structure Specification.

<table>
<thead>
<tr>
<th>Input artifacts:</th>
<th>Sub-scenario use case, Task participating objects</th>
<th>A1.1+2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output artifacts:</td>
<td>Abstract process-structure (PS_A), PS_A activity diagram</td>
<td>A2.1–6</td>
</tr>
<tr>
<td>Tools:</td>
<td>Rhapsody, DBGUI, PSA-Designer</td>
<td>T1–3</td>
</tr>
<tr>
<td>Repositories:</td>
<td>Model Repository, Task Knowledge Database, Process-Structure Repository</td>
<td>R1–3</td>
</tr>
</tbody>
</table>

6.1.1. Definition of Products - Overview

A detailed and formal description of the artifacts and their relationships that are specified within Step 2a has already been given during requirements analysis with the help of the class diagram in Figure 4.5 (page 53). Figure 6.2 gives the overview on the specification process of these artifacts that are required for the composition of an abstract process-structure. It shows that during the initial step the formal specification of task participating objects (TPOs) is required. Together with the composed operators (COPs) and symbolic facts they belong to the task representing elements (TRE) that are necessary to build up abstract process-structures.

The procedures necessary for TRE specification are discussed in the following in detail to refine the requirements for the development of a programming interface for the consistent management of TREs according to Process Requirement 6 and its specialization with Tool Requirements 1 and 5. The development of this tools is subject of Section 6.1.5.


Specification of Items

Figure 6.3 shows the activity diagram that contains the decomposition of TPO specification. Analogous to the informal specification in Step 1a (Section 5.1) it is distinguished between system items, smart component items and environmental items. Additionally, as introduced in Section 2.4.1, symbolically represented sub-symbolic items are required for PS_A specification. As shown, the TPO specification first requires the selection of one robotic system. Subsequently, a set of smart component items, environmental items and sub-symbolic items has to be chosen. All these elements can either already exist in the database or have to be created there. If the latter is the case, the actions as given by the diagrams in the Figures 6.4 and 6.5 are required. The first figure illustrates the system specification procedure, which may include the second procedure of item specification, if a certain system item is not yet available in the database.

After selection of the name for a new system, the system’s components are added, either first in the database if not yet existing, or directly as sub-component of the system. In the next step system facts have to be specified, which formulate first order predicate logic statements with the chosen set of sub-components of the system as parameters. An example for such a system fact is IsOpen( Gripper ).

Figure 6.2. Activity diagram that defines the procedure of abstract process-structure specification.
Figure 6.3. Activity diagram of the specification procedure of task participating objects.

Figure 6.4. Activity diagram of system specification procedure. The respective graphical user interface (GUI) front-end is discussed below and is given in Figure 6.18 on page 93.
describing the opening state of the manipulator’s gripper. Further examples are given along with the discussion of the respective GUI design that realizes this specification procedure as detailed in Section 6.1.5. The procedure of fact specification is analogous to EEOP specification as explained below. After the new system has been introduced, it can be directly used within the first step of the TPO specification procedure as described in Figure 6.3. The system items that are involved in the current task (usually only actuators like the manipulator) are included into the set of TPOs then.

The description of the required procedure for introducing new items (system items, smart component items, environmental items or sub-symbolic items) is given in Figure 6.5. First, name, abbreviation and type have to be selected for a new item. This is for example Manipulator as name, MP as abbreviation and SYS as enumeration literal for system item. The complete set of literals for item types is as follows:

**Definition 6** \( \text{ItemType} = \{ \text{SYS}, \text{ENV}, \text{SSD} \} \) with

- SYS: System item (includes smart components)
- ENV: Environmental item
- SSD: Sub-symbolic data item
Next, a set of characteristic strings is assigned to each item. This set must not be empty, since item characteristics are used for type conformal parameter replacement\(^1\) throughout subsequent specification procedures. However, if a valid item type has been specified before, the first characteristic is automatically derived from it:

**Definition 7** The first item characteristic is derived from the item type:

- **SYS:** \textit{IsSysObject}
- **ENV:** \textit{IsEnvObject}
- **SSD:** \textit{SS\_DATA}

For each system item it has to be determined whether the item represents a sensor or an actuator by assigning the characteristic \textit{IsSensor} or \textit{IsActor}. Further characteristics are encoded in the extensible ontology of task participating items as introduced in Section 5.1.3. The characteristic strings are set according to the specified relationships, i.e. \textit{Is+ClassName} for inheritance and \textit{Has+ClassName} for association. The final category of item characteristics describe certain properties of a certain object class, like e.g. \textit{IsGripable}. Altogether, besides the item type related automatically assigned first characteristic, the further characteristics are for example:

- Characteristic of system items: \textit{IsSensor} or \textit{IsActor}
- Characteristics based on associations: \textit{HasDoor}, \textit{HasHandle}, \textit{HasSpoon}, etc.

The last step to introduce a new item is the assignment of sub-component items according to the hierarchical ontology as introduced in Section 5.1.3. To reduce the complexity of the diagram, the section of ontology that is given there is constrained to the environmental items of the three AMaRob scenarios. However, also system items can be composed of sub-components. To keep the example of the Manipulator item, its sub-components - in accordance with the informal specification in Step 1a (see application in Chapter 9, Figure 9.1, page 170) - are: \textit{Gripper}, \textit{Robotarm} and \textit{ForceTorqueSensor}.

Finally, the completely specified new item is written to the database of task representing elements.

**Specification of COPs**

Figure 6.6 describes the process of formal introduction of a composed operator in the database. After definition of the COP name (e.g. \textit{GraspObjectInContainer}), the COP type is specified.

\(^1\)See also Section 4.3.1
According to the discussions in Section 2.4 it is known that three different COP types are allowed. Additionally, there are monitoring operations, which are not explicit part of abstract process-structures and which are discussed in [Prenzel, 2005].

**Definition 8** \( \text{COPType} = \{ \text{ASS, DISS, IST, MON} \} \) with

- **ASS**: Assembly operation
- **DIS**: Disassembly operation
- **IST**: Internal state transition
- **MON**: Monitoring operation

Further on, a COP prototype needs symbolic parameters as placeholders for the TPO items of an abstract process-structure. The mentioned principle of type conformal parameter replacement is applied for the insertion of TPO items into a COP prototype. The underlying rule is that the set of characteristics that are required via a certain placeholder has to be a non-empty subset of the characteristics that are offered by the TPO item to be inserted. Parameter placeholders will also be referred to as formal parameter. This term is used in the IEC standards (see Section 3.2.5 and [John and Tiegelkamp, 2001]).
Figure 6.7. Activity diagram of fact specification procedure. The respective graphical user interface (GUI) front-end is discussed below and shown in Figure 6.13 on page 88.

Definition 9

Type conformal parameter replacement (TCPR), summarized from [Martens, 2003]: The rule must hold that the characteristics of a prototype’s symbolic parameter (also called placeholder or formal parameter) are a non-empty subset of the set of characteristics of a TPO to be inserted at the placeholder.

As depicted in Figure 6.6 a symbolic parameter can either be selected directly or first has to be introduced in the TRE database. Subsequently, the set of characteristics is assigned.

Next step of COP specification is the assignment of pre- and post facts. A fact that has not yet been introduced in the database has to be added first, as described below. The parameters of the facts have to be selected from the set of COP symbolic parameters. Other parameters are not allowed. Further on, each pre- and post-fact needs a boolean value assigned. After complete COP specification the prototype is ready to be written to the database.

Specification of Facts

The specification of a fact prototype is partially similar to the definition procedure of a COP. As shown in the activity diagram in Figure 6.7, the differences are that different environmental modeling levels are possible for facts and, obviously, that the assignment of pre- and post-facts does not take place. The different fact types and modeling levels are as follows:
Definition 10  
**FactType** = \{FAC, SYS, TMP\} with

- **FAC**: Environmental facts to describe states and relationships of environmental objects, e.g. 
  FAC.AON.IsGripped( Manipulator, Bottle )
- **SYS**: System facts to describe the state of system components or system-internal states, e.g. 
  SYS.PN.IsOpen( Gripper ) or SYS.PN.IsAvailable( Resource )
- **TMP**: Temporary facts, which are e.g. used within Petri nets to mark data availability, e.g. 
  TMP.PN.DataAvailable( Data )

Definition 11  
**FactLevel** = \{AON, PN\} with

- **AON**: And-Or net level
- **PN**: Petri net level


Referring back to Figure 6.2 it can be seen that after specification of task participating objects three main categories of actions are required to set up an abstract process-structure:

- Add object constellations (OCs)
- Add composed operators (COPs)
- Delete or edit elements

As suggested above and manifested in Process Requirement 7 and Tool Requirement 2 a pictographic programming of OCs shall be provided by the programming interface to overcome the drawbacks as discussed in Section 2.5. The developments with respect to this requirement are subject of Section 6.1.6. The introduction of OCs and COPs in abstract process-structures from the viewpoint of product flow in the overall programming process is given in summarized form in the following.

#### Add Object Constellations

Adding a new object constellation requires to add a new node to the underlying AND/OR net which is owning this object constellation and representing the physical contact state of the involved objects. As Figure 6.8 points out the object constellation can subsequently be edited and finally a consistency check is applied to accept the object constellation as valid or to request further editing process by the programmer.
The simplest version of editing an object constellation takes place in a non-pictographic way. This means objects from the set of previously selected task participating objects are added to an object constellation or are removed. This approach has also been pursued so far and a $PS_A$ as given in Figure 2.9 on page 17 may result - with the mentioned drawbacks.

The TPOs that can be selected for an OC have to be physical objects; the other kind of symbolically represented sub-symbolic objects are only required during the application of COPs as described subsequently. A unique descriptor is generated for each OC according to the selected objects. This descriptor consists of the alphanumerically ordered instance strings of the contained objects as well as of an OC instance number - e.g. $Bo.1-MP.1-Tr.1_0$ for a manipulator (MP.1) that grasps a bottle (Bo.1) on a tray (Tr.1).

The final consistency check that is applied guarantees that only physically meaningful OCs are allowed. Since objects are not allowed to float, there is for example a platform required where the objects are placed on, a container to contain the objects or an actor with a gripper or a human being that is able to hold objects. The check for floating objects is done on the basis of the characteristics as initially assigned to the task participating objects. The following characteristics are used during the consistency check:

- IsPlatform
- IsContainer
- IsActor
- IsHuman

Add Composed Operators

The activity diagram in Figure 6.9 illustrates the COP adding procedure. This requires first the selection of start and target nodes. Subsequently the assembly rules can be applied. For AND-arcs this means

That the start node objects have to be the exact composition of the objects that are contained in the target nodes. IST arcs can only connect two nodes with the same objects.

**Definition 12**

**COP Assembly Rules:** AND arc start node objects have to be the non-overlapping composition of the objects that are contained in the target nodes. IST arcs connect two nodes with the same objects.

If the assembly rules are not violated, the programmer has to choose one or two operations for the current arc. In the case of AND arcs an assembly and/or disassembly operation can be chosen. IST arcs have a forward operation and optionally a reverse operation. Again, as during the previous programming steps, COP prototypes that do not exist have to introduced in the database according to Figure 6.6.

Some restrictions with respect to COP applicability do hold according to [Martens, 2003]. To simplify the COP insertion process, only applicable COPs are offered for selection, instead of rejecting a certain COP selection afterwards.

**Definition 13**

**COP Conformity,** summarized from [Martens, 2003]: The applicability of a COP is determined through the applicability of involved node objects at the COP’s symbolic parameters (placeholders) according to the type-conformal parameter replacement (TCPR, Definition 9 on page 80) principle. Besides the node objects, only non-physical items (symbolically represented sub-symbolic items) are allowed as COP parameters. Also, the COP type according to Definition 8 determines the applicability of a COP.

Each COP inserts its pre- and post-condition facts into object constellations in the participating nodes. The set of facts in each node has to be checked for logical consistency. However, also the propagation
Figure 6.10. Process of situation assignment in abstract process-structure.

of the facts through the complete network has to be checked for logical consistency. The mechanisms that are used here are defined in [Martens, 2003] and are summarized in the following.

**Definition 14**

**Consistency of Fact Insertion**, summarized from [Martens, 2003]:

- The insertion of a COP’s pre- and post facts in the network parts that are to be connected is checked for logical consistency with the help of an **Inference Machine** that is working based on logical rules. The logical consistency is checked in the directly affected nodes as well as throughout the complete network via propagation of the new facts into all connected nodes.

- **Vanishing facts** result from fact propagation into a node that does not own all fact parameters as objects. Vanishing facts have to be set to FALSE.

Here the Tool Requirement 4 is confirmed that an inference mechanism for logical conclusion based on first order predicate logic rules is necessary. The software-technical integration of an **Inference Machine** is discussed in Section 6.1.6.

**Situation Assignment**

The specification of situations is, according to the overview in Figure 6.2 (page 75), the last step of \( PS_A \) specification. The details of this process are given by the activity diagram in Figure 6.10.

Situations in abstract process-structures have been introduced informally in Section 2.4.1. Also, examples have been provided there, see Figure 2.9 (page 17). Here, the definition of a situation is summarized:

Definition 15 A situation Sit in a PS\textsubscript{A} is a set of OCs that fulfills the following two conditions:

- All TPOs of the PS\textsubscript{A} are uniquely included
- The situation is part of the situation graph as defined in Appendix D, i.e. the set of OCs defines a valid intermediate state in a PS\textsubscript{A} that is reachable or transformable via the PS\textsubscript{A}’s COPs.

The necessary precondition for task planning with process-structures is the existence of an initial as well as a target situation, Sit\textsubscript{I} and Sit\textsubscript{T}. Situations that are adapted to a certain state of system and environment are at hand whenever the initial monitoring process [Prenzel, 2005] completes successfully. However, default initial and target situations are assigned during the PS\textsubscript{A} specification process. To do this, as explained in Figure 6.10, the PS\textsubscript{A}’s situation graph has to be expanded first. This method, which is also documented in Appendix D, enforces some preconditions to hold:

Definition 16 The successful expansion of a complete situation graph of a PS\textsubscript{A} requires that

- There are no unconnected or separate parts in the PS\textsubscript{A}
- Each situation produces a unique set of facts

The first condition is fulfilled when the maximum situation graph that is expanded according to the algorithm in the appendix includes all OCs of the given PS\textsubscript{A}. The fulfillment of the second condition is necessary to conduct the initial monitoring procedure. The required checking algorithm for unique fact sets can be found in the appendix, too.

Summary on Functional Analysis of Product Definition

So far the procedures of product definition for development Step 2a have been worked out on the level of functional analysis. This has taken place twofold, once for the specification of task representing elements and second for the specification process of the PS\textsubscript{A} network structure itself. In the following, the development of the tools that are proposed to be used during this development step is presented. The afore identified specification procedures will be embedded into these tools.

6.1.4. Tools, Methods and Repositories - Overview

Figure 6.11 displays the tools that are necessary for process-structure specification on the abstract level. The DBGUI serves for user-friendly and consistent management of task representing elements (TRE) and interfaces the MySQL-database containing these elements. The PSA-Designer is the programming interface for pictographic programming of abstract process-structures ($PSA$s). The PSA-Designer accesses the TRE database elements as administrated via DBGUI and produces abstract process-structures that are stored in the respective repository. Additionally the helper tool ConfigGUI is required. This is part of the FRIEND::Architecture framework and is used to configure database connection parameters, file and repository paths and several other relevant tool settings. In the following the development of DBGUI and PSA-Designer is presented. Subsequently, the mechanism and methods for pictographic modeling and verification of abstract process-structures are discussed.

**Note:** A description of tool development follows. Discussion of the process model continues on page 93 (EEOP Management).
6.1.5. Development of a Tool for Consistent Management of Task Representing Elements

The task representing elements TRE are formally summarized in the UML diagrams in Figure 4.5 (page 53) and Figure 4.7 (page 56). The development of a respective tool for their consistent administration and thus a fulfillment of Process Requirement 6 as well as Tool Requirements 1 and 5 is subject of this section. This means that the part of the tool that is related to EEO process management is included in the discussion here, even though it formally belongs to the FRIEND::Process development Step 2b as presented in Section 6.2.

As already anticipated when introducing the just referenced development product diagrams, a relational database is used to store the task representing elements. More concretely, a MySQL database is used, as MySQL is widely used, freely available and many tools and programming interfaces are available [MySQL, 2009].

The complete entity-relationship model (ERM) of the TRE database can be found in the IAT repository under MASSiVE/etc/DBstructure. This ERM reflects the relationship as also specified in the diagrams in the Figures 4.5 and 4.7.

Storing and retrieving of database information takes place by using SQL query files. These files contain respective SQL commands and are used from the FRIEND database interface.

FRIEND::Database Interface

Figure 6.12 shows the hierarchy of FRIEND::Database interfaces that are relevant for the TREs. These interfaces can be used according to a certain component’s requirements. CDatabaseManagement is the common base class to all database interfaces. It organizes database connection settings with the help of the configurator class CMASSiVEConfigDatabase, maintains the connection to the MySQL database itself and provides common methods like reading SQL files. CDatabaseReadStandardTypes is a lightweight interface from the viewpoint of dependencies to other data structures. It encapsulates methods to read data to standard data types, like strings and string lists, which are, in the concrete case, for example item names or the complete list of available item characteristics. CDatabaseReadSL and CDatabaseReadNWriteSL provide reading or reading and writing access for symbolic layer elements, i.e. TREs, which are depicted in gray on the right side. CDatabase unifies inheritance branches of other FRIEND::Database interfaces, that are used for the human machine interface, the world model or the machine vision framework, but are not relevant here.

Design of DBGUI

The complete overview of DBGUI is given in Figure 6.13. On the left side all TREs are grouped in a tree structure. The database server name is displayed at the top of this tree view (localhost in this case). On the right side of the programming interface a widget with respect to the kind of element to edit is loaded. In the current view in this figure the fact widget is activated.

2 also called MASSiVE database
3 A widget (or control) is an element of a graphical user interface (GUI) that displays an information arrangement changeable by the user, such as a window or a text box [Wikipedia, 2009]
Figure 6.12. Hierarchy of interface classes to access the task representing elements (TREs) in the FRIEND::Database.

Figure 6.13. DBGUI with fact management view. See also Figure 6.7 and the discussion below for a description how to insert a new fact into the database.

Figure 6.14. DBGUI design overview.

Figure 6.14 depicts the class overview diagram for the DBGUI tool. Central element is the *Database-Model*, which inherits from the class `CDatabaseReadNWriteSL` and thus has the mentioned interface for reading and writing TREs from/to the MySQL database. The database model organizes all database access with respect to the consistent management of TREs. For this purpose it holds compositions to the TRE data structures as shown on the right side of the diagram. The main visible component of DBGUI is provided via the `Window` class, which accesses the database model as well as a `DataTree` and an `Editor`. The data tree is the tree structure with the groups of TREs and the editor represents the right side widget which adapts to the current type of element to be edited. Finally the diagram reveals that the main window inherits from `QWidget`, which means it uses the Qt library [Nokia-Corporation, 2009]. Qt builds the basis for almost all graphical user interfaces in the FRIEND::Architecture, but is also used for further purposes like for example database access.

**COP Management**

Figure 6.15 displays the widget for managing COP data. This widget is loaded by the editor on the right side of DBGUI in the case a COP is selected within the TRE tree on the left. The screenshot shows all information that specifies the COP *GraspObjectInContainer* which is set up according to the activity diagram in Figure 6.6. This means, first the COP name is entered, the COP type is selected to ASSEMBly type (full type specification is given in Definition 8, page 79), the symbolic parameters *Manipulator*, *Object* and *Container* are selected, together with their set of characteristics. Finally, the COP’s pre- and post-facts have to be chosen. The ratings *Probability* and *Expense* are required for COP of type *MONitoring* as discussed in [Prenzel, 2005].

---

4In UML a composition is a special form of association between classes, where the component’s lifetime depends on the composing unit.
The FRIEND::Process: Task Knowledge Specification and Verification

Figure 6.15. DBGUI widget for COP management. Here, the COP \textit{GraspObjectInContainer} is selected. See also Figure 6.6 for a description how to insert a new COP into the database.

The data structures, functional classes and their relationship, i.e. the architectural design that is required for COP management is depicted in Figure 6.16. The design structures for all TREs are built up equivalently and therefore this one is discussed in more detail representatively.

The central class that contains the elements as depicted in Figure 6.15 is the class \textit{CopsWidget}. This class inherits, according to the multiple inheritance approach\(^5\), from the external Qt lib class \texttt{QWidget} (expressed by the stereotype\(^6\) \texttt{ExternalLibClass}) as well as from the user-defined and externally generated class \textit{Ui::CopsWidget} (expressed by the stereotype \texttt{ExternalGeneratedClass}). The latter one results from external translation of a \texttt{User Interface} (UI) file into the shown class, with the help of the Qt User Interface Compiler (UIC). The ui-file itself is the output of the Qt-Designer, which is used to compose the widget as depicted in Figure 6.15. A special integration of the Qt library an its external compilers has been developed. Details can be found in the IAT wiki under [IAT-Wiki, 2008].

As Figure 6.16 further shows, the central class \textit{CopsWidget} is accessed from the editor class and communicates with the database model as well as the classes \texttt{CParameterDialog} and \texttt{CopFactWidget}. The latter one is again realized via multiple inheritance approach to include the externally UIC-generated user interface class that realizes the fact widgets for the pre- and post-facts as they can be seen in Figure 6.15. Finally, the data structure for storing a complete COP prototype as it is internally used

\(^5\)Multiple inheritance approach in Qt: See also Qt Designer manual, [Nokia-Corporation, 2009]

\(^6\)Stereotypes are applied to UML modeling elements to assign certain properties, characteristics or just to serve as documentation. In general the purpose of UML stereotypes is to provide an extension mechanism to the basic set of UML. They allow designers to extend the vocabulary of UML in order to create new model elements, derived from existing ones, but that have specific properties that are suitable for a particular problem domain or otherwise specialized usage [Wikipedia, 2009].
or written to the database is made available with the help of the class **CCOP Prototype** and which is aggregated as a composition with *-multiplicity\(^7\) from the database model.

**Item Management**

Figure 6.17 depicts the widget for items management. The software-technical integration of this user front-end is, like also the following ones, done analogously to the COP management widget and therefore the further discussions are limited to functional aspects. This means, the user interface first receives the item name, abbreviation and type, i.e. *Bottle*, *Bo* and *ENVironmental type* here\(^8\). Subsequently characteristics can be assigned to the new or existing item, or new characteristics can be entered into the database, if necessary. In the case an item has sub-items, they can be specified at the bottom of the widget. Finally, the new or changed item is written to the database.

**System Management**

In Figure 6.18 the GUI front-end for system management is displayed. The procedure of system specification was formally introduced in the activity diagram in Figure 6.4. In the given case the system

\(^7\)In UML the multiplicity of associations indicates the number of involved objects. "*-multiplicity means that theoretically the number is infinite.

\(^8\)The complete specification of possible types is given in Definition 6, page 77

Figure 6.17. DBGUI widget for item management. Here, the item Bottle is selected. See also Figure 6.5 for a description how to insert a new item into the database.

Nemo\textsuperscript{9} is selected with the different sub-components (only their abbreviations are displayed). These are, according to the given example in the FRIEND::Process development Step 1a (see Figure 9.1 on page 170): The wheelchair system (WCS), the manipulator (MP) with robot arm (Ro), gripper (Gr) and force torque sensor (FTS), the stereo camera system (SCS) with pan-tilt head (PTH) and the camera (Bbee\textsuperscript{10}), as well as the intelligent tray (Tr) with tactile skin (Sk) and scale (Sc). The system facts that are specified for the system’s components are provided with default values.

Facts Management

Figure 6.13 already depicted the fact management widget as it is loaded in the editor area of the DBGUI application. The procedure of fact specification has been given in the activity diagram in Figure 6.7 and can be comprehended now along with the GUI front-end as follows: The currently edited element as shown is the fact $\text{FAC.AON.IsGripped(Manipulator, Object)}$ - a fact that is used on And-Or net level and gives a description about relation between environmental objects. The latter ones are defined with the help of the symbolic parameters (placeholders) Manipulator and Object and the applicability of item instances to them is as usually defined through parameter characteristics to be resolvable via type conformal parameter replacement (TCPR, Definition 9 on page 80). The complete set of literals of type and level enumerations as well as their descriptions is given by the Definitions 10 and 11, on page 81.

\textsuperscript{9}The system is named according to the currently used wheelchair platform (see glossary for more details), to distinguish it from the other FRIEND systems. It is also referred to as FRIEND III.

\textsuperscript{10}Bbee is the abbreviation for Bumblebee, see glossary for more details.
EEOP Management

For the sake of giving a complete description of the developments with respect to consistent TRE database management, i.e. the development of the DBGUI tool here, the specification process is anticipated in this section. As hitherto, the specification procedure is defined first and subsequently the GUI front-end is discussed. For easier understanding of the formal definition of an EEOP and its parts, it is also referred back to Figure 4.7 on page 56, where the products of $PS_E$ specification are summarized.

EEOP Specification

As Figure 6.19a shows, the EEOP specification starts with determination of EEOP name and the skill server where the EEOP is executed on. The next part of EEOP setup is the introduction of EEOP parameters, i.e. sub-symbolic parameters. This procedure is given in the sub-activity diagram in part b of the figure.

The $PS_E$ programming product specification in Figure 4.7 includes the formal definition of a sub-symbolic parameter as UML class. Here, this definition is refined:\footnote{The different levels of optionality of parameter parts is expressed with the help of the number of squared brackets. This means, if the level descriptor is set, at least a data descriptor has to follow.}
Definition 17 A Sub-Symbolic Parameter consists of

- **Symbol String**: Abstract symbolic name of an object or resource that is related to the parameter. In the first case this is the same symbol as it is used on the abstract programming level and thus establishes the interconnection between the two abstraction levels. Any of the symbolic parameters as available in the database are applicable here (e.g. "Object").

- **[Semantic Descriptor]**: The semantic descriptor is optional in any case and is used to refine the meaning of the parameter or to distinguish different information of the same type for the same symbol, e.g. different locations of sub-components. Here, the set of symbolic parameters is also applicable.

- **[Level Descriptor]**: One of the values \{EEL, HMI, TMP\}. "EEL" stands for "elementary executable level" and has to be set to denote sub-symbolic data, e.g. the location of an object. "HMI" is used for human machine interface data and "TMP" for temporary data.

- **[Data Descriptor]**: Specifies the type of sub-symbolic data that is represented by the complete sub-symbolic parameter string. Possible values are \{size of cuboid (SCub), size of cylinder (SCyl), size of composed object (SizeComposed), location (Loc), location of composed object (LocComposed), 4x4 matrix (Frame), color (Col), ...\}

- **[Source Descriptor]**: The source descriptor is optional in any case and specifies the source of sub-symbolic information, usually a description of the sensor, like "SCam", or the abbreviation "SDB" if the data is a priori data from the static database.

Additionally, a "Parameter Type" is assigned to each sub-symbolic parameter. However, this is not part of the parameter string.

- **Parameter Type**: Specifies how the parameter is used. Possible values are \{IN, OUT, INOUT, RES, TMP\}. The first three literals denote data flow directions, "RES" stands for "resource parameter", a parameter that names a resource to be used by the EEOP, and "TMP" denotes temporary parameters, which are for example only used skill-internally.
Each sub-symbolic parameter is maintained as a string and consists of one or more parts according to the just introduced Definition 17. Later, during skill development (see Chapter 7), the sub-symbolic information will be accessed from the world model with the help of this parameter string. In Table 6.1, some examples of sub-symbolic parameter strings are given.

<table>
<thead>
<tr>
<th>Sub-Symbolic Parameter</th>
<th>Meaning</th>
<th>Sample Sub-Symbolic Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mt.1.EEL.ObjectClassName</td>
<td>Name of object class</td>
<td>Mealtray</td>
</tr>
<tr>
<td>Mt.1.EEL.ObjectClassCharacteristics</td>
<td>Characteristics of object class</td>
<td>IsPlatform, IsFillable, IsGrippable,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IsPlaceable</td>
</tr>
<tr>
<td>Mt.1.EEL.Loc</td>
<td>Location (position, orientation) of object or,</td>
<td>PosX, PosY, PosZ, RotX, RotY, RotZ</td>
</tr>
<tr>
<td></td>
<td>in case of comosed object, its envelope</td>
<td></td>
</tr>
<tr>
<td>Mt.1.EEL.SizeComposed</td>
<td>Sequence of strings denoting the sub-components</td>
<td>Base, Plate, SpoonHandle, SpoonMainPart, InnerLid, Inner-LidHandle, OutLid</td>
</tr>
<tr>
<td>Mt.1.Base.EEL.SCub</td>
<td>Size of a cuboid, the meal-tray’s base in this</td>
<td>Length, Width, Height</td>
</tr>
<tr>
<td></td>
<td>case</td>
<td></td>
</tr>
<tr>
<td>Mt.1.EEL.GraspingSpecification</td>
<td>Information relevant for grasping the object, e.g.</td>
<td>DIRECT,CENTRAL</td>
</tr>
<tr>
<td></td>
<td>direct approach of the object in xy-plane towards the object’s center</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1. Examples of sub-symbolic parameter strings, the explanation of represented data and sample data for meal tray with current object symbol Mt.1.

After finishing the setup of parameter strings, characteristics are assigned to the parameters. This is done analogously and with similar purpose as for COP parameters - to be able to conduct the type conformal parameter replacement when using the EEOP prototype within a process-structure. If an EEOP parameter (sub-symbolic parameter) contains an object symbol from the abstract level, it is replaced with this concrete runtime object symbol, like e.g. shown in Table 6.1 for the meal tray Mt.1. Also, the Parameter Type is assigned to each parameter according to the list of available values from Definition 17.

For further verification steps as discussed in the Sections 6.2.4 and 7, it is important to specify, whether the parameter represents a resource or a certain sub-symbolic information parameter that is either required (IN type), provided (OUT type) or both (INOUT type). This specification is achieved through the assignment of the respective value to the attribute parameter type.

When the list of EEOP parameters is completely set up, the EEOP’s return values are specified. Only the set of return values as specified is allowed and respected within task planning. The process model will enforce the conformity to the specifications done here within the development steps as described in Chapter 7. Possible EEOP return values are e.g. Success for successful execution, Failure for errors during execution, Abort when the current task has to aborted or UserTakeOver, when the system request a user-involvement into ongoing task execution.

The final step of EEOP specification is the assignment of global resources. Global resources are those which do not physically belong to one of the COP parameters. A typical example for such a resource is the stereo camera system. The robot arm, on the contrary, is a sub-component of the manipulator and thus not a global one. Even though all names of the resources that are used by a certain skill have

Figure 6.20. DBGUI widget for EEOP management. The complete procedure of setting up a new EEOP specification is given by the activity diagram in Figure 6.19a.

to be passed as a resource parameter within the EEOP parameter list, the handling of global resources differs from local resources on Petri net level (see [Martens, 2003]).

Note: A description of tool development follows. Discussion of the process model continues in Section 6.2, page 113.

Integration of EEOP Management into DBGUI

The EEOP specification process as just formally introduced is realized by the EEOP management widget depicted in Figure 6.20. Like all other TRE management widgets it is loaded into the editor part in the DBGUI as soon as an existing EEOP is edited or a new EEOP shall be entered.

The screen-shot shows the currently selected EEOP AcquireObjectBySCam which runs on the Machine Vision skill server. This skill has three parameters: The stereo camera system Bumblebee as RESource parameter and the location and size parameter for the object to detect within the EEOP. Since the EEOP’s semantic specification is to produce the object’s location and size, these two parameters are of OUT type. Implicitly, the symbolic name of the object to detect is given within the parameter string. However, in order to minimize the number of parameters, this is not handed over as extra parameter of IN type, but the object symbol is extracted within the skill implementation. The dialog which enables the configuration of one EEOP parameter is depicted in Figure 6.21. There, the parameter as

Figure 6.21. DBGUI dialog for setting up sub-symbolic parameter strings as well as to determine the parameter’s type and characteristics. The associated activity diagram is given in Figure 6.19b.

currently marked in Figure 6.20 is edited. It is set to Object.EEL.Loc with parameter type OUT and the characteristic EEL_DATA, which is a characteristic that is automatically enforced for all parameters that represent data from EEL level. The remaining parts of EEOP specification can again be seen in Figure 6.20, where the possible return values are set to Success and Failure with probability ratings of 100% and 0% in this case. These ratings are used during the probabilistic simulation within skill testing as mentioned in Section 4.5.1 on page 63 and further discussed in Chapter 8.

Summary on DBGUI

Process Requirement 6 and its specializations in Tool Requirements 1 and 5 claim the necessity of a programming interface for user-friendly and logically consistent management of the database of task representing element prototypes. The DBGUI application offers a simple to use graphical front-end for convenient administration of these elements. On the one hand, the logical correctness of the input data is assured through the programming interface itself, as it guides the user through the specification process and limits user input to logically correct interactions. On the other hand, the input data is received internally in class structures, which assure that only correct, complete and consistent data is written to the database. The mentioned requirements are thus fulfilled.

The process step that follows the specification of task representing elements is the programming of abstract process-structures itself. The development and usage of the tool that facilitates this step is subject of the following section.
6.1.6. Development of PSA-Designer

This section documents the development of PSA-Designer, a tool for user-friendly and logically consistent pictographic programming of abstract process-structures. The discussions include the modeling methods for pictographic $PS_A$, the architectural design of the tool, logical verification methods for (pictographic) $PS_A$ and finally sample work-flows for $PS_A$ composition. The development of the user interface related parts of the PSA-Designer is also documented in [Boit, 2009]. First, the rough requirements that resulted from Chapter 4 are to be refined.

Refining the Requirements

Process Requirement 7 and the Tool Requirements 2 and 3 claim in general the necessity to develop a tool capable to guide the user through the specification process of abstract process-structures, including the specification of object constellations (OC), composed operators (COPs) and situations. The most important demand is to improve the way OCs are represented visually - i.e. a pictographic representation of OCs shall be pursued.

The refinement of requirements is started by capturing the system use cases, to structure and group the requirements. These use cases are derived from the functional analysis of product specification with respect to the graphical network assembly in Section 6.1.3. There, the programming steps for graphical specification of a $PS_A$ have been discussed. Additional interactions between programmer and programming interface are necessary, as given by the complete use case diagram in Figure 6.22.

The use cases are grouped into the three categories marked in blue: Building a $PS_A$, modifying it, and the necessary input/output operations. The first group contains the mentioned and already analyzed interactions. The group ModifyPSA includes the requirement that elements of a $PS_A$ network structure have to be moved or deleted, which is an obvious claim and will not be detailed further. The same is
true for the group containing the $PS_A$-input- and -output-operations: A pictographic network structure shall be saveable to the process-structure repository and it has to be retrievable again. The extraction of the $PS_A$ logic, i.e. the basic network information without graphical information, is required as input for the task planner. The modifying interaction $EditCOP$ is the same as deleting a COP connection and inserting another COP, i.e. the changing from one COP to another, equally applicable one, is demanded here. Only the modifying interaction $EditOC$ needs some further detailing.

Figure 6.23 shows the refinement of the use case $EditOC$. First, TPOs can be added to or removed from an object constellation. This is only allowed for unconnected OCs, since the involved objects of connected OCs have been inserted into the connecting COP prototype(s). For changing an OC the same mechanisms are applied as when initially setting up an OC as discussed in Section 6.1.3, i.e. the combination of objects is checked for physical correctness, as floating objects are not allowed.

To produce pictographic representations of object constellations, a 3D scene for modeling the OC is proposed. From this 3D scene, the pictographic snapshot for the OC representation in the 2D network structure will be captured. To assemble a visually meaningful 3D object constellation, the objects have to be movable in the 3D scene. Several constraints apply for the freedom of object movements, according to the current constellation of objects. The following constraints are identified:

**Definition 18** Constraints for a 3D object constellation ($3dOC$) are:

- Objects are either gripped or placed, to avoid floating objects.
- Gripped objects cannot be move, as they are child objects of the gripper.
- Placed objects can only be moved on the platform where they are placed on.
- Containers include (at least virtually) a platform to place objects on.
- Symbolically represented sub-symbolic items (SSIs) are visualized and are used as discrete 3D locations, e.g. the SSI PlacedLoc describes a default placing location on a platform, usually in the center, EdgeLoc describes a location at the edge of a platform.

The constrained moving of objects as well as the assignment of SSIs is part of the use case / requirements refinements for editing OCs in Figure 6.23. The remaining use case $CaptureSnapshot$ means, that the view on the composed object constellation can be adjusted and finally a 2D snapshot is extracted to represent the OC in the 2D network structure in the desired pictographic manner.
To give the reader a better understanding of the tool and methods to be developed a preview of the expected results is given beforehand: Figure 6.24 shows the graphical user interface of PSA-Designer with pictographic representation of a \textit{PSA Fetch Bottle}, which is analogous to the process-structure \textit{Fetch Cup} that has been used for exemplification in the Sections 2.4.1 and 4.3.1.

The discussion of PSA-Designer development is structured as follows:

- **Large scale design of model managers:** The management of PSA is separated by the logic manager and graphic manager
- **Logic manager:** Data structures for logic management of PSAs
- **Logic manager:** Architectural design
- **Graphic manager:** Data structures for pictographic synthesis of object constellations
- **Graphic manager:** Architectural design
- **Graphic manager:** Behavioral statechart-based design
- **Graphical front-end:** Embeds all managers and modeling methods and is responsible for the visualizations

Large Scale Architectural Design of Model Managers

The large scale design of the core classes that are responsible for modeling of pictographic $PS_A$ is depicted in Figure 6.25. This design does not strictly adhere to the Model-View-Controller design pattern [Gamma et al., 1994], since there is no common controller for view and model. However, the logical and graphical parts are properly separated. The class $CPSALogicManager$ is responsible for logical modeling of $PS_A$ and offers all interfaces that are necessary from the viewpoint of the $CPSAGraphicManager$ class. The logic manager encapsulates the core logic which is realized by the class $CEnvNSys$. This class contains and logically organizes all information about the environment and system. Logical rules that supervise the logical correctness of object constellations and their sets of facts as detailed below is separated in $CTaskKnowledgeRules$. The graphic manager’s core helper class is $CPSAQtObjectConstellationManager$, which represents 3D object constellations by using the Qt library [Nokia-Corporation, 2009]. The development of the logic modeling is discussed in the following Section 6.1.6 and Section 6.1.6 contain the documentation of graphical modeling methods and related classes.

Development of PSA Logic Manager - Data Structures

Figure 6.26 shows the main data structures that are used for the modeling of the logical And-Or net layer of a $PS_A$ as they are accessed and managed by $CPSALogicManager$. Central element is the class $CAndOrNet$ which represents the basic network structure with the help of an And-Or net [Cao and Sanderson, 1998]. The network consists of a set of $CNodes$, which is realized as the *-aggregation $m$*$NodeList$. The nodes are connected by $CAndArcs$ from the *-aggregated set $m$*$AndArcList$ in a bi-directional way: The And-arcs own an association to their start node as well as a set of end node pointers. The nodes on the other hand have information about leaving and incoming And-arcs. Similar bi-directional connections are established for internal state transition (IST) arcs. The difference with IST arcs is that they are implicitly associated with the And-Or net via the nodes only. A node has furthermore a contact state, which stores the objects that are involved in the physical object constellation assigned to the node. Finally, situations as introduced by Definition 15 (page 85) have to be assigned to a $PS_A$. The class $CValidMarking$ is responsible for this and therefore maintains the situation graph for the $PS_A$. The large scale design of model managers showed that the core logic of a $PS_A$ is managed via EnvNSys. These core logic classes, their inter-relationships as well as their integration into EnvNSys is depicted in the class diagram in Figure 6.27. EnvNSys loads the prototypes of task representing elements (TRE) from the database, using the database interface as described in Section 6.1.5 (page 87). From these
prototypes, TRE runtime elements are created and managed according to the $PS_A$ currently under construction: $CItemInstance$, $CFAC_Instance$ and $CCOP_Instance$.

EnvNSys provides grouped sets of available item prototypes. All prototypes that represent physical items are composed in $m_{ProtosPhysicalItems}$ and the symbolically represented sub-symbolic items are supplied via $m_{ProtosSybsymbolItems}$. The first set is then also split up into $m_{ProtosEnvironmentItems}$ and $m_{ProtosSystemItems}$. Important part of EnvNSys is also the set of $CObjectConstellations$. Here the logical part of OCs is maintained, together with a $CContactState$ for the management of a unique OC string inside a network as well as the set of facts that are assigned to an OC. Here, it is distinguished between $m_{PossibleFacInstances}$ and $m_{LegalFACInstances}$. The first set contains all facts that are possible according to all possible permutations of items contained in the OC, the second set is the set of facts that are also called interesting facts, denoting those facts and fact values as induced via connected COP.

**Development of PSA Logic Manager: Architectural Design**

The interfaces of the PSA logic manager that are relevant for the graphic manager are given in Figure 6.28. Also, additional architectural helper classes are shown. In the following, the implemented methods of the helper classes as well as the interfaces of the logic manager will be detailed.

**Inference Machine:** The component to fulfill Tool Requirement 4, i.e. to provide a mechanism which is able to check the logical consistency of fact sets in object constellations is interfaced by EnvNSys via $IInferenceMachineConnector$. This interface provides two methods to set or retrieve facts and to check the logical consistency of a set of facts. The inference machine works on the basis of *Amzi Prolog*. At start up of the inference machine, a set of logical rules is loaded which encodes rules according to the facts that are relevant for the covered scenarios, e.g. the AMaRob scenarios. This set of rules has to be

![Figure 6.27.](image1)

**Figure 6.27.** Data structures that are managed by EnvNSys.

![Figure 6.28.](image2)

**Figure 6.28.** Architectural design of PSA logic manager and related classes.
extended, if new fact prototypes are added to the TRE database according to the procedure described in Section 6.1.2. As described in more detail in [Martens, 2003], the rules are specified with the help of Horn clauses. A Horn clause is a disjunction of literals with at most one positive literal. An example of a Horn clause formulated in Prolog is as follows:

```prolog
1 pFAC_AON_not_IsPlacedOn(Object, _):-
2     pFAC_AON_IsInFreePos(Robot),
3     pFAC_AON_IsGripped(Robot, Object),
4     !,
5     asserta(pFAC_AON_not_IsPlacedOn(Object, _))
```

This rule reasons that if the Robot grasps an Object and meanwhile, the robot is in a free position in the workspace, the object cannot be placed on some other item (denoted with "._"). With the help of these rules, the insertion of a new fact into an object constellation makes it possible to derive new facts from it on the one hand, but also can reveal logical contradiction of facts, which consequently leads to the rejection of fact insertion.

**Situation Manager:** The situation manager determines the set of all possible situations in a PSA according to Definitions 15 and 16. If the expansion of a maximum situation graph reveals that a precondition is not fulfilled, a respective error value is returned. In the other case, when successful expansion is possible, the set of situations can be retrieved from the logic manager and default initial and target situations for the PSA can be determined.

**PSALogic-IO:** The in- and output of the logic layer of a PSA is realized with the help of the packages EnvNSysIO and AON.IO, which are connected to the logic manager as shown in Figure 6.28. The in- and output of And-Or net related parts and EnvNSys related parts is separated, even though they may use one file for storing the PSA logic. In order to be able to store a complete pictographic PSA, it is also possible to stream the logic infos into or from a string and to combine it with the IO stream for graphical elements. This satisfies the requirements as identified in the use case analysis in Section 6.1.6.

**Development of PSA Graphic Manager: Data Structures**

In this section, the data structures related to the visualization layer of pictographic PSA programming will be discussed. Due to the fact that logical and graphical modeling elements are managed separately, the data structures for graphical modeling will enhance the logical data structures (Figure 6.26) in object-oriented manner. Figure 6.29 depicts how this is realized: The logical elements, marked in gray, are associated with their graphical counterparts. This means, the complete pictographically represented PSA is contained in CGraphicPSA, which aggregates a set of CGraphicObjectBase. The latter one is the common base class to the three graphic core elements CGraphicObjectConstellation, CGraphicAndArc and CGraphicIstArc. The (picto)graphic object constellations has a representation in 3D as well as in 2D, which will be part of the 3D-scene and 2D-scene respectively. The 3D-scene will have to implement the functionality as analyzed in the use case diagram in Figure 6.23 on page 99. The, PSA arcs, i.e. And arcs and IST arcs, have no 3D representation, but only a graphical representation within the 2D scene (C2DSOAndArc and C2DSOIstArc). For the sake of simpler handling the 2D scene objects also have

\[SO = \text{scene object}\]

Figure 6.29. Core data structures for (picto)graphic PSA modeling.

Figure 6.30. 3D visualization-related data structures for pictographic PSA modeling.

A common base class, C2DSceneObjectBase, so that all objects can be stored in one data container and can be handled in a polymorph manner.

During refinement of the requirements for PSA-Designer in Section 6.1.6, the general approach of pictographic OC modeling as pursued in this work has been introduced: TPOs as configured in the database are modeled in a 3D scene and their composition to object constellations can be configured by the programmer. The configurability is limited to constrained degrees of freedom for moving the objects in the 3D space. The applied constraints have been given by Definition 18.

The core class that realizes the 3D visualization of object constellations, also called viewport, is CPSAMVRView, which is depicted in Figure 6.30 together with related functional classes (green) and related data structures (gray). OpenGL (Open Graphics Library, [OpenGL.org, 2009]) is used for 3D object rendering and display. OpenGL is a standard specification that defines a cross-platform API to realize applications with 2D or 3D computer graphics and is embedded into the QtOpenGL module provided by the Qt library [Nokia-Corporation, 2009]. As shown in the class diagram, the visualization core class inherits from CWorldGLWidget, which embeds the Qt/OpenGL-based object rendering and is also used in other parts of the FRIEND::Architecture software framework, e.g. in the mapped virtual reality (MVR), which is used for visualization of robot manipulations in its workspace.
CSceneBase is the common base class for the visualization core as well as for the 3D-scene, whose integration is detailed in the next section. The main 3D view contains a set of basic objects via aggregation of CPSAMVRObjectBase. Within this set, one of the objects can be selected by the user as currently active object (m_pSelectedObjectBase). Also, a parent-child hierarchy is modeled within the visualized object’s base class. This is imposed by the 3D constraints of Definition 18 for synchronization of spatial movements of child objects with the movements of their parents objects (e.g. for moving a platform with the objects placed on it or the gripper together with the gripped object). The 3D modeling of concrete objects is provided in the package PSAMVRObjects. In Figure 6.30 only a subset of available 3D objects is shown, but all objects as defined in the AMaRob ontology of task participating objects are provided (Section 5.1.3, Figure 5.3, page 70). The classes CGroupObject and CMVRGroupObject that are used by the PSA visualization view provide methods to create and re-configure groups of 3D objects.

Development of PSA Graphic Manager: Architectural Design

Figure 6.31 depicts the architectural design of the graphic manager and related classes. The core classes are represented in blue and related functional classes are marked in green. The 3D-scene is interfaced by CPSAQIObjectConstellationManager, which manages 3D object constellations together with CPSAObjectConstellationManager. The first manager class contains the Qt-related elements and methods, i.e. everything related to the graphical user front-end. The second manager class embeds basic functional methods and therefore accesses the PS_A logic manager.

The three classes that are marked in red represent statechart classes which implement and encapsulate state-based behavior of the aggregating classes. This means, the graphic manager, the 2D scene as well as the object constellation manager contain this kind of behavioral implementation. The details of the implemented statecharts are subject to the next section.

Figure 6.32. Statechart for 2D scene.

**Development of PSA Graphic Manager: Behavioral Design**

The implemented statechart for the 2D scene is depicted in Figure 6.32. The three active states have the following meaning:

- **AddSO**: Adding of scene objects (SO = scene object). 2D scene objects are object constellations, arcs and IST arcs.

- **SelectedSceneObject**: Selection of single or groups of 2D scene objects with the purpose to move or delete them.

- **MoveSO**: Moving of 2D scene objects.

These states are connected to several use cases as identified in the use case diagram in Figure 6.22.

The statechart of the graphic manager is depicted in Figure 6.33. It has the same states as the 2D scene, which are activated at the same time in both statecharts due to same triggering events. However, besides common states the graphic manager has additional states:

- **EditArc**: Change the composed operation(s) assigned to the arc.

- **EditOC3D**: Switch scene to 3D object constellation edit mode. Here the OCs can be composed with the help of 3D objects.

- **EditOC2D**: Switch scene to 2D object constellation edit mode. Here, only the textual representation of OC parts can be modified.

- **EditSSI**: Edit sub-symbolic items (SSI). Graphical representation of SSI can be assigned to task participating objects (TPO).

The statechart for the object constellation manager as depicted in Figure 6.34 again contains overlapping states with the previous statechart. In this case these are the states for editing either the 2D or 3D
Figure 6.33. Statechart of graphic manager.

The new state serves to mark the SnapshotMode. In this mode, the view on the previously composed 3D object constellation can be configured for eventually taking the snapshot that will then be integrated into the 2D network representation.

**Graphical Front-End**

The discussion in this section now unifies all developed parts of the PSA-Designer as presented within the preceding sections. On the one hand it is referred back to Section 6.1.6, i.e. the refinement and structuring of the requirements for PSA-Designer. On the other hand, the specifications as gathered throughout the functional analysis in Sections 6.1.1 and 6.1.3 have to be reviewed in order to compare the desired functionality with the actually implemented one as summarized here. Especially important is the complete overview of the PSA specification process as given by the activity diagram in Figure 6.2 on page 75.

The first step in the specification process is the selection of task participating objects. This is done via the dialog as shown in Figure 6.35. In the shown dialog, the physical items *Bottle* (Bo), *Fridge* (Fr), *Manipulator* (MP), *Tray* (Tr) and the sub-symbolic items *InsertLoc* and *PlacedLoc* have been selected. If the user wants to load an already existing network, he selects this option from the this dialog.

The next screenshot in Figure 6.36 shows the 3D scene for configuration of an object constellation. The implemented functionality is according to the functional analysis of Section 6.1.3 and the different architectural and behavioral elements as previously discussed in the preceding sections (e.g. the 3D visualization data structures in Figure 6.30 and the statechart of the object constellation manager in Figure 6.34). On the right side of the application’s screen-shot the 3D object constellation rendering scene is shown, where a bottle in a fridge is modeled. In the button toolbar of the shown view, the user can accept the current configuration, cancel it, directly take a snapshot of the current object constellation, or switch to the snapshot mode to change the view on the scene and subsequently take the snapshot.

Figure 6.35. Dialog for selection of task participating objects (TPO).

Figure 6.36. 3D scene for editing an object constellation.
Figure 6.37. COP selection and connection of pictographic object constellations.

Figure 6.37 shows the COP selection and assignment process. Here, the functionality as analyzed in Section 6.1.3 and detailed in Section 6.1.6 is implemented. First, the assembly rules are checked according to Definition 12 and subsequently the list of possible COPs is calculated based on the rules as given by Definition 13. This list is offered to the programmer for selection of the desired COP(s), as shown in the screenshot. Before final insertion of the COP(s), the fact propagation throughout the network is performed and the logical consistency of each node’s set of facts is verified based on Definition 14 and with the help of the inference machine as introduced in Section 6.1.6.

The final step of completing the $PS_A$ specification is the assignment of default initial and target situations according to the rules defined in Section 6.1.3. A screenshot of the graphical front-end showing the Situation Navigator Dialog is given in Figure 6.38. If no conflicts with respect to Definitions 15 and 16 are found, the complete situation graph (Section 6.1.6) is available and situations can be selected for the network. The situation that is currently marked in the list of available situations in the dialog is also highlighted in the network structure (yellow). Furthermore, the initial situation is marked in red and the target situation is marked in green. In the shown case, the selected initial situation sets the bottle to be located in the fridge, the manipulator is in a free location and the tray is empty. Within the target situation, the fridge is empty, the manipulator in a free location again, and the bottle is placed on the tray.

Summary on PSA-Designer

Based on the refinements of requirements as given in Figures 6.22 and 6.23 the methods for pictographic synthesis of object constellations as main achievement of PSA-Designer have been realized. Besides this important aspect which targets at user-friendly and intuitive programming of abstract process-structures, further integrated programming support and verification methods have been embedded. The
Figure 6.38. Complete pictographic $PS_A$ with situation navigator and assigned initial and target situations.

logical manager in the PSA-Designer keeps track of the overall logical correctness of $PS_A$s and filters logically meaningful task representing elements from the prototype database according to a certain programming context. The graphic manager assures the correctness of a pictographic PSA on the visual level and organizes the user-friendly setup of the 3D-OCs and the network structure. The complex behavioral design of the graphic manager is realized with the help of executable UML statecharts. They serve as implementation-consistent documentation and facilitate(d) the debugging and future enhancement of the application. The necessary steps of using the graphical front-end of PSA-Designer to configure a $PS_A$ have been demonstrated and discussed and will be further evaluated in Chapter 9.

A direct coupling of PSA-Designer and the PSE-Designer (to be discussed in the following) is also available: A right-click on a certain COP-arc leads to a selection dialog, where either the assembly- or disassembly-COP can be edited on PSE-, i.e. function-block-network-level.

<table>
<thead>
<tr>
<th>Input artifacts:</th>
<th>System, Items, COP, Facts</th>
<th>A2.1–4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output artifacts:</td>
<td>EEOPs, Elementary process-structures (PSE)s</td>
<td>A2.7+8</td>
</tr>
<tr>
<td>Tools:</td>
<td>DBGUI, PSE-Designer</td>
<td>T2+4</td>
</tr>
<tr>
<td>Repositories:</td>
<td>Task Knowledge Database, Process-Structure Repository</td>
<td>R2+3</td>
</tr>
</tbody>
</table>

6.2.1. Definition of Products - Overview

Again, as for development Step 2a, the formal summary of in- and output products of the FRIEND::Process Step 2b have been anticipated in Figure 4.7 on page 56. One of the central development artifacts are the EEOPs. The specification of new EEOP prototypes in the database, including functional analysis, design and implementation, was already matter of Section 6.1.5 along with the discussion of the DBGUI development. The further task representing elements that are required as shown in Figure 4.7, i.e. COP with symbolic item parameters, facts and the target system (the latter one not shown in the figure), are input products from the previous development step, if a top-down process is carried out.

The complete overview on the PSE specification process with function blocks is given in Figure 6.40. The functional analysis of this process is done in the following, building the basis for the discussion of the subsequent tool development.

6.2.2. Functional Analysis of Product Definition: PSE Specification with Function Block Networks

To fulfill the various requirements with respect to PSE programming as gathered in Section 4.6, new methods using function block networks (FBN) for the programming task will be developed here\(^\text{13}\). They target at the construction of network structures as exemplified in Section 4.3.2 with Figure 4.8 (page 57). The complete set of FBN building blocks is given as structured overview in Figure 6.41. This set of building blocks matches the demand as formulated by the Tool Requirement 10, i.e. four categories of building blocks are defined: EEOP blocks, Fact blocks, Logical blocks and Control blocks. Logical

---

\(^{13}\)The methods for PSE programming with function block networks have been published in [Prenzel et al., 2008, Kampe, 2008]
blocks are used to create AND- or OR-linked control paths in an FBN; the control blocks Start and Target are mandatory and unique in each FBN in order to have a well-defined start and end of the control flow, the Abort block is optional, since abort connections may be implicitly modeled as to be discussed below, but the block is also unique within one FBN.

Referring back to the overview diagram of the complete specification process in Figure 6.40, it is shown that \( PSE \) programming starts with the selection of the target system. This system should normally already exist in the TRE database due to previous development steps. However, in order to keep the procedures independent from each other, the specification of a new system may take place first according to the procedure as specified in Section 6.1.2.
An elementary process-structure $P_{SE}$ represents the decomposition of one composed operator (COP). Therefore, the next mandatory step is to determine the COP to be modeled in the $P_{SE}$. This selection process is either done explicitly from the list of available COPs in the database or as interactive procedure from $P_{SA}$ level via COP arc context menu. Again, if not yet specified, a new COP may also be introduced in the database according to the descriptions in Section 6.1.2. After completion of these preparatory steps, the setup of the function block network starts.

Add EEOP Blocks

The procedure of adding an EEOP function block is detailed in the sub-activity diagram in Figure 6.42. First the name of the EEOP is chosen, or a new EEOP specification is first introduced in the database according to the activity specification described in Figure 6.19 (page 94). The skill server name, which the skill execution module later needs in order to determine on which server to activate the EEOP execution is implicitly encoded in the EEOP name and thus not explicitly specified. To complete the EEOP block setup, the symbolic part of the EEOP’s formal parameters (aka sub-symbolic parameters, see Section 6.1.5) have to be mapped to the formal symbolic parameters of the COP prototype that is assigned to the currently programmed $P_{SE}$. This process is called parameter mapping and is necessary to propagate the runtime parameters (e.g. Bo.1 for a bottle) that are inserted into the COP prototype to the EEOP prototypes that are part of the COP decomposition. The parameter mapping principle is summarized in the following definition and is referenced again in Section 6.2.4.

**Definition 19**

**Parameter Mapping:** The EEOP blocks have a set of formal parameters, which is, for the EEOP blocks, the set of symbolic parts of all parameters of one EEOP prototype. The formal parameters of the COP prototype that is assigned to the currently modeled $P_{SE}$ must be mapped to the set of formal EEOP parameters. This mapping is done with the help of parameter characteristics that are defined in the database of task representing elements. The underlying checking routine is again the type conformal parameter replacement as introduced in Definition 9. The parameter mapping is done analogously for fact blocks.

Add Fact Blocks

The procedure of adding and configuring a fact block is basically equivalent to EEOP block setup. Therefore, an extra activity diagram is omitted. The procedure is, that first the fact name is selected.
Here, only names of facts should be offered that accept as parameter the symbolic formal parameters of
the current COP according to the TCPR principle. This formal parameter mapping then takes place in
the similar manner as in the case of EEOP blocks, based on the parameter characteristics as configured
in the TRE database.

Add Links

The connection between different function blocks is established through links. The links model the
control flow in the PSE. While gathering the requirements in Section 4.3.2 it has been decided that the
function blocks as used in the FRIEND::Process will use the port notation as known from UML. On the
left side the required ports are arranged and the provided ports are at the right side of a function block
(Tool Requirement 9). In the following the linking rules will be defined:

Definition 20

FBN Linking Rules:

1. Only links from provided to required port are allowed.

2. Several links may leave a provided port, but the required port only accepts one link (N to
   1 connection). The reason is that a logical control flow connection (AND/OR) has to be
   established explicitly with the help of the logical blocks to achieve a clear interpretation of
   the network structure. Parallel EEOP execution is modeled by parallel leaving branches at
   provided ports (Tool Requirement 11).

3. The provided port of the Target block only accepts connections to fact blocks, to set all post-
   facts of the modeled COP (Tool Requirement 12) and thus to establish the interconnection
   from elementary to abstract process-structure.

4. The connection of Fact block ports is optional, except the mandatory setting of post-facts
   as defined in the previous Linking Rule 3. The connection of the EEOP Abort port is also
   optional, since an unconnected Abort port is implicitly interpreted as execution abort. All
   other ports have to be connected to produce a complete and valid PSE.

5. Direct recursion from provided to required port of the same function block is not allowed.
   This is considered as modeling mistake.

The linking rules are checked whenever a link is established in the function block network. The sum-
marized actions for link setup are given by Figure 6.43

The insertion of logical or control blocks is not discussed separately, since there are no special config-
urations required for these block types. Also the discussion of deleting or editing elements is postponed
to Section 6.2.4 where the implementation details of PSE-Designer are presented. The remaining action
that is required to finish the programming of a function block network according to Figure 6.40 is the
final verification of the new PSE and its storing in the process-structure database.

Figure 6.43. Procedure of adding a link in a function block network.

**Verify and Save** $PS_E$

Figure 6.44 shows the different actions that are required for finalizing the $PS_E$ specification procedure before exporting it to the process-structure repository.

**Check Data Flow:** The checking of the correct data flow in a $PS_E$ is done on the basis of parameter types that are specified for each EEOP parameter according to Definition 17 (page 94). That means, the sub-symbolic information that is denoted by a parameter of type $IN$ or $INOUT$ must have been produced beforehand with the help of an EEOP block that has the same parameter as $OUT$ type. For example, the location of an object to be grasped must be calculated by a preceding monitoring operation, e.g. with the help of the stereo camera system.

**Check COP Post Fact Conformity:** It is checked, whether the post-facts of the modeled COP are set correctly within the $PS_E$ network with respect to the specification of the COP.

**Adjust Petri Net Marking in Database:** The Petri net resulting from FBN conversion requires a default initial marking. The set of places in the Petri net that belong to the marking can be gathered solely on the basis of FBN information. The following rules apply (see also Section 6.2.5):

**Definition 21**

**Default Initial Marking in Petri Net:**

1. A Start place, corresponding to the Start function block.
2. The places that mark the COP’s pre-fact states.
3. The places that mark the availability of resources.
4. All places that mark the absence of sub-symbolic data (e.g. TMP.PN.*not*.DataAvailable(Data)).
During the currently described procedure it is checked, whether the default initial marking for the COP as it is currently stored in the database has to be modified or not.

Conversion to Petri Net and Verification: The function block network is converted to a Petri net with the methods as described in Section 6.2.5. With the help of the Petri net it is verified, whether the network’s target is reachable. This check is only possible with the help of the Petri net’s reachability graph. This graph is also used to find out transitions that form isolated parts of the net, what gives a hint to a modeling mistake, as this unconnected network part is not reachable.

6.2.3. Tools, Methods and Repositories - Overview

Figure 6.45 contains the tools, repositories and the flow of programming artifacts that are necessary for developing elementary process-structures. As in the case of abstract process-structures the TRE database management tool DBGUI is used to set up the task representing elements. The development

and usage of DBGUI was subject of Section 6.1.5. The tool that realizes the specification of the $P_{SE}$ network structure with the help of function block networks is the $PSE$-Designer. $PSE$-Designer embeds all the specification steps that have been identified during the functional analysis in Section 6.2.2. The development of the user interface related parts of $PSE$-Designer is also documented in [Boit, 2009]. The complete development process and usage of $PSE$-Designer is presented in the following.

**Note:** A description of tool development follows. Discussion of the process model continues in Section 6.3, page 130.

6.2.4. Development of $PSE$-Designer

The discussion of $PSE$-Designer development includes the following sections: First, all its requirements that have been gathered in Chapter 4 are structured with the help of a use-case analysis. These requirement refinements also include the results that have been gathered throughout the functional analysis of the FBN specification process as documented above. Then, the methods for modeling and verification of function block networks are introduced, including the architectural design of the FBN model manager and the required classes for data structuring. Furthermore, the design and implementation of the graphical front-end is presented and the embedded parameter mapping method is discussed:

- Refinement of Requirements
- Large Scale Design of $PSE$-Designer
- Data Structures and Model Manager for Logical Layer
- Function Block Network Verification Methods
- Core Data Structures and Controller for Graphical Layer
- Behavioral Design of $PSE$-Designer
- Graphical Front-End
- Parameter Mapping

**Refinement of Requirements**

The main use cases of $PSE$-Designer are depicted in Figure 6.46. This use case diagram is a grouped representation of the steps that have been identified within the functional analysis and that are contained in the overview of the $P_{SE}$ specification process as shown in Figure 6.40.

The four main groups of programming interactions include configuration steps, the building and modification of a $P_{SE}$ structure, as well as in- and output operations for FBN-based $P_{SE}$s. The sub-use-cases that are included in these four main use cases have their obvious counterparts of functional descriptions in Section 6.2.2. All these functionalities together, embedded into one programming tool, will lead to the fulfillment of the identified requirements, particularly, as the function block networks will
hide the $PSE$ complexity (Process Requirement 8),

- the user will be guided through the specification process with maximum amount of automation in each programming step (Tool Requirement 6),

- verification routines are integrated (Tool Requirement 7) and

- overall integration of the FBN based approach into the development process FRIEND::Process (Tool Requirement 8).

Large Scale Design of PSE-Designer

The $PSE$ Designer has been implemented according to the Model-View-Controller software design pattern [Gamma et al., 1994] to separate the visualization, modeling and logical control of a function block based $PSE$ from each other. Figure 6.47 contains the respective large scale design. The base classes QGraphicsScene and QGraphicsView are the core parts of the Qt Graphics View Framework [Nokia-Corporation, 2009], which is designed to display 2D graphical scenes capable to display thousands of items in real-time. The scene controller administrates all items that are contained on the complete scene, whereas the view renders the subset of items that is currently visible, e.g. due to zooming or scrolling.

The scene controller manages the status and position of all items and forwards scene events that influence the current status of the network to the logical network model. The latter one is managed by the class CModelManager. The forwarded scene events are mouse clicks/drag/release events and FB item interactions (insert/delete/edit). The scene also takes care of event propagation caused by user interaction with the view that do not affect the network model (zooming/scrolling). The MainWindow class embeds scene and viewport(s) into the PSE-Designer main application.
Data Structures and Model Manager for Logical Layer

The Model Manager takes the responsibility for the composition of a logically correct function block network. As depicted in the diagram in Figure 6.48 it maintains the sets of TRE prototypes as retrieved from the database and it provides all the methods to build up the logical structure of a function block network from these basic elements.

A function block network is represented and organized with the help of the class \texttt{CFunctionBlockNetwork}, which essentially composes a set of function blocks as well as a set of function block links. A function block, as modeled via \texttt{CFunctionBlock}, represents any function block according to the overview in Figure 6.41 on page 114 with the help of the \texttt{FUNCTION\_BLOCK\_TYPE\_E} enumeration. Function blocks (as well as function block links) have unique IDs in the network. With the help of these IDs, the function block links store the start and target function block for example. A function block, if of EEOP or fact type, owns a set of formal parameters, either sub-symbolic ones in the former case ore symbolic ones in the latter case. The mapping as described in Definition 19 from function block parameter to COP parameter is done with the help of a parameter map. Finally, a function block contains a set of input and output ports (required and provided ports), marks its conformity to the database prototypes and stores the name of the actually used TRE prototype.

Function Block Network Verification Methods

After the selection of a particular COP that shall be modeled as a function block network, the model manager maintains the consistency of the model throughout the programming process as initiated by the user from the graphical front-end. For this purpose, the manager contains several checking routines that supervise the FBN construction process online.

Conformity of Parameters: The conformity of parameters is checked according to Definition 19 during the parameter mapping process.
Uniqueness of Blocks: With the exception of logical blocks (AND and OR Blocks), each block exactly appears only once within an FBN. This means, there is one Start Block, one Target Block, and optionally one Abort Block. The Fact Blocks are unique with respect to the fact’s name and their set of formal parameters. The EEO Blocks are unique with respect to their configuration consisting of server name, EEOP name, and set of formal parameters.

Consistency of Linkage: Whenever the programmer tries to establish a link between function blocks, the checking procedures as specified in Definition 20 are applied to allow or reject the linking.

Data flow: The correctness of the data flow within a PS_E is checked based on the parameter types (IN, OUT, INOUT). An IN-parameter must have been produced with the help of an OUT-parameter of a preceding function block.

Structure of Petri nets: Function blocks and links are transformed to their Petri net counterparts with the help of the methods to be discussed in Section 6.2.5. Additional logical and organizational Petri net elements model complexity hidden by the FBN design.

Explicit Formal Tests: Although most of the error avoidance is achieved by guiding and limiting the actions of the user, some specific aspects cannot be covered in this way. The validity test can be invoked anytime during the network building process and checks the following conditions: A possible path must exist from the Start Block to the Target Block that can be executed by the task planner. All specified post-conditional facts must be set in succession of the Target Block. Special EEOPs need data generated by other EEOPs and any EEOP Block with missing but required data is reported (checking for correct data flow).
Core Data Structures and Controller for Graphical Layer

The essential data structures that realize the graphical elements of a function block network are shown in the diagram in Figure 6.49. The most important visualization elements are graphical links and graphical function blocks. The different required kinds of function blocks as also contained in Figure 6.41 derive from the common base class, which itself is a child of the `QGraphicsItem` class provided by Qt to realize general graphical items within the scene/view. The graphical links between the function blocks also inherit from a Qt base class, namely `QGraphicsPathItem`, which is especially suitable because its shape can be defined exactly for 2D collision detection with other items. The graphics controller class `CQtScene` only aggregates one graphical link as well as one graphical function block object (pointer) to set up a new item of either type. During this item configuration, the function blocks are handled polymorphically and thus equally. Finally, when the item configuration is completed, it is pushed to the QScene base class which organizes the management further on.

Behavioral Design of PSE-Designer

The states of the scene controller are specified as well as implemented with the help of a statechart. Figure 6.50 depicts those states that are related to the function block network model editing. Whenever the user requests an action causing a change in the network model from the tool bar in the GUI (e.g. delete/insert FB or link between FBs), the GUI forwards this information to the model which in turn updates the current network model. The user can insert, delete and edit FBs of different FB types by clicking on the buttons in the tool bar, respectively via using the context menu of an item or by double clicking on an FB (for editing).

Graphical Front-End

Figure 6.51 shows a screenshot of the PSE-Designer application with the known sample $P_{SE}$ from Figure 4.8. All icon elements and toolbox button options are designed to maximize identifiability, allowing for a fast composition of the network in a drag and drop fashion. Graphical FBs can be connected by drawing directed links between them.
User interaction is limited to those actions that guide the user towards correct network assembly. Based on the model data, the GUI only permits actions which are meaningful within a specific scene and consistent with the model. For example, it is impossible to draw an arrow towards an FB output, as this would be in defiance of the underlying logic, specifically with Definition 20 in this case. All FBs and FB links have a logical representation within the model where their current network status is kept. Decisions which affect the logic of the network status are taken by the model manager alone for a clean implementation scheme.

Parameter Mapping

A built-in editor displays control dialogs that assist the user with the correct mapping of resources and parameters for EEOP and fact blocks. The list of available choices for EEOP blocks is dynamically generated during network assembly, so that correct parameter (re-)allocation is supported and guided by the user interface.

Summary on PSE-Designer

Compared to the previously existing PSE-programming method based on manual setup of Petri nets (see e.g. Figure E.1) the function block based approach is much more ergonomic and intuitive. First of all, function block networks have a considerably improved readability compared to Petri nets. Second, using function block networks is the first step towards realization of an Automation by Configuration approach, which will be pursued further on in the FRIEND projects: Programming of a robotic system will be replaced by configuration of the system. This principle will be applied on more and more programming levels in the future.
The previous manual Petri net programming approach required a cyclic and iterative programming-verification-programming procedure. An external self-made tool provided 19 different tests for the correctness of the PS\(E\)s and their conformity to the task representing elements as specified in the task knowledge database [Kampe, 2007]. With the PSE-Designer, most of the verification routines have become obsolete since the PS\(E\)s are constructed correctly and according to the database prototype specifications from the first moment on, as the tool prohibits wrong user input. Remaining required verifications are integral component of the PSE-Designer and are executed automatically during the PSE-configuration process.

### 6.2.5. Conversion of Function Block Networks to Petri nets

After specification of a PS\(E\) on function block level (as e.g. given in Figure 6.51) and successful completion of the previously discussed validity tests of the function block network, the conversion into a special Petri net (as e.g. given in the Figure E.1 or in a simplified version in Figure 2.10) takes place. To summarize the applied conversion method, the function block network elements and their counterparts within the representation as Petri net are presented in Table 6.2. More detailed explanations of the conversion of the different elements follow in the subsections 6.2.5 to 6.2.5.

For the purpose of conversion, the class \texttt{CFBN2PNConverter} has been implemented; this class and its relations are depicted in Figure 6.52. A properly initialized instantiation of the converter class holds a pointer \texttt{m_pFunctionBlockNetwork} to a valid function block network as well as a pointer \texttt{m_pPetriNet} to the Petri net to be generated during the conversion process. Additionally, the complete set of function
blocks that are contained with the FBN is stored in \( m_{.pvFunctionBlocks} \) for facilitated access during conversion.

The class \texttt{CFBN2PNConverter} is equipped with several helper methods that will be used and explained in detail alongside the subsequent explanation of the complete conversion process. The entry point of conversion is the \texttt{Convert} method. Utilizing the model driven development approach, to tackle the complexity of the underlying algorithm, the \texttt{Convert} method is modeled entirely with a hierarchical flowchart which is automatically converted into executable C++ code. The decisive advantage of modeling and implementing an algorithm via flowcharts is the reduced complexity on different levels of the flowchart representation. Furthermore, to adhere to the rules of structured programming with flowcharts as introduced by Nassi/Shneiderman [Nassi and Shneiderman, 1973][14], a strict modularization is enforced.

The flowchart of the complete function block network to Petri net (FBN2PN) conversion algorithm is depicted in Figure 6.53. In the following, this algorithm is discussed within corresponding sub-sections as depicted in the overview flowchart. Some formatting conventions are introduced here:

---

[14] Nassi-Shneiderman diagrams (NSD) are also known as structograms and are standardized e.g. according to the German DIN 66261.

**Definition 22** The following formatting conventions are introduced for flowcharts in the FRIEND::Process:

- **Bold actions** (e.g. the Init action) internally specify C++ code
- **Bold actions with a sub-chart symbol at the right corner of the action box** (e.g. GenerateStart) are modeled via sub-flowcharts
- **All other actions** (normal font weight) directly display C++ code

For the illustration of the algorithm’s discussion, it is recommended to consider Figure 6.51 as sample function block network and Figure E.1 as the resulting Petri net.

**Preparation**

During the initialization stage of the algorithm (Init action), various locally used variables are initialized, e.g. the vector `m_pvFunctionBlocks` of FBN function blocks (`m_pFunctionBlockNetwork`) providing convenient access to the blocks to be processed. Subsequently, the Start Block is determined and a respective place labeled Start is added to the PN. In case of parallel branches leaving the Start Block, an additional splitting transition has to originate from the Start place which is connected to a Start_i as well as a StartEnable_i place. The index i is the branch number starting from i = 1 representing the first branch (see Figure E.1). The necessity of the additional StartEnable places will be explained in the Finalization part of the algorithm (subsection 6.2.5). The place for the Target Block of the FBN is created in the Petri net to link to its associated elements. If necessary, as in the case of an Or Block as an input, additional input transitions are created and linked accordingly. Similar measures have to be taken into account with regard to the Abort Block.

**Generation of EEOPs**

As depicted in Figure 6.53, the conversion algorithm treats EEOP, Fact and OR Blocks in a similar manner regarding the generation of their Petri net counterparts. The algorithm loops over the list of all function blocks to find and subsequently process the next block of a certain function block type.

Figure 6.54 shows a representative example of a sub-flowchart from the complete hierarchical flowchart model. As introduced in Section 2.4.2, an EEOP is represented within the Petri net with a tuple of transitions, where the number of transitions is determined by the number of possible EEOP return values. Therefore, these return values are retrieved within the action GetEeopReturnValue and stored in the set `pvReturnValues`. Subsequently, a transition is added to the Petri net for each return value (`BuildTransitionPerReturnValue`). The connection of these transitions has to comply with the basic construction rules for Petri nets, i.e. that a transition always has to be connected to one or more places and vice versa. Thus, depending on the Petri net representation of the converted FBN elements according to Table 6.2, the inclusion of Intermediate State Places (labeled ISP.<UniqueName>) or Intermediate Transitions (labeled IT.<UniqueName>, see Figure E.1 for ISP and IT examples) may

---

15 Figure 2.10 can also be used as an example of a resulting Petri net, but due to the graphical simplification in this diagram it does not contain all elements.
Figure 6.53. Flowchart of the complete conversion algorithm.
become necessary. These rules are applied during the connection of the outputs in \textit{ConnectFBOutputs} as well as the inputs in \textit{ConnectFBInputs}. These two actions include the linking of places that model the resource availability. The final category of linkings is established within \textit{ConnectParameterData} which connects the places for data flow modeling.

**Generation of Facts**

As specified in Table 6.2, \textit{Fact Blocks} are converted to places in the Petri net. For each boolean state of a fact, a separate place is provided. However, only the states that are used within the \textit{PS}_E are included. For example, in case the "FALSE" state is neither read nor set, it is omitted in the Petri net. Figure 6.53 shows two conversion steps for fact generation, namely \textit{GenerateFactInput("SetFALSE")} and \textit{GenerateFactInput("SetTRUE")}. The reading of a fact’s state does not have to be considered in this part of the conversion algorithm, since the linking from the fact place to a transition using this fact is covered during the generation of the respective FBN entity.

Within the \textit{GenerateFactInput} function it is first checked, whether the block’s input port for the given boolean state is used. If so, the input helper transition (\textit{IT.SetFactTrue}) is added to the Petri net and linked from both sides (i.e. the link from the FBN entity setting the fact as well as the link between helper transition and fact place is established). Figure 6.55(a) gives an example for fact-representing elements in the Petri net.

Finalization

The elements of the FBN that remain to be converted are the OR Blocks. Figure 6.55(b) and Table 6.2 point out that AND Blocks are not explicitly modeled, neither as place nor as transition. They are rather indirectly modeled by additional ingoing links during the conversion of the FBN element they link to. In Figure 6.55(b) for example, the successful execution of a preceding EEOP is linked by an AND Block to the system fact representing the open gripper of the manipulator.

As Figure 6.55(c) illustrates, Or Blocks are converted to (intermediate state) places which the incoming blocks or helper elements are linked to.

Finally, to finish the conversion process, a special procedure has to be taken into account for parallel branches leaving the Start Block. The problem is that when the Target Block is reached, it is basically possible that one of the branches could not yet have been activated, in the sense that an EEOP has not yet been executed, e.g. due to a fact state that never became true during a certain execution procedure. However, whenever the target of a PSE is reached, no execution can be allowed afterwards. Therefore, the conversion algorithms (HandleParallelStarts in Figure 6.53) searches for transitions preceding the Target place and connects them to the Start_Enable_i places (see Section 6.2.5 and Figure E.1), this way disabling any parallel branch leaving the Start Block.

6.3. Summary

The process steps described in this chapter are concerned with task knowledge engineering based on process-structures. The input for these steps are the development artifacts from scenario analysis, i.e. the task participating objects (TPO) as well as the sub-scenario use-cases. To achieve a simple programming procedure suitable for application in practice, several integrated tools for task knowledge specification and programming with process-structures have been developed.

In Step 2a the abstract process-structures (PS_A) are created. The complete procedure for this is shown in Figure 6.2 (page 75). First, the formal specifications of TPOs have to be entered into the task knowledge database. The DBGUI tool provides a user-programmer-friendly interface for the introduction of new and editing of existing task representing elements (TRE). Subsequently, the configuration of object constellations (OCs) can take place. One of the central improvements introduced in the scope of the FRIEND::Process is the pictographic synthesis of object constellations. With respect to the desired Automation by Configuration (AbC\(^{16}\)) paradigm, a system-controlled and constraint-based configuration of 3D-OCs takes place. After creation of OCs, they are connected with arcs that represent abstract level operators, so-called composed operators (COPs). The pre- and post-condition facts that are assigned to the COPs are introduced in the connected OCs and are also propagated throughout the complete network in a logical consistent manner. Illogical PS_A-configurations (e.g. the manipulator grasping a table) are rejected immediately. All the methods required for pictographic configuration of PS_A are embedded in the PSA-Designer tool, which decisively increases the comfort of creating PS_A and makes the accessibility of this tool possible by non-technical care personal, or even the impaired user himself.

\(^{16}\)see Definition 5 on page 58 or Glossary

130
Step 2b is dedicated to the development of elementary process-structures ($PSE$), which are the decomposition of a COP from $PS_A$-level. In the FRIEND::Process, the previously used manual composition of Petri net based $PSE$s is replaced by a function block network (FBN) based approach. This enables user-friendly and system-controlled configuration of $PSE$s, again in the line with the AbC-paradigm. The configuration methods, including verification mechanisms and synchronization with the task knowledge database, are embedded in the PSE-Designer application. This tool completes the $PSE$ programming with a conversion of the FBN into a Petri net. Within the Petri net, verification routines are performed that are not done on FBN-level, like the test of reachability of targets and the exclusion of deadlocks. Also, the Petri nets serve as input for the task planner of the FRIEND::Architecture. The methods proposed here do not only achieve a decisive increase of the programming comfort (configuration instead of programming), but they also decrease the required task knowledge engineering time tremendously. By building the $PSE$s immediately in the right way, the time-consumption for the construction of one $PSE$ is reduced from hours to 10-15 minutes, as it will be shown in Chapter 9, Section 9.4.
Within this chapter, the important transition step from skill specifications as determined and verified throughout the preceding development step to the actual implementation of a skill is achieved. The uniformity of skill interfaces and implementation allows to guide the skill programmer with clear design and implementation instructions with the help of a process model. This will lead to the fulfillment of Process Requirement 9. After finishing the development Step 3a - i.e. the skill design - the skill testing as discussed in development Step 4a can be performed immediately. Alternatively, the actual skill implementation, which is subject of Step 3b, takes place. Iterative enhancement of the skill implementation and repeated intermediate testing is performed in a cyclic manner without strict prescription of the order of development steps. This is also reflected in Figure 7.1.

### 7.1. Step [3a]: Skill Design

<table>
<thead>
<tr>
<th>Input artifacts:</th>
<th>Output artifacts:</th>
<th>Tools:</th>
<th>Repositories:</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEOPs</td>
<td>Skill-Use-Cases, -Requirements, -Skeleton+Behavior, Skill Test Sequences, World Model Test Data</td>
<td>DBGUI, Rhapsody</td>
<td>Task Knowledge Database, Test Data Repository, Model Repository</td>
</tr>
</tbody>
</table>

A2.7
A3.2–7
T2+1
R2+4+1
7. The FRIEND::Process: Skill Development

7.1.1. Definition of Products

The flow of products and the interactions with the different repositories throughout the development step *Skill Design* is shown in Figure 7.1. A formal representation is given in the same manner as for the previous development steps in Figure 7.2.

![Figure 7.1. Process Step 3a: Skill Design.](image)

**Figure 7.1.** Process Step 3a: Skill Design.

**Figure 7.2.** Development artifacts of skill design procedure. (Input artifacts are gray, output artifacts are blue.)

The output of the previous and the input of this development step are EEOPs. This artifact is marked in gray together with the aggregated artifacts, which are: Resource items, sub-symbolic parameters and the possible return values.
The new product which is produced by taking the EEOP specification into account, is a **Skill Skeleton**. As shown, a skill skeleton has a set of **Use Cases**, which serve for grouping skill **Requirements**. The **Skeleton Behavior** contains the basic infrastructure of a skill, which consists of parts that are common to all skills, or parts that are derived from the skill specification respectively. The skeleton behavior is either modeled with the help of a skeleton activity diagram (AD\_skel), a skeleton flowchart (FC\_skel), a skeleton sequence diagram (SD\_skel) or a skeleton statechart (SC\_skel). Also, a combination of behavioral specifications is possible.

### 7.1.2. Use Case / Skill Requirements Specification

For each skill a use case diagram is created in the model repository. An example of such a skill use case diagram is given in Figure 7.3 for the skill *CoarseApproachToObjectInContainer*. The purpose of this skill is to grasp an object with the gripper in a container-like place, e.g. in a fridge, microwave oven or cupboard. The figure also shows the organization of the skill requirements within the hierarchical model tree structure.

**Figure 7.3.** Skill use cases and requirements for the skill *CoarseApproachToObjectInContainer*.

#### Skill Use Cases

There are two kinds of skill requirements:

- Basic skill requirements
- Skill specific requirements
Basic skill requirements are applicable to skills in general. Basic skill requirements are e.g. the following ones:

- **Stop and kill a skill:** It must be possible to stop a skill any time. If a skill receives a stop signal, it will stop currently running processes, like movements of the manipulator. Besides stopping a skill, it also has to receive *kill* signals. This means, the skill directly stops and quits its own execution thread, without influencing child threads.

- **Specific skill simulation level behavior:** The different skill simulation/execution levels as listed in Section 4.5.1 have to be taken into account within the skill implementation. The minimum number of levels that have to be respected are the *probabilistic* and the *execution level*.

- **Correct skill signature:** The signature of a skill is determined through EEOP specification. The first skill parameters list the name of resources to be used within a skill. Further on, parameter identifier for sub-symbolic object data follow. The last skill parameter is the *Call-Back* object parameter that is required - due to the asynchronous nature of skill calls - for communication with the calling scope (i.e. the Sequencer).

- **Correct world model access:** The world model has to be accessed correctly within the skill implementation. This means, world model data is either read or written according to the skill parameter specification, or temporary data is used, e.g. when helper skills are called from within a skill. Temporary data always has to be deleted at the end of skill execution.

- **Correct sending of call-back message:** Whenever a skill terminates, either in successful or erroneous case, it has to send the correct call-back message to the calling scope (Sequencer). It is only allowed to send a value from the list of call-back message that are assigned as possible skill return values.

The skill-specific use cases are usually related to the object classes that are considered by the skill implementation. For the example given in the use-case diagram in Figure 7.3, i.e. the skill *GraspObjectInContainer*, there is the use-case *FetchBottleFromFridge*, specifying that a *large container* and *cylindrical* objects are involved as object classes. The use-cases, i.e. the different object classes, are also later used as input stimuli for skill testing.

**Actors**

The actors in the skill use cases denote the involved resources and the possible object classes. In the given case the resource is the *robotarm*. The object classes for the item to grasp are: cylindrical objects, composed objects and flat cuboid objects. Object class for the container parameter are: large container and small container.

### 7.1.3. Skill Skeleton Behavior

Skills are methods, which are similar in their structure, i.e. they are all build up according to a specific infrastructural pattern. The basic skill skeleton contains the following parts:
7.1. Step [3a]: Skill Design

- Skeleton-internal declarations
- Set up proxy connections to other skill servers or to data servers
- Extract world model data
- Skill core implementation, containing the core behavior
- Write back data to the world model

All these sub-actions are embedded into a flowchart. Flowcharts inherently enforce modularization, structuring and documentation, partly through the nature of a flowchart diagram itself, but also due to the fact that the rules of Nassi/Shneiderman [Nassi and Shneiderman, 1973] as first introduced in Section 6.2.5 have to be respected to obtain structured code in the target programming language after generation. The FRIEND::Process claims the flowchart-based behavioral specification of a skill as mandatory. The other types of behavioral modeling means (activity diagram, sequence diagram, statechart) are optional and serve either as documentation support or supplementary specification of skill behavior.

The skill skeleton flowchart is built up hierarchically with at least 3 layers. The first layer contains the declaration part, the second layer represents the main skill infrastructure, containing the leaves of the hierarchy in the third layer or sub-flowcharts branching into further layers if required.

Before discussing the different parts of the skill skeleton flowchart, the parameter manager is introduced as an important helper module to enforce a proper and maintainable skill implementation. This concept of a parameter manager also prepares the enhancements of the FRIEND::Process with automatic skill verification methods. These enhancements, which are outside the scope of this thesis, can be realized based on the unification of the skill design and implementation that is enforced with the help of this manager.

Skill Parameter Management

Each skill skeleton has its skill parameter manager, where the skill parameters are managed. The required data structures are specified in the diagram in Figure 7.4.

The skill parameters encapsulate different sub-symbolic data types, e.g. an object location or an object size. To handle all parameters analogously from the parameter manager’s viewpoint, the parameter class is equipped with several template methods and also uses template helper classes. These helper classes provide template methods to extract data from the world model, to insert it, or to delete it. For example, to extract data from the world model, the `getData()` method is called from the parameter manager as follows:

```plaintext
ContainerLocation = ParameterManager[1] -> getData<SubSymbolicLayer::CLocation>();
ContainerSize = ParameterManager[2] -> getData<SubSymbolicLayer::CSizeCuboid>();
ObjectLocation = ParameterManager[3] -> getData<SubSymbolicLayer::CLocation>();
```
During data extraction, the skill parameter class checks, whether the parameter is used according to its specification (IN, OUT, INOUT type, see Definition 17, page 94) and marks the parameter as used, to check for unused parameters at runtime.

Further provided methods in the skill parameter and manager class are to retrieve certain parameter characteristics, e.g. whether a parameter describes an elementary body like cylinder or cuboid or is a more complex composed object. Also, parameter characteristics can be retrieved with the help of the parameter manager from the world model. The connection to the world model is maintained globally for the whole skill parameter class with the help of a static association.

**Skill Declarations**

Each skill skeleton behavior is equipped with a declaration part in the top-level flowchart. Here, all important aspects with respect to the skill specification are shown. An example is given in Figure 7.5. This top-level part can be seen as a cover sheet for the skill documentation.

For easy access of information at one glance, the name of the skill is shown with the help of a comment box at the top (here: CoarseApproachToObjectInContainer). Next, all declarations with respect to the skill interface specification are formally introduced. These are the skill-internally used name string of the skill method, as well as the skill parameters. The latter ones are introduced in the parameter manager with the help of the addParameter() method, which needs, besides the parameter itself, the parameter’s type specification (IN, OUT, INOUT or RES).

Further declarations are the resources to be used in the skill, like the robot arm hardware server in the given case. Also, the MVR data server is used, and CORBA proxies for skill-internal connections are required for connecting the mirror skill server for executing helper-skills, which makes the further declaration of a call-back servant and proxy necessary.

Eventually, the allowed incoming set of skill messages as well as the allowed skill return values are specified with the help of vector variables, to be able to track later, whether a certain message is allowed according to the skill specification as gathered here at the top-level.
Main Infrastructure of a Skill

Subsequent to the declarative part, the main skill part follows. This part contains the main infrastructure of a skill as shown in Figure 7.6.

Besides the formatting conventions that have been introduced for flowcharts in Section 6.2.5, Definition 22, additional conventions are introduced here:

**Definition 23** The following conventions for skill flowcharts are defined:

- **Green action, normal font weight**: Universal implementation is directly given as code in the target language, here C++.
- **Green action, bold font weight**: Universal action is contained here, but the action has a summarizing label and the code is hidden.
- **White action**: Skill-specific implementations are hidden via sub-flowchart.

The typical skill infrastructure embeds the core execution part of the skill in start and finishing actions, which means the start and finishing of a skill is logged and internal exception handling takes place. Between catching internal exceptions and logging the skill termination, the skill’s return value is sent via call-back to the calling scope (Sequencer). Furthermore, the probabilistic simulation of a skill is handled uniformly in each skill. One side branch in parallel to the main execution path is foreseen, which calls...
7. The FRIEND::Process: Skill Development

Figure 7.6. Main infrastructure flowchart of a skill skeleton.
a skill server base class method for probabilistic skill simulation. This means that one of the possible skill return values is determined, based on given probabilities of all return values as specified in the TRE database (see also Section 6.1.5). All these actions mentioned so far are universal for all skills. The remaining actions, as exemplarily detailed in the following, are skill-specific.

**Check and Prepare Connections**

Special template methods are provided to acquire valid CORBA proxy objects. These methods have to be called before using the proxies. A sample sub-flowchart for setting up the connections is given in Figure 7.7. First, the proxy for the robot arm hardware server, the resource to be used in the sample skill, is prepared. Afterwards, it is possible to either set the resource to operative state, if a real execution of the skill is desired, or to set the resource to simulative mode in case the skill is simulated only. Further on, all remaining proxies and servers for local connections are prepared. These are the mirror skill server, the MVR server, the world model server and the call-back servant for helper skill calls on the mirror skill server or on other skill servers.

![CheckNPrepareConnections](image)

**Figure 7.7.** Example for preparation of proxy objects.

**Extraction of World Model Data**

As soon as the proxy objects are prepared, the sub-symbolic data that has to be processed within the skill can be extracted from the world model. The flowchart in Figure 7.8 shows how this is done.

First, the parameter manager initializes all parameters. This means, all available information about the parameters is extracted, like *object characteristics*, *object modeling state* and *object class*. In the next step, basic parameter-specific data is extracted. Here, parameters are meant that encapsulate sub-symbolic data types that are independent of the skill use case. Example code for this quasi static types is given in the code listing above. In this example, the container’s location and size data and the object’s location are extracted, since these data types always remain the same in this skill. After this basic extraction, the data for dynamic types is retrieved. Here, the object’s shape characteristic...
is requested from the parameter manager, and depending on this, the shape description for a cylinder or cuboid is extracted, or an internal exception is thrown, since in this example no other types are supported.

**Skill Core Implementation Frame**

If the skill is run in execution level, the skill’s core implementation frame is executed next. During design of the skill skeleton, no special implementation is mandatorily included here. This is usually postponed to the successive enhancement of the core implementation as exemplified in Section 7.2.1.

**Set and Clear World Model Data**

Within this sub-action, the following tasks are carried out:

- Write OUT-data to world model
- Delete temporary data from world model
- Check parameter usage

The first action is skill-specific. The data that is specified as OUT or INOUT type is written to the world model with the help of the skill parameter’s `setData()` template method. The two subsequent actions are universal ones for skills in general. Here, the parameter manager’s methods are used to erase all temporary world model data that might have been stored in the world model, e.g. due to usage of helper skills. Eventually, the parameter manager iterates over all parameters and performs a runtime check, whether the parameters have been used according to their specification. If this is not the case, a warning is emitted and protocoled.
Handle Incoming Skill Message

Within the execution of a skill, the handling of potentially incoming skill messages is required. In general, any messages can be sent to the skill from the calling scope, e.g., in order to re-parameterize a skill. However, the standard skill messages that are sent are STOP and KILL messages, as also discussed in Section 7.1.2 along with the basic requirements for a skill. The processing of these messages has to be included in each time-consuming loop within a skill, to make the interruption of the ongoing execution possible at all times. Only for reasons of exemplification it is included as last action before finishing the execution of the skill here.

Figure 7.9 shows the handling procedure for the default skill messages: First, the call-back object is requested for the arriving of a skill message update. If so, it is checked whether the message is in the set of allowed messages as set up within the skill skeleton declaration part (Section 7.1.3). If this is also the case, an internal exception is thrown to signalize the successful termination in this simplified example here.

Figure 7.9. Flowchart to handle standard incoming skill messages STOP and KILL.

Herewith, the minimum required part of skill skeleton specification is finished. Optionally, for documentation purposes, another form of behavioral specification is presented in the following, i.e., the description via sequence diagram.

Skill Design with Sequence Diagram

Even though a sequence diagram can model alternative or optional branches of execution and can also be set up in a hierarchical manner, it does not (yet) provide a behavioral specification that can serve as
 executable model in Rhapsody, as a flowchart can. The top level sequence diagram, which models the method executions in the skill `CoarseApproachToObjectInContainer` is given in Figure 7.10.

![Sequence Diagram](image)

**Figure 7.10.** Top level sequence diagram for the skill `CoarseApproachToObjectInContainer`. Shows optional and alternative branches as well as referenced subsequence diagrams.

The skill calling scope, which is usually the Sequencer, here denoted as `ENV` for environment, first calls the skill server base class method `InitConnections(SystemResources)` to (re-)establish the connections in the CORBA network related to the skill server. Subsequently, the skill simulation/execution level is set, before starting the skill itself.
All further actions are exactly those of the previous skill skeleton flowchart. Alternatives are specified with a box that separates the alternatives and contains the label \textit{alt} at the left top corner. The boolean condition for the alternative is specified at the top of each branch within squared brackets. Optional actions are introduced analogously with the label \textit{opt}. Referenced sub-sequence diagrams contain the label \textit{Ref}. Two interesting sub-sequence diagrams are shown in more detail in the following.

First, Figure 7.11 shows the methods that are called in the referenced diagram \textit{CheckNPrepareConnections}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure7.11.png}
\caption{Sub-sequence diagram for the action \textit{CheckNPrepareConnections}.}
\end{figure}

The template methods $\text{Convert()}$ of the classes $\text{CManageHardwareServer}$, $\text{CManageSkillServer}$ and $\text{CManageDataServer}$ are used to prepare the CORBA proxies with the help of local and concrete proxy pointers. A conversion has to take place from base class objects that resulted from the preparations when the \textit{InitConnection()} method has been called initially as mentioned before.

Further on, the robot arm hardware server is either set to simulative or to the operative state with the help of the \textit{ManageState()} method. The initialization status of the world model server proxy is determined via $\text{CORBA::is\_nil()}$ request. And if tested as valid, the static association of the skill parameter class is initialized.

The other depicted sub-sequence diagram in Figure 7.12 shows how to model loops with the help of sequence diagrams. It is iterated over the set of parameters, the data of out-typed parameters is written to the world model and the parameters are marked as \textit{used} according to their specification.
7. The FRIEND::Process: Skill Development

```
foreach(SkillParameter)

loop

[SkillHasOutParameters]

foreach(OutParameter)

opt

[SkillHasOutParameters]

foreach(OutParameter)

:CManipulatorSkillServer_impl

:SkillParameterManager

:SkillParameter

deleteTempDataFromWM()

letData(Data)

The parameters are only marked as "used" when they are used according to their specification (IN, OUT, INOUT, RESOURSE, TEMP)
```

Figure 7.12. Sub-sequence diagram for the action SetAndClearWorldModelData.

7.1.4. Setup of Test Data

As soon as the design of a skill is complete, it is possible to execute and thus use it within testing. Therefore, it can be included in the various test sequences that are stored in the test data repository and used according to the discussions in Section 8.1.

If the sequence of skills needs world model data for the simulation of a skill, this has to be configured in the test data repository, too. For example, if a vision skill is not yet implemented and the recognition of an object to be grasped is not yet possible, the simulative execution of the manipulative skill can be tested with the help of world model test data. For this purpose, a configuration of test data according to the different skill use cases takes place. A more detailed discussion about the nature of this test data is also done below in Section 8.1.

7.1.5. PSA Sample Skill Sequence Diagrams

In order to put the execution context of one skill into a broader frame, sample skill sequence diagrams are used to illustrate a possible sequence of skills that are contained in one abstract process-structure. The setup of this sequence diagram is done manually in the UML repository (i.e. in Rhapsody). But, since the skill interfaces do exist in the model after skill design, the method calls that represent skill executions do not have to be entered manually in the sense of text input, but can be selected as existing model elements during the setup of the sequence diagram.

A sample sequence diagram that contains the skill CoarseApproachToObjectInContainer in the scope of the PSA Fetch meal is shown in Figure 7.13. This PSA is analogous to the Fetch cup scenario from Figure 4.6, page 54.

The diagram shows the involved skill servers for User Interaction, Machine Vision, Manipulations, the Intelligent Tray, Calculations and the Smart Fridge component; as well as the additionally involved components Sequencer and World Model Server.

First, the Sequencer loads scenario relevant knowledge into the world model. Subsequently, the initial monitoring process is started to determine the initial situation within the given process-structure.
7.1. Step [3a]: Skill Design

General remark: The described sequence of skills is only a sample sequence. The Function Block Diagrams specify the control flow for alternative operation sequences. (COP names, e.g. MonHoldsNothing (Manipulator), in yellow comments, are linked to the respective Function Block Diagram)

In case a fast interaction device is used (as e.g. chin control) and under the prerequisite of successful coarse approach (this information is retrieved in the HMI via HMI context given from the sequencer) this skill returns immediately with "SystemTakeOver" to initiate the autonomous fine approach.

Figure 7.13. Sequence diagram with possible skill sequence for fetching a meal from a fridge.
about the initial monitoring and the involved object anchoring are outside the scope of this work but are discussed in [Prenzel, 2003, Prenzel, 2005]. However, the result of monitoring as perceived from a certain skill server is the activation of monitoring skills for the purpose of determining the states of relevant facts; like the determination of the fact HoldsNothing(Manipulator) with the monitoring skill MonHoldsNothing(SCam, Robotarm).

After the successful determination of the initial situation, the task planning starts and the sequence of skills as given in the second compartment of the sequence diagram may result. This part shows the skills in groups of the COPs as they are first planned on the abstract level, before decomposition to EEOP/skill level takes place.

In the given example, the fridge is equipped with an automatic door. After opening that one, the skills of the COP GraspObjectInContainer are executed, followed by those ones of GetObjectOutside, PutDownObject (on the wheelchair tray), Depart from grasped and placed meal tray and finally, the COP CloseAutomaticDoor is executed.

7.1.6. Future Enhancements of Skill Design Methods

Figure 7.14 gives an overview on the tools and repositories that are used throughout the skill design. It also shows an element that has not yet been included in the discussions: the Rhapsody API.

![Figure 7.14. Tools for skill design, including future enhancements with Rhapsody API.](image)

In future enhancements of the FRIEND::Process, the Rhapsody API will be used to automate different verification and generation steps for skill design that, at the moment still have to be conducted manually. With the help of the API it is possible to access the model repository and to read and write model
elements. Thus, the introduction of a new EEOP in either the DBGUI or the PSE-Designer can trigger the verification of the model repository to check, whether the respected skill already exists and matches the specification. This is not limited to the operational contract of a skill method, but may also include the behavioral specifications, since they are forced to be uniform with respect to certain aspects due to the usage of the helper classes `skill parameter manager` and `skill parameter`. For example, the code lines that are included in the skill skeleton declaration (Section 7.1.3) are pre-determined through the EEOP specification already.
7.2. Step [3b]: Skill Implementation

![Diagram](image)

**Figure 7.15.** Process Step 3b: Skill Implementation.

<table>
<thead>
<tr>
<th>Input artifacts:</th>
<th>Skill-Use-Cases, -Requirements, -Skeleton+Behavior</th>
<th>A3.1–4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output artifacts:</td>
<td>Complete Skill Behavior</td>
<td>A3.8</td>
</tr>
<tr>
<td>Tools:</td>
<td>Rhapsody</td>
<td>T1</td>
</tr>
<tr>
<td>Repositories:</td>
<td>Model Repository</td>
<td>R1</td>
</tr>
</tbody>
</table>

#### 7.2.1. Successive Enhancement of a Skill’s Core Implementation Frame

As already depicted in Figure 7.15 and mentioned above, the skill implementation step completes the skill skeleton behavior to the full skill implementation. This means the skeleton flowchart is enhanced, the enhancements are tested, and the cycle is restarted until all requirements are matched.

Enhancing the skeleton mainly means to implement the behavior of the main part. To maintain the structured approach, several hierarchy levels within the flowchart enhancements will be necessary. A summarized overview of further actions to be implemented for the sample skill *CoarseApproachToObjectInContainer* is depicted in Figure 7.16.

The approaching of an object is split, as the skill’s name already points out, into a coarse and a fine approach. Therefore, the motion planner first determines a plan to the pre-location and the plan to reach the goal location is established in the second step. If the planning to the goal location results in success, the robot arm is moved to the pre-location.

#### 7.3. Summary

The methods discussed in this chapter introduce a consequent *model-centric* (instead of code-centric) approach for the development of basic system functionalities, the skills. This enforces a consistency during the transition from elementary task knowledge specification level to actual implementation of skills. The methods set up on the fact that the skills impose uniform interfaces and infrastructure so that the guiding of the skill programmer by a process model is possible.
Figure 7.16. First level of the main implementation part of the sample skill \textit{CoarseApproachToObjectInContainer}. 
Step 3a: The initial setup of skill use-cases structures the requirements for skill development according to object classes that have to be accounted for in the skill implementation. The modeling of the skill behavior via hierarchical flowchart automatically enforces a clean documentation, structuring and modularization scheme for the skill. Different formatting conventions give at one glance the overview on the important skeleton-internal declarations and the main infrastructure of a skill. The first is important to compare the skill design with the specifications from the previous development step, namely the EEOP specification. The second, i.e. the compact overview on the complete main infrastructure, serves as standard template for the setup of new skill designs via simple copy and paste from existing skills. Finally, the white action blocks of the main infrastructure flowchart allow an easily visible access to the skill-specific implementations that have to be adapted to a certain skill implementation. The advantage of this clear scheme is, beside other aspects, important for novice developers, to focus on the important aspects for skill development and not to get lost in the complexity of the complete framework.

The pre-structured flowchart templates consist of the following elements: One section contains the main declarations, which reflect the adhering of a certain skill to its specification from $PS_E$ level. The next section models the skill’s main infrastructure, including an exception handling infrastructure, a sub-flowchart which connects other required CORBA servers, and another sub-flowchart for data extraction and writing back to the world model. The skill’s core implementation is separated within a sub-flowchart, also. Furthermore, the proposed skill design approach includes helper constructs, which enforce the consistency to the task knowledge specification ($PS_E$) level. This is e.g. the parameter and parameter manager class, which organizes a runtime supervision of the correctness of parameter usage with respect to the skill specification.

After finishing the construction of a skill skeleton, immediate testing of its behavior is possible. The skill test sequences and skill test data are derived from skill use cases directly. During testing, the full features of model-driven development are used, like online execution visualization with animated sequence diagrams.

Step 3b: The skill skeleton behavior is enhanced iteratively, until the complete behavior is realized. The development cycles switch between implementation and testing; the latter one is discussed in the following chapter.
The FRIEND::Process: Skill and Scenario Testing

8.1. Step [4a]: Skill Testing

Figure 8.1. Process Step 4a: Skill Testing.

<table>
<thead>
<tr>
<th>Input artifacts:</th>
<th>EEOPs, Skill Test Sequences, World Model Test Data</th>
<th>A2.7, A3.6+7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output artifacts:</td>
<td>Skill Test Sequence Diagrams</td>
<td>A4.1</td>
</tr>
<tr>
<td>Tools:</td>
<td>Skill Tester, World Model Browser, Helper Tools, Rhapsody</td>
<td>T5+6+9+1</td>
</tr>
<tr>
<td>Repositories:</td>
<td>Task Knowledge Database, Skill Test Data Repository, Model Repository</td>
<td>R2+4+1</td>
</tr>
</tbody>
</table>
8. The FRIEND::Process: Skill and Scenario Testing

8.1.1. Tools and Repositories

The testing of a skill means to verify, whether it fulfills the requirements as gathered during skill design and as assigned to the different skill use cases. Within the skill design phase, the skill test sequences and world model test data for simulative executions have been determined and have been stored in the test data repository. However, the setup of the complete skill testing environment is still rather complex, as indicated by Figure 8.2. The current step of the process model will guide through the testing procedure. Central tool of skill testing is the skill tester.

As depicted in the component diagram, showing also the flow of data and control between the entities, the skill tester mainly interacts with the skill servers. Also the data servers are accessed, as e.g. the world model to set test data for simulative skill executions. Furthermore, several helper tools are involved, like the control server and its GUI to start and stop servers, the naming viewer to inspect the current status of the name server which manages the CORBA references of all units in the network [Henning and Vinoski, 1999], and the ConfigGUI for various configuration tasks.

Note: A description of tool development follows. Discussion of the process model continues in Section 8.1.3, page 156.
8.1.2. Development of Skill Tester

Refinement of Requirements

The requirements for the skill tester tool are gathered in Figure 8.3. The list of skills, as determined during skill design, has to be configurable, together with the skill use case specific sets of parameters. After configuration of skill list and parameters, the sequential execution of skills takes place. During execution, it must be possible to stop the ongoing execution. The test profiles, i.e. the sequence of skills, has to be savable and loadable to/from the test data repository. Editing the preferences includes to set the skill simulation/execution level as well as various other configurations that are relevant for skill testing. All these interactions are performed by the programmer, e.g. the programmer of the skill.

![Figure 8.3. Requirements for skill tester tool.](image)

Architectural Design

The resulting architectural design is depicted in Figure 8.4. The Tool Requirement 14, which claims to develop one module for skill execution to be used during skill testing as well as for the performance of complete scenarios with the inclusion of task planning, is realized by the class `CEEEOPExecuterBase`.

The usual multiple inheritance approach, typical for Qt-based applications, is realized within the class `CSkillTesterDialog`, which inherits the QDialog base class through the `CSkillTesterBase` and the UIC-generated user interface design class.

Functional aspects of skill testing are located in the skill tester base, e.g. the initialization procedure, the loading of testing profiles as well as the execution of skill lists. The latter is performed through forwarding the execution of one skill to the EEOP executer base, which maps the skill execution to a CORBA call on the respective skill server. For this purpose, the interface `ISkillExecuter` is used, either initiating a CORBA execution via the class `CCORBASkillExecuter` or are mere skill simulation in `CSkillSimulator`. 
The skill tester has several compositions of EEOP Prototypes and EEOP Instances to store the list of available skills as they are retrieved from the TRE database (CDatabaseReadSL) and also to hold the list of skills as they are configured with actual CEEOPParameters for execution.

The inclusion of the base class ACE_Task [Huston et al., 2003] makes the skill tester base to an active class with own execution thread. Thus, the skill tester front-end remains interactive during skill execution to receive stop requests from the user.

**Graphical Front-End**

A screenshot of the graphical front-end of the skill tester application is given in Figure 8.5.

On the top left side the list of available skills as retrieved from the database is offered. The list of selected skills (middle) can either be composed from these available skills, or an existing configuration is loaded from the test data repository. On the top right side the list of parameters for the currently selected skill is displayed. After execution of the skill sequence, the execution results are shown in the output section at the bottom of the tool. The skill simulation/execution level can be configured in the preference section of the skill tester.

**8.1.3. Configuration of World Model Data**

The configuration and loading of world model data, as required for the simulative execution of a skill, is done with the help of the additional tool WorldModelBrowser. A screenshot of this application is provided in Figure 8.6.
8.1. Step [4a]: Skill Testing

After connection to the world model server, the data set can be loaded from the test data repository. In the given case, the data for the test of the skill `CoarseApproachToObjectInContainer` has been loaded. The identifier of sub-symbolic data that are visible are related to the fridge, denoted with `Fr.1`. For each entry in the world model, it is possible to inspect the currently stored data. Here, the details of the location for the right plane of the fridge are currently under inspection.

With the help of the `Add` button new data can be entered. `UpdateMVR` updates the mapped virtual reality server with the current contents of the world model server. This set of data is subsequently respected during motion simulation of the manipulative skill for the coarse approach to the object in the container. A sample skill test based on the process and tools introduced here is discussed in Section 9.6.
Figure 8.6. Screenshot of world model browser.
8.2. Step [4b]: Scenario Testing

![Diagram showing process steps and artifacts]

**Figure 8.7.** Process Step 4b: Scenario Testing.

<table>
<thead>
<tr>
<th>Input artifacts:</th>
<th>Process-Structures A2.1–5+7+8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output artifacts:</td>
<td>Scenario Test Sequence Diagrams A4.2</td>
</tr>
<tr>
<td>Tools:</td>
<td>Sequencer, Helper Tools, Rhapsody T7+8+9+1</td>
</tr>
<tr>
<td>Repositories:</td>
<td>Task Knowledge Database, Process-Structure Repository R2+3+1</td>
</tr>
<tr>
<td></td>
<td>Model Repository</td>
</tr>
</tbody>
</table>

### 8.2.1. Tools and Repositories

The **Scenario Testing** is the highest-level testing that takes place within the FRIEND::Process. It tests the correct system functioning from the viewpoint of a complete scenario and is therefore comparable to integration and system testing.

Figure 8.8 shows the setup and interaction of involved tools, units and repositories. At first sight it is very similar to the setup of skill testing, see Figure 8.2. However, the important difference is that instead of the execution of a predefined test sequence of skills, the task planner (as part of the Sequencer) is involved now, which plans and executes a sequence of skills according to the current situation of environment and system and based on a certain process-structure. The only difference to the later application on the robotic system is that the tasks are activated from the SequencerGUI instead of from the Human-Machine-Interface (HMI). However, from the viewpoint of skill planning and execution there is no difference in task activation due to the usage of the same interface of the Sequencer. Thus, it is eligible to replace the HMI that comes along with its attached input devices and other components with the SequencerGUI, to reduce the setup of system components. The remaining part of the test setup is the same as during skill testing and has been described in Section 8.1.1 already. In the following, a deeper understanding of the procedure of scenario testing shall be achieved with the help of the discussion of the core component used here - the Sequencer.

**Note:** A description of tool development follows. Discussion of the process model continues in Section 8.3, page 165.
Figure 8.8. Tools for scenario testing.
8.2.2. Sequencer - Architectural Large Scale Design

Figure 8.9 provides the large scale design of the Sequencer. The Sequencer is included in the CORBA network via the interface `ISequencerServer`, providing the methods to issue a task request (`EnqueueHighlevelCommand`), to abort a certain running task (`AbortHighlevelCommand`) and to retrieve the current states of enqueued commands via `GetCommandStates`.

![Figure 8.9. Large scale architectural design of the Sequencer.](image)

The `SequencerCore` realizes the implementation of these interface methods and the class `CSequencerServer_impl` unites the CORBA interface and the core class. The core class also composes the And-Or net of the currently activated task, aggregates the `EnvNSys` class for the representation of environment and system and connects the database of task representing elements. Furthermore, the Sequencer core holds aggregations on the `TaskPlanner` and `EEOPExecuter` classes, which are the core classes responsible for planning and execution of a task. Both communicate via a bi-directional association and are realized as active objects [Gamma et al., 1994] with the help of the base class `ACE_Task` [Huston et al., 2003]. This means, they both have an independent service routine which runs in a separate thread and enables them to act independently from each other. These concepts have also been discussed previously in Section 4.4.1 and they are especially illustrated in Figure 4.10 on page 60.

The class `CORBASkillExecuter` organizes the execution of skills in the reactive layer network and is explained in more detail below.

8.2.3. Sequencer - Architectural Design of Task Planner

The architectural design of the task planning modules is given in Figure 8.10.
Within the main service routine of the task planner (svc() method), the method ProcessHighLevelCommand is called upon the reception of a new task request. The monitoring of the initial situation within the given process-structure takes place in DoInitialMonitoring and subsequently the planning on the abstract level of process-structures is performed. This is done in the class PlannerCore with the help of the method CreateHighLevelPlan. The resulting list of COPs is then executed, resulting in the creation of a low level plan for each COP within the method CreateLowLevelPlan in the planning core. The planning on both levels takes place with the help of a Petri net. Therefore, the same basic planning method, namely PlanPassedPetriNet within the planner core is used. The task planner and also the planning core engine access the class that represents the symbolic world model (DynEnvModel) to store and retrieve symbolic information of the system and environment in the form of predicate logic facts. The class PlanRepresent loads the task representing elements from the database and manages the instantiation of these prototype elements according to the task that is currently under planning and execution.

8.2.4. Sequencer - Architectural Design of Skill Executer

Figure 8.11 shows the architectural design of the classes that are involved in the execution of EEOPs/skills. The EEOPExecutorBase class and related classes are already known from Section 8.1.2. They encapsulate the skill execution mechanism within a network of CORBA skill servers and are include in the design of both testing/execution environments, skill tester and Sequencer, in accordance with Tool Requirement 14. Formerly, the detailed discussion of these execution classes did not take place, therefore this is done here in this section.

The EEOPExecutor receives execution requests within the svc() method and with the help of the message queue that is provided in its base class ACE_Task. The method ProcessReceivedEeop is responsible for EEOP processing. That method that is also used within the skill tester for the execution of one EEOP is ExecuteSingleEeop in the EEOPExecutorBase. This class forwards, through the interface ISkillExecutor, the EEOP execution to either a local simulator or the CORBASkillExecutor. The interface claims the existence of an initialization method and subsequently the method InterpretEeop is used to map the EEOP to a skill call. After EEOP/skill activation, the method PollSkillResults checks for finished skill
8.2. Step [4b]: Scenario Testing

executions (skills can be activate in quasi parallel executions). The management of EEOPs to be executed in parallel is done with a queue and the already executed (finished) EEOPs are stored in the set of EEOPInstances.

A more detailed view on the module that maps the EEOP execution requests to skill calls on the different skill servers in given in Figure 8.12.

The CORBA skill executer holds the proxies to all required skill servers for a certain system. With a different system layout, the given design may differ. To manage a unique mapping of asynchronous skill calls and the different call back objects that are used for communication with the skill - either the sending of a skill runtime message or the retrieval of the skill’s return value - a set of call back containers is used. This containers stores the mapping of EEOP and call back and contains both call back communication ends, i.e. the call back servant and the proxy. More details about the call back communication as used here can be found in [Martens et al., 2007].

A typical skill execution with the CORBA skill executer is performed as follows:

- Initialize skill executer
- Start EEOP interpretation
- Initialize simulation and execution data from the database
- (Re-)Initialize the CORBA connections
- Update mapped virtual reality (MVR) if new sensor data is available

Figure 8.11. Overview on architectural design of skill execution modules.
Figure 8.12. Architectural design of CORBA skill executer.
8.3. Summary

- Start skill call on skill server (within the method InterpretEEOP)
- Poll skill results
- Process received EEOP

8.2.5. Sequencer - Behavioral Design

The procedure of task processing, which starts with a task request that arrives at the Sequencer, initiates subsequent task planning and finally the execution of planned operation sequences, is a complex scheme of actions and will not be discussed in full detail here. A simplified documentation of this process is given with the help of sequence diagrams which are explained in the following.

The first part of actions is contained in Figure 8.13. The task reception thread of the Sequencer core has to be activated first, to enable the reception of task requests through the Sequencer’s CORBA interface as introduced in Section 8.2.2. To simplify the discussion here, all CORBA related actions are omitted to point out the main task processing scheme. In the given sequence diagram, a test component, namely CSequencer_main_woCORBA, triggers the task processing with the method EnqueueHighlevelCommand(sTaskFileName). The respective abstract process-structure for the requested task is opened and subsequently the task processing thread is started via activation of the svc() routine. And-Or net and EnvNSys information is extracted from the PS\textsubscript{A} and the initial and target states are set according to default initial and desired target situations in the process-structure. After further system initialization the initial monitoring determines the actual initial situation according to the state of environment and system.

As shown in the sub-sequence diagram in Figure 8.14 the independent processing threads of task planner and skill executer are activated now. Afterwards, the conversion of the PS\textsubscript{A} to a Petri net takes place and planning is performed on abstract level first. The result of this is a sequence of COPs, so that the PS\textsubscript{E} of the first COP can be taken to plan on the elementary level Petri net. The final result in the task planner is a sequence of COP steps, which in fact is nothing else than a sequence of EEOPs. These EEOPs are consecutively enqueued for execution. If (quasi) parallel execution of EEOPs has been planned, they are enqueued in (quasi) parallel, i.e. directly within two successive processing loop steps. The skill executer (embodied in the class CORBASkillExecuter, see Section 8.2.4) maps each received EEOP to a call of a CORBA skill method on the respective skill server, polls for skill results and forward them to the task planner’s message queue. Here, the evaluation of the skill’s return value takes place, i.e. the next planned EEOP is executed if the return value is in accordance with the current plan, otherwise re-planning takes place. In case of a user task abortion request or if the system reaches an irresolvable state, ongoing task execution and further planning is aborted and the Sequencer is ready to receive new task input requests.

8.3. Summary

In this chapter the two different testing procedures of the FRIEND::Process are described, targeting at different scopes of testing: The skill testing, which can be compared to unit testing, as well as the scenario testing, which is equivalent to the testing of the complete system.
8. The FRIEND::Process: Skill and Scenario Testing

Figure 8.13. Task processing in Sequencer (here without CORBA method calls on skill servers), first part.
Figure 8.14. Task processing in the Sequencer (here without CORBA method calls on skill servers), second part.
8. The FRIEND::Process: Skill and Scenario Testing

**Step 4a:** The skill testing is done within the **Skill Tester** tool. A predefined sequence of skills with predefined parameters is executed in different simulation or execution levels. The test setups have been created during skill use-case elaboration to cover all skill requirements. The skill execution can take place in the mode of *probability simulation*, where the correct communication between CORBA servers, the correctness of the CORBA interfaces as well as the basic skill skeleton behavior is tested. The further execution modes are a *skill simulation* of a skill’s core functionality, a *motion simulation* of manipulative skill, a *hardware simulation*, as well as finally the full *execution* of a skill.

**Step 4b:** Here, the complete execution of scenarios with online *task planning* takes place. The task planner performs planning first on the level of abstract process-structures, and subsequently on the level of elementary process-structures. The element within the Sequencer which organizes the activation of skills in the reactive layer of the FRIEND::Architecture is the CCORBASkillExecuter. This module has to be enhanced if new skill servers shall be added to the reactive layer.
Application and Evaluation of the FRIEND::Process in AMaRob Scenarios

‘In theory, practice is simple.’ (Trygve Reenskaug)

The FRIEND::Process is not a process model that has been developed as a pure theory only. It is applied in practice. The first application is in the rehabilitation robotic support scenarios as introduced in Section 2.3. The exemplary application of the process model for the development of one of these scenarios is illustrated in the following. Some resulting development artifacts have already been used throughout the description of the FRIEND::Process’ development itself. They are repeated here to obtain a complete overview on the development procedure.

The informal description of the AMaRob ADL scenario, i.e. preparation of a meal and support for the user to eat the meal - was subject of Section 2.3.1. In the following its stepwise development according to the FRIEND::Process is exemplified.

9.1. Step 1a - Scenario Specification

Sample products of scenario specification are given in Figure 9.1. This example refers to FRIEND III in a kitchen environment and is related to the sketch up as shown in Figure 4.3 (page 51).

As specified in the sample UML diagram, the robotic system consists of the main components Wheelchair, Manipulator, StereoCamera and Tray. Each of these main system items is composed of sub-items, like the Robotarm, the Gripper and a Force Torque Sensor as components of the manipulator. Smart tools that will be used in the kitchen scenario are a Fridge with an automatic door opener as well as a tactile
Figure 9.1. Sample products of scenario specification, here for the kitchen scenario, i.e. preparation of a meal and support for eating.
9.2. Step 1b - Sub-Scenario Specification

The result of specification of sub-scenarios in the ADL scenario has been used previously, namely during the introduction of the AMaRob scenarios in Section 2.3.1. The resulting use case diagram is given again in Figure 9.2, showing the sub-scenarios that are needed for the complete procedure of meal preparation as well as for supporting the user to eat the meal and finally for clearing the tray. Each of the shown sub-scenarios is detailed in Table 9.1. This table also contains the list of TPOs of each sub-scenario.


During development Step 2a, each sub-scenario from the previous step is formally specified with the help of an abstract process-structure $PS_A$. The complete process is described in Figure 6.2 (page 75). For each $PS_A$ the TPO have to be introduced in the task knowledge database first. Some TPOs belong to the robotic system that has to be specified in the database also.
### 9. Application and Evaluation of the FRIEND::Process in AMaRob Scenarios

<table>
<thead>
<tr>
<th>Sub-scenario</th>
<th>Description</th>
<th>TPOs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch meal from fridge</td>
<td>Fetch a meal tray from the fridge and transfer it to a free position in the workspace. The opening of the fridge door takes place automatically but is not included in the scenario since the wheelchair has to be moved after opening and closing. This in turn requires the interruption of automatic task execution and new initial monitoring. Therefore, the user should rather activate the door opener as single operation from the human machine interface.</td>
<td>Fridge Manipulator Meal tray</td>
</tr>
<tr>
<td>Cook meal</td>
<td>Insert a meal on a meal tray into a microwave oven and start the heating process. The user drives the wheelchair in front of the microwave oven. The automatic door opening mechanism of the oven opens the door and the manipulator inserts the meal tray into the oven. The gripper pushes the door to close. Eventually, the heating process is started with meal specific parameters (time and intensity, retrieved from the database).</td>
<td>Manipulator Meal tray Microwave oven</td>
</tr>
<tr>
<td>Fetch meal from oven</td>
<td>Fetch the meal tray from the microwave oven, place it on the wheelchair-tray in front of the user; finally close the oven’s door. The automatic door opener of the microwave oven is actuated to open the door. To close it, the gripper pushes the door.</td>
<td>Manipulator Meal tray Microwave oven Wheelchair-tray</td>
</tr>
<tr>
<td>Take away lid</td>
<td>Lift the lid that covers the plate on the meal tray and place it on some free place on a table. Instead of a table another suitable platform-like place can be used alternatively.</td>
<td>Lid Manipulator Meal tray Table Wheelchair-tray</td>
</tr>
<tr>
<td>Eating support</td>
<td>Food is taken from the meal tray with a spoon. The spoon is transferred near the user’s mouth so that he can take the food from the spoon on his own. This process contains three different use cases: Initially, the spoon that is located at a special resting place on the meal tray has to be grasped and to be lifted. The main cycle of taking food and offering it to the user is repeated according to the user’s desire (user input). Finally, the spoon is inserted into its resting place on the meal tray again.</td>
<td>Manipulator Meal tray Spoon Wheelchair-tray</td>
</tr>
<tr>
<td>Clear tray</td>
<td>Take the meal tray from the wheelchair-tray and transfer it to a free place on a table. Again, another suitable platform-like place may be used to clear away the meal tray.</td>
<td>Manipulator Meal tray Table Wheelchair-tray</td>
</tr>
</tbody>
</table>

**Table 9.1.** Sub-scenario specifications for ADL scenario.

System name: FRIEND-III

<table>
<thead>
<tr>
<th>Components</th>
<th>Abbrev.</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wheelchair</td>
<td>WCh</td>
<td>IsSysObject, IsActor, IsSensor</td>
</tr>
<tr>
<td>WheelchairControl</td>
<td>WChC</td>
<td>IsSysObject, IsActor, IsSensor</td>
</tr>
<tr>
<td>Manipulator</td>
<td>MP</td>
<td>IsSysObject, IsActor</td>
</tr>
<tr>
<td>ForceTorqueSensor</td>
<td>FTS</td>
<td>IsSysObject, IsSensor</td>
</tr>
<tr>
<td>Robotarm</td>
<td>Ro</td>
<td>IsSysObject, IsActor</td>
</tr>
<tr>
<td>Gripper</td>
<td>Gr</td>
<td>IsSysObject, IsActor</td>
</tr>
<tr>
<td>StereoCameraSystem</td>
<td>SCam</td>
<td>IsSysObject, IsSensor</td>
</tr>
<tr>
<td>Bumblebee</td>
<td>BB</td>
<td>IsSysObject, IsSensor</td>
</tr>
<tr>
<td>PanTiltHead</td>
<td>PTH</td>
<td>IsSysObject, IsActor</td>
</tr>
<tr>
<td>Tray</td>
<td>Tr</td>
<td>IsSysObject, IsSensor, IsPlatform</td>
</tr>
<tr>
<td>IRSurface</td>
<td>IRS</td>
<td>IsSysObject, IsSensor, IsPlatform, IsSmartSurface</td>
</tr>
<tr>
<td>Scale</td>
<td>Sc</td>
<td>IsSysObject, IsSensor</td>
</tr>
</tbody>
</table>

Table 9.2. System specification of FRIEND-III with components, sub-components, abbreviations and characteristics. Categorization and relationships of involved objects is given in Figure 9.1.

9.3.1. System Specification

Since the system that is used throughout the application of the FRIEND::Process in this chapter is always FRIEND-III, its specification is given first in Table 9.2. This specification takes the informal TPO specification of development Step 1a (scenario analysis) into account. See also the related diagram in Figure 9.1 which shows the organization and relation of all involved objects.

For each component the component’s name, its abbreviation and its characteristics are given. The underlying rules for assignment of characteristics are discussed in detail in Section 6.1.2. Due to the fact that system components are handled here, the characteristic *IsSysObject* is automatically assigned to all objects according to Definition 7 (page 78). (Also based on the inherent relationship between first characteristic and item type according to this definition, the item type specification is omitted in the table.)

Furthermore, it is distinguished between sensor and actor with the respective characteristics. Additional characteristics result from the fact that a system component is part of the object ontology as introduced in Section 5.1.3. For example the tray is used in the scenarios as a platform and is equipped with a sensor surface. Therefore the tray owns the characteristics *IsPlatform* and *IsSmartSurface*. Each system component manages different hardware resources. These ones are modeled as sub-components of the system’s main components.

9.3.2. Further Task Participating Objects

Analogously to the system specification, the further TPOs in the ADL scenario are formally introduced in the database as given in Table 9.3. Again, the informal identification of TPOs during scenario analysis is the input for this specification process and the diagram in Figure 9.1 shows the organization and relationships of affected TPOs.
9. Application and Evaluation of the FRIEND::Process in AMaRob Scenarios

<table>
<thead>
<tr>
<th>Components</th>
<th>Abbrev.</th>
<th>Type</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fridge</td>
<td>Fr</td>
<td>SYS</td>
<td>IsSysObject, IsEnvObject, IsActor, IsContainer, IsContainer-WithDoorAndShelves</td>
</tr>
<tr>
<td>DoorOpener</td>
<td>DOp</td>
<td>SYS</td>
<td>IsSysObject, IsActor</td>
</tr>
<tr>
<td>MicrowaveOven</td>
<td>Mw</td>
<td>SYS</td>
<td>IsSysObject, IsEnvObject, IsActor, IsContainer, IsContainer-WithDoor</td>
</tr>
<tr>
<td>MicrowaveControl</td>
<td>MwC</td>
<td>SYS</td>
<td>IsSysObject, IsActor, IsSensor</td>
</tr>
<tr>
<td>MealTray</td>
<td>Mt</td>
<td>ENV</td>
<td>IsEnvObject, IsFillable, IsGrippable, IsPlaceable, IsPlatform, FitsToContainer</td>
</tr>
<tr>
<td>Base</td>
<td>Bs</td>
<td>ENV</td>
<td>IsEnvObject, IsCuboid, IsPlatform, IsGrippable, IsPlaceable</td>
</tr>
<tr>
<td>Handle</td>
<td>Hdl</td>
<td>ENV</td>
<td>IsEnvObject, IsCylinder, IsGrippable</td>
</tr>
<tr>
<td>Plate</td>
<td>Pl</td>
<td>ENV</td>
<td>IsEnvObject, IsCylinder, IsFillable, IsPlaceable</td>
</tr>
<tr>
<td>Spoon</td>
<td>Sp</td>
<td>ENV</td>
<td>IsEnvObject, IsFillable, IsGrippable, IsPlaceable</td>
</tr>
<tr>
<td>Lid</td>
<td>Ld</td>
<td>ENV</td>
<td>IsEnvObject, IsCylinder, IsGrippable, IsPlaceable</td>
</tr>
<tr>
<td>Table</td>
<td>Ta</td>
<td>ENV</td>
<td>IsEnvObject, IsPlatform</td>
</tr>
</tbody>
</table>

Table 9.3. Further TPO in ADL scenario, including the components, sub-components, abbreviations and characteristics of the involved objects. Categorization and relationships of involved objects are also included in Figure 9.1.

Some of these further TPOs are smart distributed components and therefore are specified as system object and as environmental object with the help of the respective characteristics. Nevertheless, the first characteristic is again fix according to Definition 7 (on page 78). The fridge is such a smart component, since it is equipped with a mechanism for automatic door opening and closing. This door opener can be actuated from the system and can therefore theoretically\(^1\) be included into task planning as a system resource. However, the fridge is physically separated from the robotic system and is therefore considered as environmental object as well. Additional object characteristics are again introduced according to the ontology as specified in Figure 5.3 on page 70 and as described in more detail in Section 6.1.2.

Special attention has also to be given to the main environmental object that is in the focus of this scenario - the meal tray. This object is a special tool as introduced and explained in Section 2.3.1. As it is also visible in Figure 2.3 (page 13) is consists of several components: A base, where the plate is inserted in, and which is equipped with a handle specially adapted for easy recognition by the vision system and to be grasppable by the manipulator. Furthermore the spoon has a special resting place and a lid covers the plate. The sub-components spoon and lid are also handled as separated components in certain sub-scenarios as detailed in the following.

9.3.3. Sub-Scenario #1: Fetch Meal from Fridge

TPOs of this sub-scenario are fridge (Fr), meal tray (Mt) and manipulator (MP). Additionally, the sub-symbolic item InsertLoc is necessary to parameterize the pair of COPs that model the retrieval of the meal tray from the fridge and the reverse operation. Figure 9.3 depicts the resulting \(PS_A\) and the tables below describe the involved OCs with assigned facts as well as the COPs that connect the OCs. The default initial situation is marked with orange in the \(PS_A\) and the default target situation is marked

\(^1\)Even if this option has been excluded as argued in Section 9.2 it is nevertheless possible in other cases and scenarios.

Sub-scenario 1: Fetch meal from fridge

(i.e. take a meal on a meal tray out of the fridge and transfer it to a free position in the workspace).

Object constellations and facts

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fr.1-Mt.1.0</td>
<td>ContainerAccessible(Fr.1), IsInsideContainer(Mt.1,Fr.1), not.HasFreeStoringPlace(Fr.1)</td>
</tr>
<tr>
<td>2</td>
<td>MP.1.0</td>
<td>HoldsNothing(MP.1), IsInFreePos(MP.1)</td>
</tr>
<tr>
<td>3</td>
<td>Fr.1-MP.1-Mt.1.0</td>
<td>not.HasFreeStoringSpace(Fr.1), not.IsInRelLoc(MP.1,Mt.1,Fr.1,InsertLoc), IsInsideContainer(Mt.1,Fr.1), not.IsInFreePos(MP.1), ContainerAccessible(Fr.1), not.HoldsNothing(MP.1)</td>
</tr>
<tr>
<td>4</td>
<td>Fr.1-MP.1-Mt.1.1</td>
<td>not.IsInFreePos(MP.1), IsInRelLoc(MP.1,Mt.1,Fr.1,InsertLoc), IsGripped(MP.1,Mt.1), ContainerAccessible(Fr.1), HasFreeStoringSpace(Fr.1), not.IsInsideContainer(Mt.1,Fr.1), not.HoldsNothing(MP.1)</td>
</tr>
<tr>
<td>5</td>
<td>Fr.1.0</td>
<td>ContainerAccessible(Fr.1), HasFreeStoringPlace(Fr.1)</td>
</tr>
<tr>
<td>6</td>
<td>MP.1-Mt.1.0</td>
<td>IsGripped(MP.1,Mt.1), IsInFreePos(MP.1), not.HoldsNothing(MP.1)</td>
</tr>
</tbody>
</table>

Table 9.4. Object constellations and facts in sub-scenario #1: Fetch meal from fridge.

COPs

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AOP: GraspObjectInContainer.1( MP.1, Mt.1, Fr.1 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOP: DepartFromContainer.1( MP.1, Mt.1, Fr.1 )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>IST: PlaceObjectInside.1( MP.1, Mt.1, Fr.1, InsertLoc )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IST: GetObjectOutside.1( MP.1, Mt.1, Fr.1, InsertLoc )</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>AOP: MoveObjectToRelLoc.1( MP.1, Mt.1, Fr.1, InsertLoc )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOP: MoveObjectFromRelLoc.1( MP.1, Mt.1, Fr.1, InsertLoc )</td>
<td></td>
</tr>
</tbody>
</table>

Table 9.5. Composed operators in sub-scenario #1: Fetch meal from fridge.
9.3.4. Sub-Scenario #2: Cook Meal

The TPOs in the second sub-scenario are the meal tray (Mt), the microwave oven (Mw), the manipulator (MP) and the wheelchair tray (Tr). Sub-symbolic items are InsertLoc and GripperInFrontOfDoor, the first one to represent the insert location for the meal tray in front of the microwave oven and the latter one to model the gripper in front of the microwave door, where it pushes the door to close. This location in front of the door is a special relative location with respect to the door. Therefore, the same symbol can be used independent of the closing/opening state of the microwave oven. The pictographic $PS_A$ as it results from the PSA-Designer is depicted in Figure 9.4 and the object constellations with facts as well as the involved COPs are extracted from the $PS_A$ and are summarized in the tables in the following.

<table>
<thead>
<tr>
<th>Object constellations and facts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Mw.1_0</td>
</tr>
<tr>
<td>2 Mw.1_1</td>
</tr>
<tr>
<td>3 MP.1-Mt.1_0</td>
</tr>
<tr>
<td>4 MP.1-Mt.1-Mw.1_0</td>
</tr>
<tr>
<td>5 MP.1-Mt.1-Mw.1_1</td>
</tr>
<tr>
<td>6 MP.1_0</td>
</tr>
<tr>
<td>7 Mt.1-Mw.1_0</td>
</tr>
<tr>
<td>8 MP.1-Mt.1-Mw.1_2</td>
</tr>
<tr>
<td>9 MP.1-Mt.1-Mw.1_3</td>
</tr>
<tr>
<td>10 Mt.1-Mw.1_1</td>
</tr>
<tr>
<td>11 Mt.1-Mw.1_2</td>
</tr>
</tbody>
</table>

Table 9.6. Object constellations and facts in sub-scenario #2: Cook meal.

9.3.5. Sub-Scenario #3: Fetch Meal from Oven

In the third sub-scenario the meal is fetched from the oven after the completion of the heating procedure and is placed on the tray of the wheelchair in front of the user. The respective $PS_A$ is is not depicted here since it is similar to the Fetch meal from fridge sub-scenario (Figure 9.3). TPOs in this $PS_A$
Figure 9.4. Sub-scenario 2: *Cook meal* (i.e. place it in the microwave oven and start heating).
9. Application and Evaluation of the FRIEND::Process in AMaRob Scenarios

<table>
<thead>
<tr>
<th>COPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 IST: OpenAutomaticDoor.1( Mw.1 )</td>
</tr>
</tbody>
</table>
| 2 AOP: MoveObjectToRelLoc.1( MP.1, Mt.1, Mw.1, InsertLoc )  
  DOP: MoveObjectFromRelLoc.1( MP.1, Mt.1, Mw.1, InsertLoc ) |
| 3 IST: PlaceObjectInside.1( MP.1, Mt.1, Mw.1, InsertLoc )  
  IST: GetJsonObject.1( MP.1, Mt.1, Mw.1, InsertLoc ) |
| 4 AOP: GraspObjectInContainer.1( MP.1, Mt.1, Mw.1 )  
  DOP: DepartFromContainer.1( MP.1, Mt.1, Mw.1 ) |
| 5 AOP: MoveToRelLoc.1( MP.1, Mw.1, GripperInFrontOfDoor )  
  DOP: MoveFromRelLoc.1( MP.1, Mw.1, GripperInFrontOfDoor ) |
| 6 IST: CloseDoor.1( MP.1, Mw.1, GripperInFrontOfDoor ) |
| 7 AOP: MoveToRelLoc.2( MP.1, Mw.1, GripperInFrontOfDoor )  
  DOP: MoveFromRelLoc.2( MP.1, Mw.1, GripperInFrontOfDoor ) |
| 8 IST: CookMeal.1( Mw.1, Mt.1 ) |

Table 9.7. Composed operators in sub-scenario #2: Cook meal.

are the meal tray (Mt), the microwave oven (Mw), the manipulator (MP) and the wheelchair tray (Tr). Sub-symbolic items are InsertLoc, PlacedLoc and GripperInFrontOfDoor. The insert location is required for the COPs GetJsonObject and MoveObjectFromRelLocation. The term InsertLoc rather fits to their reverse COPs, but each pair of COPs has to use the same parameters according to the logical rules of And-Or nets. The item PlacedLoc represents a free location on the wheelchair tray and GripperInFrontOfDoor is equivalently used as in the sub-scenario before.

9.3.6. Sub-Scenario #4: Take Away the Lid

After placing the meal in front of the user, the lid which covers the meal has to be taken away\(^2\). This procedure is subject to sub-scenario number four and the respective PS\(_A\) is depicted in Figure 9.5. TPOs in this process-structure are the lid (Ld), the meal tray (Mt), the manipulator (MP), the wheelchair tray (Tr) as well as a table (Ta). The latter one represents a suitable target location for placing the lid after lifting it from the meal tray. The sub-symbolic item PlacedLoc is used twice in this scenario. It denotes the lid's resting position on the meal tray and also the target location of the lid on the table. This multiple usage is possible, since the item is interpreted as an abstract descriptor that is finally used as skill parameter on elementary level to parameterize the skills in the sense of specifying some arbitrary free location on a platform.

9.3.7. Sub-Scenario #5: Eating support

In this sub-scenario the user in the wheelchair is supported in eating the meal. The PS\(_A\) that describes this process is depicted in Figure 9.6. Besides the already known TPOs (Mt, MP, Tr) a spoon (Sp), which is part of the composed object meal tray, and the user (Usr) are modeled. Furthermore, the three sub-symbolic items SpoonOnMealtray, TakeFoodLoc and ServeFoodLoc are required. The first one represents the spoon’s resting location on the meal tray and TakeFoodLoc specifies the location of the spoon with respect to the meal tray to take food from the plate. ServeFoodLoc is the location near

\(^2\)It has to be remarked that the lid has not been modeled explicitly before and therefore it is only visible in this PS\(_A\).
Table 9.8. Object constellations and facts in sub-scenario #3: Fetch meal from oven.

| 1 | Mt.1-Mw.1 | not.ContainerAccessible(Mw.1), IsInsideContainer(Mt.1,Mw.1), not.HasFreeStoringSpace(Mw.1) |
| 2 | Mt.1-Mw.1 | ContainerAccessible(Mw.1), IsInsideContainer(Mt.1,Mw.1), not.HasFreeStoringSpace(Mw.1) |
| 3 | MP.1 | HoldsNothing(MP.1), IsInFreePos(MP.1) |
| 4 | MP.1-Mt.1-Mw.1 | not.HasFreeStoringSpace(Mw.1), not.IsInRelLoc(MP.1,Mt.1,Mw.1,InsertLoc), IsInsideContainer(Mt.1,Mw.1), ContainerAccessible(Mw.1), IsGripped(MP.1,Mt.1), not.HoldsNothing(MP.1), not.IsInFreePos(MP.1) |
| 5 | MP.1-Mt.1-Mw.1 | ContainerAccessible(Mw.1), HasFreeStoringSpace(Mw.1), IsGripped(MP.1,Mt.1), IsInRelLoc(MP.1,Mt.1,Mw.1,InsertLoc), not.IsInsideContainer(Mt.1,Mw.1), not.HoldsNothing(MP.1), not.IsInFreePos(MP.1) |
| 6 | MP.1-Mt.1 | IsGripped(MP.1,Mt.1), IsInFreePos(MP.1), not.HoldsNothing(MP.1) |
| 7 | Mw.1 | ContainerAccessible(Mw.1), HasFreeStoringSpace(Mw.1) |
| 8 | Tr.1 | |
| 9 | MP.1-Mt.1-Tr.1 | not.IsInFreePos(MP.1), IsInPlacedLoc(Mt.1,Tr.1,PlacedLoc), IsPlacedOn(Mt.1,Tr.1), IsGripped(MP.1,Mt.1), not.HoldsNothing(MP.1) |
| 10 | Mt.1-Tr.1 | IsPlacedOn(Mt.1,Tr.1), IsInPlacedLoc(Mt.1,Tr.1,PlacedLoc) |
| 11 | MP.1-Mw.1 | GripperIsInRelLoc(MP.1,Mw.1,GripperInFrontOfDoor), not.IsInFreePos(MP.1), HoldsNothing(MP.1), ContainerAccessible(Mw.1), HasFreeStoringSpace(Mw.1) |
| 12 | MP.1-Mw.1 | not.ContainerAccessible(Mw.1), GripperIsInRelLoc(MP.1,Mw.1,GripperInFrontOfDoor), not.IsInFreePos(MP.1), HoldsNothing(MP.1), HasFreeStoringSpace(Mw.1) |
| 13 | Mw.1 | not.ContainerAccessible(Mw.1), HasFreeStoringSpace(Mw.1) |

Table 9.9. Composed operators in sub-scenario #3: Fetch meal from oven.

| 1 | IST: OpenAutomaticDoor.1( Mw.1 ) |
| 2 | AOP: GraspObjectInContainer.1( MP.1, Mt.1, Mw.1 ) | DOP: DepartFromContainer.1( MP.1, Mt.1, Mw.1 ) |
| 3 | IST: GetObjectOutside.1( MP.1, Mt.1, Mw.1, InsertLoc ) | IST: PlaceObjectInside.1( MP.1, Mt.1, Mw.1, InsertLoc ) |
| 4 | AOP: MoveObjectToRelLoc.1( MP.1, Mt.1, Mw.1, InsertLoc ) | DOP: MoveObjectFromRelLoc.1( MP.1, Mt.1, Mw.1, InsertLoc ) |
| 5 | AOP: PutDownObject.1( MP.1, Mt.1, Tr.1, PlacedLoc ) | DOP: LiftObject.1( MP.1, Mt.1, Tr.1, PlacedLoc ) |
| 6 | AOP: GraspObject.1( MP.1, Mt.1, Tr.1 ) | DOP: Depart.1( MP.1, Mt.1, Tr.1 ) |
| 7 | AOP: MoveToRelLoc.1( MP.1, Mw.1, GripperInFrontOfDoor ) | DOP: MoveFromRelLoc.1( MP.1, Mw.1, GripperInFrontOfDoor ) |
| 8 | IST: CloseDoor.1( MP.1, Mw.1, GripperInFrontOfDoor ) |
| 9 | AOP: MoveToRelLoc.2( MP.1, Mw.1, GripperInFrontOfDoor ) | DOP: MoveFromRelLoc.2( MP.1, Mw.1, GripperInFrontOfDoor ) |

the user’s mouth which is the target location of serving food to the user. After the manipulator has transferred the spoon to this location, the user will be able to take the food from the spoon on his own.
9. Application and Evaluation of the FRIEND::Process in AMaRob Scenarios

Figure 9.5. Sub-scenario 4: Take away lid (i.e. take the lid from the mealtray).

<table>
<thead>
<tr>
<th>Object constellations and facts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Ld.1-Mt.1-Tr.1_0</td>
</tr>
<tr>
<td>2 MP.1_0</td>
</tr>
<tr>
<td>3 Ld.1-MP.1-Mt.1-Tr.1_0</td>
</tr>
<tr>
<td>4 Ld.1-MP.1_0</td>
</tr>
<tr>
<td>5 Mt.1-Tr.1_0</td>
</tr>
<tr>
<td>6 Ta.1_0</td>
</tr>
</tbody>
</table>

Table 9.10. Object constellations and facts in sub-scenario #4: Take away the lid.

<table>
<thead>
<tr>
<th>COPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 AOP: GraspObject.1( MP.1, Ld.1, Mt.1 )</td>
</tr>
<tr>
<td>DOP: Depart.1( MP.1, Ld.1, Mt.1 )</td>
</tr>
<tr>
<td>2 AOP: PutDownObject.1( MP.1, Ld.1, Mt.1, PlacedLoc )</td>
</tr>
<tr>
<td>DOP: LiftObject.1( MP.1, Ld.1, Mt.1, PlacedLoc )</td>
</tr>
<tr>
<td>3 AOP: PutDownObject.2( MP.1, Ld.1, Ta.1, PlacedLoc )</td>
</tr>
<tr>
<td>DOP: LiftObject.2( MP.1, Ld.1, Ta.1, PlacedLoc )</td>
</tr>
<tr>
<td>4 AOP: GraspObject.2( MP.1, Ld.1, Ta.1 )</td>
</tr>
<tr>
<td>DOP: Depart.2( MP.1, Ld.1, Ta.1 )</td>
</tr>
</tbody>
</table>

Table 9.11. Composed operators in sub-scenario #4: Take away the lid.
Step 2a - Abstract Process-Structure Specification

Sub-scenario 5: Meal eating support (i.e. the robot supports the user in eating).

According to the shared-mode control paradigm the user might be able to intervene in the process to determine a comfortable location for emptying the spoon.

**Special attention** has to be paid to the setting of the (default) target situation in this process-structure. At first glance, the target situation that involves $OC_{11}$ (MP.1-Sp.1-Usr.1) might be appropriate. In this situation the user has taken the food from the spoon and the main eating cycle is ready to be continued. However, it makes no sense to model a process-structure with cycles and to define initial and target situation as the same set of OCs. In such a case, the task planner won’t be able to determine any operation sequence, since the target seems to be reached right at the beginning already. Therefore, the special consideration in this process-structure is to define the target situation as the situation that includes $OC_9$ (MP.1-Sp.1-Usr.1). This is the situation where the filled spoon is transferred near the user’s mouth, right before the user will take the food from the spoon. Thus, the COP number 4 (UserTakesFood) is included in the $PS_A$ for the sake of achieving a complete description of the process, however it is not included into the real task execution. In fact, no action is required by the system during this user action. The user, after emptying the spoon, just activates the next cycle or requests to stop eating by input of a new task request via the human machine interface. In both cases the initial monitoring will determine the situation with $OC_{11}$ (MP.1-Sp.1-Usr.1) as initial situation and the task planner will determine the right sequence of actions due to the unidirectional COP number 4 (UserTakesFood).

---

3If the human machine interface is able to recognize the user’s intention in this situation, the input of the task request, especially in this situation where the multiple repetition of a cyclic action is desired, can be simplified. The concrete implementation of this functionality is not discussed here as it is outside the scope of this work.
Table 9.12. Object constellations and facts in sub-scenario #5: Eating support.

<table>
<thead>
<tr>
<th>COPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 AOP: GraspObject.1( MP.1, Sp.1, Mt.1 )</td>
</tr>
<tr>
<td>DOP: Depart.1( MP.1, Sp.1, Mt.1 )</td>
</tr>
<tr>
<td>2 AOP: InsertObject.1( MP.1, Sp.1, Mt.1, Tr.1, SpoonOnMealtray )</td>
</tr>
<tr>
<td>DOP: LiftObjectFromInsertLoc.1( MP.1, Sp.1, Mt.1, Tr.1, SpoonOnMealtray )</td>
</tr>
<tr>
<td>3 AOP: MoveObjectToRelLoc.1( MP.1, Sp.1, Mt.1, TakeFoodLoc )</td>
</tr>
<tr>
<td>DOP: MoveObjectFromRelLoc.1( MP.1, Sp.1, Mt.1, TakeFoodLoc )</td>
</tr>
<tr>
<td>4 IST: TakeFood.1( MP.1, Sp.1, Mt.1, Tr.1, TakeFoodLoc )</td>
</tr>
<tr>
<td>5 AOP: MoveObjectToRelLoc.4( MP.1, Sp.1, Mt.1, TakeFoodLoc )</td>
</tr>
<tr>
<td>DOP: MoveObjectFromRelLoc.4( MP.1, Sp.1, Mt.1, TakeFoodLoc )</td>
</tr>
<tr>
<td>6 AOP: MoveObjectToRelLoc.2( MP.1, Sp.1, Usr.1, ServeFoodLoc )</td>
</tr>
<tr>
<td>DOP: MoveObjectFromRelLoc.2( MP.1, Sp.1, Usr.1, ServeFoodLoc )</td>
</tr>
<tr>
<td>7 IST: UserTakesFood.1( Usr.1, Sp.1, MP.1, TakeFoodLoc )</td>
</tr>
<tr>
<td>8 AOP: MoveObjectToRelLoc.3( MP.1, Sp.1, Usr.1, ServeFoodLoc )</td>
</tr>
<tr>
<td>DOP: MoveObjectFromRelLoc.3( MP.1, Sp.1, Usr.1, ServeFoodLoc )</td>
</tr>
</tbody>
</table>

Table 9.13. Composed operators in sub-scenario #5: Eating support.

Modeled in the way as just specified, the three different usage modes can be triggered through different combinations of initial and target situations as follows:

Figure 9.7. Sub-scenario 6: Take away mealtray (i.e. take mealtray from tray).

- Start eating: \(S_I = \{OC_1 \ (MP.1.0), \ OC_2 \ (Mt.1-Sp.1-Tr.1.0), \ OC_{10} \ (Usr.1.0)\} \) and \(S_T = \{OC_5 \ (Mt.1-Tr.1.0), \ OC_9 \ (MP.1-Sp.1-Usr.1.0)\}\) (these are also the situations shown in Figure 9.6)

- Continue eating: \(S_I = \{OC_{11} \ (MP.1-Sp.1-Usr.1.1), \ OC_5 \ (Mt.1-Tr.1.0)\} \) and \(S_T = \{OC_9 \ (MP.1-Sp.1-Usr.1.0), \ OC_5 \ (Mt.1-Tr.1.0)\}\)

- Finish eating: \(S_I = \{OC_{11} \ (MP.1-Sp.1-Usr.1.1), \ OC_5 \ (Mt.1-Tr.1.0)\} \) and \(S_T = \{OC_1 \ (MP.1.0), \ OC_2 \ (Mt.1-Sp.1-Tr.1.0), \ OC_{10} \ (Usr.1.0)\}\)

9.3.8. Sub-Scenario #6: Take Away the Mealtray

The last sub-scenario that completes the AMaRob ADL scenario is structurally similar to sub-scenario #4 (Take away the lid). It operates on the same TPOs, with the only difference that the lid is not included anymore. The abstract process-structure is depicted in Figure 9.7.
### Object constellations and facts

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MP.1&lt;sub&gt;_0&lt;/sub&gt;</td>
<td>HoldsNothing(MP.1), IsInFreePos(MP.1)</td>
</tr>
<tr>
<td>2</td>
<td>Mt.1-Tr.1&lt;sub&gt;_0&lt;/sub&gt;</td>
<td>IsPlacedOn(Mt.1,Tr.1), IsInPlacedLoc(Mt.1,Tr.1,PlacedLoc)</td>
</tr>
<tr>
<td>3</td>
<td>MP.1-Mt.1-Tr.1&lt;sub&gt;_0&lt;/sub&gt;</td>
<td>not.HoldsNothing(MP.1), IsGripped(MP.1,Mt.1), not.IsInFreePos(MP.1), IsPlacedOn(Mt.1,Tr.1), IsInPlacedLoc(Mt.1,Tr.1,PlacedLoc)</td>
</tr>
<tr>
<td>4</td>
<td>MP.1-Mt.1&lt;sub&gt;_0&lt;/sub&gt;</td>
<td>IsGripped(MP.1,Mt.1), IsInFreePos(MP.1), not.HoldsNothing(MP.1)</td>
</tr>
<tr>
<td>5</td>
<td>Tr.1&lt;sub&gt;_0&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Ta.1&lt;sub&gt;_0&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>MP.1-Mt.1-Ta.1&lt;sub&gt;_0&lt;/sub&gt;</td>
<td>not.IsInFreePos(MP.1), IsInPlacedLoc(Mt.1,Ta.1,PlacedLoc), IsPlacedOn(Mt.1,Ta.1), IsGripped(MP.1,Mt.1), not.HoldsNothing(MP.1)</td>
</tr>
<tr>
<td>8</td>
<td>Mt.1-Ta.1&lt;sub&gt;_0&lt;/sub&gt;</td>
<td>IsPlacedOn(Mt.1,Ta.1), IsInPlacedLoc(Mt.1,Ta.1,PlacedLoc)</td>
</tr>
</tbody>
</table>

**Table 9.14.** Object constellations and facts in sub-scenario #6: Take away the mealtray.

### COPs

<table>
<thead>
<tr>
<th></th>
<th>AOP:</th>
<th>DOP:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GraspObject.1( MP.1, Mt.1, Tr.1 )</td>
<td>Depart.1( MP.1, Mt.1, Tr.1 )</td>
</tr>
<tr>
<td>2</td>
<td>PutDownObject.1( MP.1, Mt.1, Tr.1, PlacedLoc )</td>
<td>LiftObject.1( MP.1, Mt.1, Tr.1, PlacedLoc )</td>
</tr>
<tr>
<td>3</td>
<td>PutDownObject.2( MP.1, Mt.1, Ta.1, PlacedLoc )</td>
<td>LiftObject.2( MP.1, Mt.1, Ta.1, PlacedLoc )</td>
</tr>
<tr>
<td>4</td>
<td>GraspObject.2( MP.1, Mt.1, Ta.1 )</td>
<td>Depart.2( MP.1, Mt.1, Ta.1 )</td>
</tr>
</tbody>
</table>

**Table 9.15.** Composed operators in sub-scenario #6: Take away the mealtray.

In this process step the decomposition of COPs is developed in the form of elementary process-structures $PS_E$. All COPs that have been used in the six sub-scenarios as specified in the preceding section are summarized in Table 9.16. In total 19 COPs are required and it has to be pointed out that a high degree of re-usage of COPs is achieved throughout the scenarios.

The full set of function block networks of all required COPs can be found in the IAT repository under MASSiVE/etc/PlanningKnowledge/COPs. A subset of COPs is given in Appendix B. The COP $GraspObjectInContainer$ is discussed here in more detail as representative example.

From a structural viewpoint all FBNs are built up in a similar manner. The FBN of the example COP is shown in Figure 9.8. For the sake of constructing the FBNs as independent modules, the first step to be undertaken is the retrieval of necessary data about the objects to be manipulated (in the case of manipulative skills). The basic underlying assumption from the viewpoint of one FBN is that no data is available at the beginning. During runtime, the object anchoring framework [Prenzel, 2005] decides, whether to actually execute a skill or whether to omit its execution if the data to be acquired is already available and is considered as up-to-date.

Basic object data as required for manipulating an object is its location and size/shape. In the case an object of interest is located on top of a base object like on a platform or inside a container, this base object has to be acquired first. The latter aspect may also be relevant for objects that are already gripped. These are rather rare cases, but if for example the system is (re)started with an object already gripped, the determination of this object, i.e. the gripping location and size, is necessary. Therefore, the inclusion of the function blocks to acquire gripped objects are included for the reason of completeness, but their real execution will not take place in general.

<table>
<thead>
<tr>
<th>Type</th>
<th>COP Name and Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>IST</td>
<td>CloseDoor(Manipulator, Container, DoorLocation)</td>
</tr>
<tr>
<td>IST</td>
<td>CookMeal(Container, Object)</td>
</tr>
<tr>
<td>DOP</td>
<td>Depart(Manipulator, Object, Platform)</td>
</tr>
<tr>
<td>DOP</td>
<td>DepartFromContainer(Manipulator, Object, Container)</td>
</tr>
<tr>
<td>IST</td>
<td>GetObjectOutside(Manipulator, Object, Container, InsertLoc)</td>
</tr>
<tr>
<td>AOP</td>
<td>GraspObject(Manipulator, Object, Platform)</td>
</tr>
<tr>
<td>AOP</td>
<td>GraspObjectInContainer(Manipulator, Object, Container)</td>
</tr>
<tr>
<td>AOP</td>
<td>InsertObject(Manipulator, GrippedObject, TargetObject, Platform, PlacedLoc)</td>
</tr>
<tr>
<td>DOP</td>
<td>LiftObject(Manipulator, Object, Platform, PlacedLoc)</td>
</tr>
<tr>
<td>DOP</td>
<td>LiftObjectFromInsertLoc(Manipulator, GrippedObject, TargetObject, Platform, PlacedLoc)</td>
</tr>
<tr>
<td>DOP</td>
<td>MoveFromRelLoc(Manipulator, Container, GripperInFrontOfDoor)</td>
</tr>
<tr>
<td>AOP</td>
<td>MoveObjectToRelLoc(Manipulator, Object, Container, InsertLoc)</td>
</tr>
<tr>
<td>DOP</td>
<td>MoveObjectFromRelLoc(Manipulator, Object, Container, InsertLoc)</td>
</tr>
<tr>
<td>AOP</td>
<td>MoveToRelLoc(Manipulator, Container, GripperInFrontOfDoor)</td>
</tr>
<tr>
<td>IST</td>
<td>OpenAutomaticDoor(Container)</td>
</tr>
<tr>
<td>IST</td>
<td>PlaceObjectInside(Manipulator, Object, Container, InsertLoc)</td>
</tr>
<tr>
<td>AOP</td>
<td>PutDownObject(Manipulator, Object, Platform, PlacedLoc)</td>
</tr>
<tr>
<td>IST</td>
<td>TakeFood(Manipulator, Spoon, Object, Platform, TakeFoodLoc)</td>
</tr>
<tr>
<td>IST</td>
<td>UserTakesFood(User, Spoon, Manipulator, TakeFoodLoc)</td>
</tr>
</tbody>
</table>

Table 9.16. Complete list of COP prototypes in AMaRob scenario #1 (ADL, meal preparation and eating) in alphabetical order.
Figure 9.8. Function block network for COP GraspObjectInContainer.
According to these considerations, the first function blocks in Figure 9.8 are responsible for first acquiring the location and size of the container where the object of interest is placed in and subsequently determine the location and size of the object itself. The coloring of the function blocks with different colors in the given $PSE$ represents the different skill servers where the functions (skills/EEOPs) are executed on\(^4\). As shown in the given FBN, the required object data is tried to be acquired fully autonomously first with the help of machine vision skills. If these actions fail, the user is involved for object acquisition within a user-interaction skill. This setup rule for FBNs is based on the essential shared-mode control development paradigm of the pursued programming approach according to Definition 1 (page 4).

In general (even though not in the given example) a parallel activation of skills is possible. A basic precondition for parallel execution of skills is under all circumstances that no resource conflicts occur. This means, concurrently executed skills are not allowed to access the same resource. This is a condition which is not explicitly modeled on FBN level but on Petri net level. Taking the EEOP specification (which includes resource assignment) into account, the converted Petri net as the final form of $PSE$ and as being used for task execution includes unique places for each resource and thus blocks mutual resource access. Compare also Figure 2.10 on page 20 which shows a simplified example of the COP $\text{GraspObjectInContainer}$ as Petri net. There, the resource $SCam$ (stereo camera system) is shown as resource place.

Within the remaining part of the illustrated COP the manipulator skill plans and executes a coarse approach to the object in the container, based on the given object data. Subsequently, a user interaction skill is enforced, before continuing with the fine approach to the object. This forced user interaction is foreseen in the current development state of the system. The user is asked here to decide, whether the approaching of the object has been performed exactly enough to be able to continue with the automatic fine approach. Also, the user is able to perform the final approach on his own, if desired or required. In later development steps, this enforced user interaction will be removed if the system can judge on the approaching precision on its own via usage of local sensors mounted at the gripper. After finishing the complete approach to the object, the gripper is closed and the COP’s post-facts are set as final step.

Comparing the complexity to configure the exemplified function block network with the effort that is necessary to specify the equivalent Petri net, it has to be pointed out that the FBN-based approach reduces the required amount of time from hours to approximately 10 minutes. A manually specified Petri net like that one depicted in the Appendix in Figure E.1 on page 226 requires a very time consuming iterative specification/verification procedure. Also, as easily visible, the clarity of the network structure even in simpler networks gets quickly lost in manually composed Petri nets.

9.5. Step 3a/b - Skill Development

Table 9.17 shows the complete list of EEOPs that are included in the set of COPs as specified for the AMaRob ADL scenario throughout the last development step. Each of these EEOP specifications now serves as input for skill development according to the process-part described in Chapter 7.

The given EEOP overview shows a diverse distribution of skill implementations among the skill servers of the system. The largest category are the manipulative skills with 46.5% of all needed skills. This higher

---

\(^4\)Setting the color scheme is a functionality implemented in PSE-Designer. It can be chosen between coloring by block type (i.e. EEOP, Fact, Control, Logical block) or coloring by server type (e.g. MachineVision, UserInteraction or Manipulator skill server).
<table>
<thead>
<tr>
<th>Server</th>
<th>EEOP Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Calculation</td>
<td>CalcPlatformInsertLocation</td>
</tr>
<tr>
<td>2 MachineVision</td>
<td>AcquireObjectBySCam</td>
</tr>
<tr>
<td>3 MachineVision</td>
<td>AcquireObjectOnPlatformBySCam</td>
</tr>
<tr>
<td>4 MachineVision</td>
<td>AcquireObjectInContainerBySCam</td>
</tr>
<tr>
<td>5 MachineVision</td>
<td>AcquireGrippedObjectBySCam</td>
</tr>
<tr>
<td>6 MachineVision</td>
<td>SearchFreeLocationOnPlatformBySCam</td>
</tr>
<tr>
<td>7 Manipulator</td>
<td>CoarseApproachToObject</td>
</tr>
<tr>
<td>8 Manipulator</td>
<td>CoarseApproachToObjectInContainer</td>
</tr>
<tr>
<td>9 Manipulator</td>
<td>CloseGripper</td>
</tr>
<tr>
<td>10 Manipulator</td>
<td>CloseDoorByFTSControl</td>
</tr>
<tr>
<td>11 Manipulator</td>
<td>DepartFromObject</td>
</tr>
<tr>
<td>12 Manipulator</td>
<td>DepartFromObjectInContainer</td>
</tr>
<tr>
<td>13 Manipulator</td>
<td>DepartFromObjectOnPlatform</td>
</tr>
<tr>
<td>14 Manipulator</td>
<td>DepartFromObjectToLocation</td>
</tr>
<tr>
<td>15 Manipulator</td>
<td>DepartWithObject</td>
</tr>
<tr>
<td>16 Manipulator</td>
<td>DepartWithObjectToLocation</td>
</tr>
<tr>
<td>17 Manipulator</td>
<td>FineApproachToObject</td>
</tr>
<tr>
<td>18 Manipulator</td>
<td>FineApproachToObjectInContainer</td>
</tr>
<tr>
<td>19 Manipulator</td>
<td>MoveObjectInByFTSControl</td>
</tr>
<tr>
<td>20 Manipulator</td>
<td>MoveObjectToRelLocation</td>
</tr>
<tr>
<td>21 Manipulator</td>
<td>MoveObjectOutByFTSControl</td>
</tr>
<tr>
<td>22 Manipulator</td>
<td>MoveToLocation</td>
</tr>
<tr>
<td>23 Manipulator</td>
<td>MoveToRelLocation</td>
</tr>
<tr>
<td>24 Manipulator</td>
<td>OpenGripper</td>
</tr>
<tr>
<td>25 Manipulator</td>
<td>PlaceObjectOnPlatformByFTSControl</td>
</tr>
<tr>
<td>26 Manipulator</td>
<td>TakeFood</td>
</tr>
<tr>
<td>27 MicrowaveOven</td>
<td>CookMeal</td>
</tr>
<tr>
<td>28 MicrowaveOven</td>
<td>OpenAutomaticDoor</td>
</tr>
<tr>
<td>29 Tray</td>
<td>EnhanceObjectAcquisitionOnSmartPlatform</td>
</tr>
<tr>
<td>30 Tray</td>
<td>SearchFreeLocationOnSmartPlatform</td>
</tr>
<tr>
<td>31 UserInteraction</td>
<td>AdjustObjectRelToObject</td>
</tr>
<tr>
<td>32 UserInteraction</td>
<td>AdjustToObject</td>
</tr>
<tr>
<td>33 UserInteraction</td>
<td>DetermineCookingInfo</td>
</tr>
<tr>
<td>34 UserInteraction</td>
<td>DetermineDepartLocation</td>
</tr>
<tr>
<td>35 UserInteraction</td>
<td>DetermineGrippedObjectBySCam</td>
</tr>
<tr>
<td>36 UserInteraction</td>
<td>DetermineObjectBySCam</td>
</tr>
<tr>
<td>37 UserInteraction</td>
<td>DetermineObjectDepartLocation</td>
</tr>
<tr>
<td>38 UserInteraction</td>
<td>MoveObjectIn</td>
</tr>
<tr>
<td>39 UserInteraction</td>
<td>MoveObjectOut</td>
</tr>
<tr>
<td>40 UserInteraction</td>
<td>MoveObjectToFreeLocation</td>
</tr>
<tr>
<td>41 UserInteraction</td>
<td>MoveToFreeLocation</td>
</tr>
<tr>
<td>42 UserInteraction</td>
<td>SearchLocationOnPlatformByUser</td>
</tr>
<tr>
<td>43 UserInteraction</td>
<td>UserTakesFood</td>
</tr>
</tbody>
</table>

Table 9.17. Complete list of EEOP prototypes in AMaRob ADL scenario (meal preparation and eating), sorted according to their assignment to a skill server.
number of different skills in one skill server originates from the variability of skill requirements that have to be considered. Manipulative skills cannot be implemented as universal as this is possible for other skill types. During manipulator motion planning various strategies have to be considered depending on whether to move with or without a gripped object or whether a platform or container have to be considered as obstacle in the workspace.

The user interaction skills are the second largest group of skills (30.2\% of complete set of skills). This is due to the fact that according to the shared-control concept a user interaction skill is usually applied in the case of erroneous execution of an autonomous system skill. The slightly smaller amount of required user interaction skills hints to a higher degree of universality of this skill type in comparison to manipulative skills. This fact becomes obvious if typical representative implementations of these skills are inspected. As the user interaction skills rely on the integration of the user’s cognitive capabilities into task execution, these skill types are less complex and less specific in their implementation. For example, many use cases can be satisfied with the help of the skills AdjustToObject or AdjustObjectRelToObject without considering specific nearby obstacles like a container or a platform. Respecting these different kinds of obstacles is not included in the algorithmic implementations, but the user is responsible for this.

The third group of skills is the machine vision skills (11.6\% of all skills). The rather small amount of necessary skills is not correlated to the complexity of skill implementations. The machine vision skills as specified here have to work under different conditions and situations. The most prevalent strategy as specified within the PS\_E\_S is to acquire the base object first (e.g. platform or container) which carries the object of interest, and subsequently acquire this target object. Consequently, the skill AcquireObjectBySCam covers most use cases, followed by object acquisition either on a platform or in a container.

The discussion of the concrete design and implementation of all the skills gathered here is far beyond the scope of this work. The concepts for the realization of manipulative skills within the FRIEND::Architecture are discussed in [Ojdańi, 2009] and related papers. The underlying principles used in machine vision skills can be found in [Grigorescu et al., 2008, Grigorescu et al., 2009]. With respect to the impact of the FRIEND::Process, the implementation examples used for illustration in Section 7.1 show, how the process model achieves a uniform pattern for the overall infrastructure of each skill. This in turn leads to easier orientation and guidance for a skill programmer and finally paves the way for a longterm maintainability of skill implementations.

9.6. Step 4a - Skill Testing

According to the elaborations of the process model phase skill testing in Section 8.1, the input artifacts of this development step are a single or a sequence of EEOPs (skills) that are under test. Concrete input stimuli of testing are skill test sequences as well as world model test data. Exemplifications of both is given in Table 9.18 (skill test sequence) and Table 9.19 (world model test data).

The skill test sequence used here starts with a simulative execution of a coarse approach to a meal tray in a fridge based on the specified world model test data. The second skill opens the gripper and subsequently the final approach of the meal tray is performed, before closing the gripper. The given test sequence finishes with retrieval of the meal tray from the fridge.
### 9. Application and Evaluation of the FRIEND::Process in AMaRob Scenarios

<table>
<thead>
<tr>
<th>Skill Server</th>
<th>Skill Name</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Manipulator CoarseApproachToObjectInContainer</td>
<td>Robotarm, Fr.1.EEL.Loc, Fr.1.EEL.SCub, Mt.1.EEL.Loc, Mt.1.EEL.SCub</td>
</tr>
<tr>
<td>2</td>
<td>Manipulator OpenGripper</td>
<td>Gripper</td>
</tr>
<tr>
<td>3</td>
<td>Manipulator FineApproachToObjectInContainer</td>
<td>Robotarm, Fr.1.EEL.Loc, Fr.1.EEL.SCub, Mt.1.EEL.Loc, Mt.1.EEL.SCub</td>
</tr>
<tr>
<td>4</td>
<td>Manipulator CloseGripper</td>
<td>Gripper</td>
</tr>
<tr>
<td>5</td>
<td>Manipulator MoveObjectOutByFTSControl</td>
<td>Robotarm, FTSensor, Mt.1, Fr.1.EEL.Loc, Fr.1.EEL.SCub</td>
</tr>
</tbody>
</table>

**Table 9.18.** Sample skill test sequence for grasping a meal tray (Mt.1) in a fridge (Fr.1).

<table>
<thead>
<tr>
<th>Data ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fr.1.EEL.Loc Location of composed object fridge</td>
</tr>
<tr>
<td>2</td>
<td>Fr.1.EEL.SCub Size of envelope of composed object fridge</td>
</tr>
<tr>
<td>3</td>
<td>Fr.1.EEL.ModelingState Detailed (sub-components are modeled) or enveloped modeling state</td>
</tr>
<tr>
<td>4</td>
<td>Fr.1.EEL.SizeComposed IDs for size data of sub-components (e.g. Fr.1.Back.EEL.SCub). For each sub-component location data also exists.</td>
</tr>
<tr>
<td>5</td>
<td>Fr.1.Back.EEL.Loc/SCub Location and size of fridge back plane</td>
</tr>
<tr>
<td>6</td>
<td>Fr.1.Door.EEL.Loc/SCub Location and size of fridge door</td>
</tr>
<tr>
<td>7</td>
<td>Fr.1.Down.EEL.Loc/SCub Location and size of fridge bottom plane</td>
</tr>
<tr>
<td>8</td>
<td>Fr.1.Left.EEL.Loc/SCub Location and size of fridge left plane</td>
</tr>
<tr>
<td>9</td>
<td>Fr.1.Right.EEL.Loc/SCub Location and size of fridge right plane</td>
</tr>
<tr>
<td>10</td>
<td>Fr.1.LowerShelf.EEL.Loc/SCub Location and size of fridge lower shelf</td>
</tr>
<tr>
<td>11</td>
<td>Fr.1.UpperShelf.EEL.Loc/SCub Location and size of fridge upper shelf</td>
</tr>
<tr>
<td>12</td>
<td>Fr.1.Up.EEL.Loc/SCub Location and size of fridge top plane</td>
</tr>
<tr>
<td>13</td>
<td>Mt.1.EEL.Loc Location of composed object meal tray</td>
</tr>
<tr>
<td>14</td>
<td>Mt.1.EEL.SCub Size of envelope of composed object meal tray</td>
</tr>
<tr>
<td>15</td>
<td>Mt.1.EEL.ModelingState Detailed (sub-components are modeled) or enveloped modeling state</td>
</tr>
<tr>
<td>16</td>
<td>Mt.1.EEL.SizeComposed IDs for size data of sub-components (e.g. Mt.1.BaseHandle.EEL.SCyl). For each sub-component location data also exists.</td>
</tr>
<tr>
<td>17</td>
<td>Mt.1.Base.EEL.Loc/SCub Location and size of meal tray base</td>
</tr>
<tr>
<td>18</td>
<td>Mt.1.BaseHandle.EEL.Loc/SCyl Location and size of meal tray base handle</td>
</tr>
<tr>
<td>19</td>
<td>Mt.1.MealCover.EEL.Loc/SCyl Location and size of meal cover</td>
</tr>
<tr>
<td>20</td>
<td>Mt.1.MealCoverHandle.EEL.Loc/SCyl Location and size of meal cover handle</td>
</tr>
<tr>
<td>21</td>
<td>Mt.1.MealPlate.EEL.Loc/SCyl Location and size of meal plate</td>
</tr>
<tr>
<td>22</td>
<td>Mt.1.SpoonMainPart.EEL.Loc/SCub Location and size of spoon main part</td>
</tr>
<tr>
<td>23</td>
<td>Mt.1.SpoonHandle.EEL.Loc/SCyl Location and size of spoon handle</td>
</tr>
<tr>
<td>24</td>
<td>Mt.1.Top.EEL.Loc/SCub Location and size of top cover</td>
</tr>
<tr>
<td>25</td>
<td>Mt.1.TopHandle.EEL.Loc/SCub Location and size of handle of top cover</td>
</tr>
<tr>
<td>26</td>
<td>Mt.1.GraspRelLocation.EEL.Loc.Temp Grasping relative location for meal tray</td>
</tr>
</tbody>
</table>

**Table 9.19.** World model test data for grasping a meal tray (Mt.1) in a fridge (Fr.1).
A visualization of this simulative execution of skills is depicted in the screenshots in Figure 9.9. In the first sub-figure, the initial situation is given. In the second view (Figure 9.9(b)), the manipulator has approached and already grasped the meal tray and in the last figure on the right, the tray is to be taken out of the fridge.

besides inspecting the skill motion simulation, the animation feature of model execution within rhapsody is used. this means, in parallel to the motion visualization, the skill execution on the level of method calls is visualized with the help of animated sequence diagrams. the corresponding diagram for the motion simulation of the skill coarseapproachtoobjectincontainer is given in figure 9.10.

this diagram shows the three involved main components manipulator skill server as well as skill parameter manager and the skill parameters. the core part with motion simulation has been cropped out for the reason of simplifying the diagram and the discussion here.

now, a comparison to the sequence diagram as specified during skill design (see section 7.1.3) has to take place to detect mismatches which hint to implementation defects. in the given simulation output the following actions take place in accordance to the skill design specification:

- logging of skill start
- creation of parameter manager
- addition of parameters to parameter manager
- conversion of local corba proxies for hardware, skill and data server
- activation of call back servant for calling of helper skills
- initialization of parameters with sub-symbolic data from world model
Figure 9.10. Animated sequence diagram, resulting from simulative skill execution.
9.7. Step 4b - Scenario Testing

- Execution of main skill part (cropped out here, i.e. in this case the motion simulation takes place)
- Sending of skill return value via call-back
- Logging of skill termination
- Destruction of local objects like skill parameters and skill parameter manager

The most relevant animated sequence diagrams are kept and stored in the model repository to document the implementation status of a skill as well as to keep track of wanted and unwanted skill behavior. The different cases have to be clearly marked in the model to be easily distinguishable.

Besides the manual validation and verification an automatic step is also provided by Rhapsody, namely the automatic comparison of sequence diagrams. This feature is used to automate the comparison process. A sample comparison of the specification sequence diagram and the animated sequence diagram as obtained during skill simulation is shown in Figure 9.11. There, the methods calls from the specification (left side) are included in the animated sequence diagram (right side) and thus, the specification is fulfilled.

The automatically detected differences are highlighted and the tester is able to scan through the marked elements faster to see whether there is some erroneous mismatch which imposes a defect in the skill. The differences in the given example reveal no relevant mismatches: The method `setCompleteParamString()` which is marked here as missing and which is responsible to assign a complete parameter string has been omitted during skill design.

9.7. Step 4b - Scenario Testing

Scenario testing is done with the purpose to obtain a functional verification of the actions required for a complete scenario (task). The task planner is included in this testing step and therefore, several reactions on specific skill return values are under test. Thus it becomes possible to test all possible variations in a certain scenario.

As the tools required for scenario testing are known from Section 8.2, a sample testing procedure on this highest abstraction level will be discussed along with a sample execution protocol of skill testing. To avoid loss of overview this protocol has been moved to the Appendix C, showing the actions that are related to the execution of the skills within the COPs `GraspObjectInContainer` and `GetObjectOutside`. To remind the general overview of task planning actions, it is referred back to the sequence diagrams in the Figures 8.13 and 8.14 (page 166).

Especially, it has to be pointed out that the discussion of a typical task execution protocol is also of practical value for the application of the process model: An explanation of all generally required execution steps is exemplified within the protocol discussion.

The time stamps on the left side of the protocol are the same as those used in the following for easier comparison of descriptions and protocoded steps. They denote seconds and milliseconds.

- (13.832) The Sequencer core receives the task request in the form of a textual string.
Figure 9.11. Automatic comparison of animated sequence diagram (resulting from simulative skill execution, right side) with specification sequence diagram (resulting from skill design, left side).
9.7. Step 4b - Scenario Testing

- (13.836) The usage of task execution preparation and finishing is activated. This is configured via MASSiVE configurator. Preparation and finishing may include any action that is globally required before or after the execution of a certain task. The main justification for it is the transferring of the manipulator to a specific configuration to achieve a maximum motion freedom for the next manipulative step.

- (14.429) The world model server is connected and the modeling states of the TPOs is set to the detailed modeling level, i.e. explicit modeling of sub-components.

- (14.435) The default initial situation is extracted from the given process-structure and the related facts are set. In the case the initial monitoring is omitted (e.g. during scenario simulation) this initial situation directly serves as input for task planning.

- (15.985) During task execution preparation, the skill executer is initialized. The execution level is set to PROBABILITY_SIM, i.e. a probabilistic simulation of skills is performed (see also Section 4.5.1).

- (15.986) The connections to the available (and required) CORBA servers according to the system configuration via MASSiVE configurator are established.

- (16.724) Simulation and execution data is retrieved from the database. This includes for example the probabilistic values for skill simulation, skill time out values, etc.

- (23.285) The execution of skills always takes place according to the same pattern. However, the task preparation skill\ Manipulator.GoToReadyPose()\) is executed as single skill outside the scope of a task planning process (even though the task planner sends the execution command to the skill executer).

- (23.589) After reception of the skill return value Success the main task execution starts. Therefore, the skill executer and task planner’s main service threads are activated and planning is performed first on abstract and then on elementary level.

- (23.649) The two first skills that are planned within the COP GraspObjectInContainer (see also Figure 9.8 on page 186) are AcquireObjectBySCam for container recognition and AcquireObjectInContainerBySCam for the recognition of object to be grasped - the meal tray in this case. In the task execution as protocoled the execution of these skills is omitted. The data to be acquired is still available and valid, or in terms of object anchoring, the object symbols are still grounded/anchored.

- (23.653) The first executed skill is Manipulator.CoarseApproachToObjectInContainer. The skill execution level is set on all relevant skill servers and the successful start is protocoled.

- (23.696) The skill executer polls for a call back message of asynchronously executed skills.

- (24.056) Due to simulative execution the skill return value is received very fast after 60 ms. The call back proxy is released and the servant is deleted. The skill return value, Success in this case, is enqueued back to the task planner.
9. Application and Evaluation of the FRIEND::Process in AMaRob Scenarios

- (24.154) The task planner compares the execution result with the planned one. If there is a difference, re-planning has to be carried out. Otherwise, the skill execution according to the current plan can be continued.

- Altogether, the given protocol documents the following sequence of skills:

In COP GraspObjectInContainer:

- Manipulator.GoToReadyPose( Robotarm, fetch_meal_getoutsideonly.START ): Success
- Manipulator.AcquireObjectBySCam( Bumblebee, Fr.1.EEL.Loc, Fr.1.EEL.SCub ): Omitted, data available
- Manipulator.AcquireObjectInContainerBySCam( Bumblebee, Fr.1.EEL.Loc, Fr.1.EEL.SCub, Mt.1.EEL.Loc, Mt.1.EEL.SCub ): Omitted, data available
- Manipulator.CoarseApproachToObjectInContainer( Robotarm, Fr.1.EEL.Loc, Fr.1.EEL.SCub, Mt.1.EEL.Loc, Mt.1.EEL.SCub ): Success
- UserInteraction.AdjustToObject( Robotarm, Mt.1 ): SystemTakeOver
- Manipulator.FineApproachToObjectInContainer( Robotarm, Fr.1.EEL.Loc, Fr.1.EEL.SCub, Mt.1.EEL.Loc, Mt.1.EEL.SCub ): Success
- Manipulator.CloseGripper( Robotarm ): Success

In COP GetObjectOutside:

- Manipulator.AcquireObjectBySCam( Bumblebee, Fr.1.EEL.Loc, Fr.1.EEL.SCub ): Omitted, data available
- Manipulator.MoveObjectOutByFTSControl( Robotarm, FTSensor, Mt.1, Fr.1.EEL.Loc, Fr.1.EEL.SCub ): Success

- (25.788) The following is important to notice: The SystemTakeOver return value of the user interaction AdjustToObject represents in this case a simulated user intention that the system shall go on with task execution. In the protocol it is shown that re-planning of the high level plan is required. At this point it can be seen how the task planner and skill executer act independently: At time stamp 25.861 the planner sends an execution stop command to the skill executer and performs the re-planning. In parallel the skill executer checks for active skills being under execution. As there are none, it returns to the idle state at time stamp 25.892.

- (27.415) Task planner and executer terminate their main service thread.

- (27.578) Task execution finishing is performed, i.e. the manipulative skill GoToReadyPose is executed.

- (29.378) The Sequencer core changes to READY state.
9.8. Evaluation of Application Results

Throughout this chapter the complete development procedure according to the process model FRIEND::Process has been applied. The elaboration of the AMaRob ADL scenario served as example. During the final testing steps it could be shown that the development artifacts realize complete tasks as desired and intended according to the specifications gathered during scenario analysis.

However, besides thorough testing in different skill execution levels, the developed entities have all been applied in practice. The scenario *serve drink to user* has been shown during two public exhibitions, namely the 10th International Conference on Rehabilitation Robotics 2007 (ICORR’07) in The Netherlands and on the CeBIT fair, Hannover, Germany, in March 2008. The scenario shown there is very similar in its structure in comparison to the AMaRob ADL scenario: A drink in a bottle is fetched from the fridge, the drink is poured from the bottle into a glass on the tray and finally the glass is served to the user. Here, very similar process-structures and skills are involved and their development also took place with the help of the methods described in this work. Definitely, it has to be pointed out that the overall complexity of the complete AMaRob scenario is much larger than the drink-serving scenario. For the first time, this large number of sub-scenarios have been successfully concatenated and executed in a sequence.

The most recent and most complete demonstrations took place as a milestone presentation within the AMaRob consortium in December 2008. Here, the complete AMaRob ADL scenario as developed in this chapter was shown successfully. The development status is documented with the help of a video which is available on the AMaRob web page [IAT, 2009a].
A new process model for the development of semi-autonomous service robots has been elaborated within this work: **The FRIEND::Process**. This process model is an important contribution in line with the ongoing research efforts in the field of service robotics. Even though several experts have estimated the breakthrough of robotic solutions in the domestic field earlier within the last decades (e.g. [Engelberger, 1989]), the reality claims a more conservative view, as many scientific challenges remain [Kemp et al., 2007]. Today’s service robots, which really managed to enter human living environments, are for example autonomous vacuum cleaning or lawn mowing machines. These systems only display traces of intelligence and are specialists, designed for mostly one very particular task [Graefe and Bischoff, 2003].

A solution for the programming of service robots has been proposed in [Martens, 2003], which takes two essential development paradigms into account to construct a feasible and technically manageable service robotic system even based on the currently available technology: A semi-autonomous instead of fully autonomous control concept is chosen (Paradigm 1) and a pre-structuring of task knowledge is applied (Paradigm 2).

The FRIEND::Process embeds these two essential paradigms into a broader frame, as it is required for the practical development of a service robot. The resulting process model guides the developments within four main steps: (1) Scenario-driven analysis of new tasks, (2) user-friendly process-structure-driven task knowledge engineering as well as (3 + 4) model-driven skill development and testing. Besides providing a structured guidance for these four development steps, the following open problems have been solved with the process model:

- The programming steps before had not been or rather loosely been coupled with each other. A consistency throughout the programming steps mostly lay in the programmer’s responsibility. A structured enforcement of consistency was missing.
10. FRIEND::Process – Retrospective and Perspective

- Process Steps 1, 3 and 4, i.e. scenario analysis, skill development & skill testing, did not exist before and have been added.

- Development Step 2, i.e. task knowledge engineering, has been enhanced in all aspects, to become more user(programmer)-friendly. Before, difficult, time consuming and error prone manual programming was mandatory.

- Model-driven development (MDD) had not been applied before. Even though it was always aimed at object oriented programming in the software development of the FRIEND project, only tool engineered executable UML models exact a consistency of diagrammatic descriptions and implementations. Further main impact of MDD is the enforcement of uniform, re-usable and maintainable design and implementations. Without this technique, the complexity of large software systems can hardly be managed in practice and an integrated view into the complete system is often not available.

The FRIEND::Process

The FRIEND::Process integrates the following concepts:

- Scenario-driven task analysis
- User-friendly configuration of task knowledge
- Pictographic process-structures on scenario-Level
- Function block networks on system-level
- Model-driven development and testing of system skills

These concepts are integrated within the following development steps:

Development Step 1 - Scenario Analysis

During scenario analysis a formal and structured collection of task participating objects (TPO) that are in the focus of a certain scenario takes place. This is the initial step to prepare the pre-structuring of a task via process-structures. Since each process-structure represents one task (as e.g. Fetch meal, Serve drink, Clear tray), a scenario is split up into task-equivalent sub-scenarios. Each new TPO is added to the hierarchical object ontology, which collects and organizes the specification of object classes. Object classes are used throughout further development steps, like the configuration of process-structures and the algorithmic implementation of basic system functionality within the skills.
Development Step 2 - Configuration of Process-Structures

Based on the scenario analysis artifacts (TPOs), the task knowledge configuration takes place. First, the user-friendly and consistent management of task representing elements (TREs) takes place. Subsequently, the TREs are used as the basis for the configuration of process-structures on two different abstraction levels: the abstract scenario level as well as the elementary system level. On scenario level, a pictographic configuration approach is introduced, where the involved objects are assembled graphically in object constellations. In contrast to the previously existing method, the embodiment of involved objects does not have to take place in the programmer’s mind, but is visualized directly in the provided tool. Based on the AND/OR net syntax, the object constellations are connected with operators, the so-called composed operators (COPs). On system level, each COP is configured with the help of a function block network instead of previously used manually composed Petri nets. During the configuration of process-structures on both abstraction levels, integrated logical verification takes place, rejecting wrong user input. Finally, a conversion of function block networks to Petri nets is performed, where a final verification within the reachability graph excludes remaining errors, to assure the reachability of target states, as well as to avoid deadlock situations and resource conflicts.

Development Step 3 & 4 - Model-Driven Skill Development and Testing

Model-driven development techniques achieve a considerable improvement of the skill development and testing procedure. The skill design procedure starts with capturing of skill requirements and their grouping into skill use-cases. In the second step the skill interface specifications, which result from the previous development step, are used to parameterize a skill skeleton template, which is provided as UML flowchart. In conjunction with skill test data, which is set up according to the skill use-cases, the skill functionality is elaborated iteratively via model execution (skill testing) and enhancement of the basic skill skeleton behavior. The testing of skills can be compared to unit testing, whereas the testing on scenario scope is considered as the complete and final system testing. Due to the opportunity of conducting tests in different simulation and execution levels, the number of involved software and hardware components can be increased gradually; allowing to facilitate early tests in the development procedure. The proposed model-driven skill development strategy enforces a uniform implementation infrastructure, adequate modularization, documentation and organization of the development artifacts. The proposed method includes the usage of several helper constructs, which assure the integrity of the implementation and the specifications from the top-down direction of system development.

Evaluation of the FRIEND::Process

The FRIEND::Process tightly integrates the complete chain of developing service robot behavior and gives a structured guidance through all required steps. Just by comparison of the required efforts of manual programming versus the new and enhanced methods that are incorporated into the process, the evaluation can be easily quantified. However, besides the reduction of effort, the overall quality and clarity of development artifacts is decisively increased. This is expressed in the application of model-driven development techniques, which make it possible, to construct architectural designs as well as the implementation of behavior from executable UML models.
10. FRIEND::Process – Retrospective and Perspective

To name concrete examples of model elements, which are generally omnipresent throughout the process, it shall be referred to the skill design methodology, which prescribes flowchart templates. This hierarchical representation of a complex but re-usable infrastructure greatly facilitates the skill realization, even for novice programmers, who apply the FRIEND::Process for the first time. Furthermore, improved maintainability, modularization and documentation is enforced.

Another improvement that could be achieved with the FRIEND::Process is a tight consistency of developed artifacts. To give an example again, the enforced consistency between EEOP specifications (as they are derived from elementary process-structures) and the skill implementations shall be mentioned.

The FRIEND::Process organizes the collaboration of different new tools, which have not been used before for the development of FRIEND (or equivalent systems). These tools simplify the development procedure and provide more user-friendly and more intuitive mechanisms than before, leading to a much more user-friendly overall procedure.

Altogether, the FRIEND::Process decisively increases the development comfort (configuration instead of programming) and creates a solid base for the accessibility of the task knowledge configuration tools even by non-technical care personal or the impaired user himself. Consequently, it is imaginable that the patient in the wheelchair sets up process-structures on his own to enhance the system capabilities. Through development guided by the FRIEND::Process, the development artifacts are built right in the first instance. Thus, the basis for the application of the proposed development approach in practice is realized.

**Perspective of the FRIEND::Process**

Like all software artifacts, those ones involved in the FRIEND::Process are still improvable in several aspects. One issue is the fact that at the moment, only parts of the process model are already in line with the Automation by Configuration (AbC) principle, which shall be pursued further in the FRIEND projects.

Another unsolved problem is the automatic combination of sub-scenarios. At the moment, a manual combination and sequencing of different sub-scenarios takes place. The user has to select the right sequence and, in case of not matching ones, the system will return with an error. In the future, automatic sequencing of sub-scenarios will take place, realized by a planning above sub-scenario-level, i.e. above $PS_A$ level. The issue that has to be satisfied with automatic planning is the fact that a target situation of the preceding and the initial situation of the subsequent $PS_A$ have to be compatible with each other. This compatibility depends on the involved task participating objects and the fact state of the involved object constellations. Furthermore, this selection of $PS_A$s can also be coupled to a context/environment-sensitivity or to the user intention and might become an integral part of the FRIEND::Human-Machine-Interface.

As already anticipated in Section 7.1.6, an important enhancement of the FRIEND::Process is the usage of the Rhapsody API and an automatic coupling of the task knowledge database and the model repository. Thus, the consistency between skill specification and implementation can be verified automatically. Moreover, automatic generation of skill interfaces and the basic skill skeleton will become possible. Consequently, the introduction of a new EEOP in either the DBGUI or the PSE-Designer will
trigger the verification of the model repository to check, whether the respected skill already exists and
matches the specification or whether new model elements have to be generated.

FIN
A Compact Overview on the Process Model “FRIEND::Process”

Figure A.1 provides the complete overview of the process model FRIEND::Process, including all process steps (S) and process repositories (R), where the development artifacts (A) are stored. Different tools are used to produce, organize and store the various development artifacts. These tools embed a number of methods for specification and verification of the development artifacts. An online version of the FRIEND::Process summary is included in the IAT Wiki: [Prenzel, 2009].

A.1. Process Steps

Process Step 1 (S1): Scenario Analysis

**S1a - Scenario Specification:** Within the first process step the system items and environmental items involved in a certain task are specified and added to the item ontology. This is the informal specification of task participating objects (TPO, A1.1).

<table>
<thead>
<tr>
<th>Tool</th>
<th>Repository</th>
<th>Artifacts (IN)</th>
<th>Artifacts (OUT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rhapsody</td>
<td>Model Repository</td>
<td></td>
<td>Task Participating Objects (TPO)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tool</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T1</td>
<td>R1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A1.1</td>
</tr>
</tbody>
</table>
A. Compact Overview on the Process Model "FRIEND::Process"

Figure A.1. Scheme of process model "FRIEND::Process".
**S1b - Sub-Scenario Specification:** The task is split into re-usable sub-scenarios. A use case with detailed use case description is assigned to each sub-scenario.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Rhapsody</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repository</td>
<td>Model Repository</td>
<td>R1</td>
</tr>
<tr>
<td>Artifacts (IN)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Artifacts (OUT)</td>
<td>Sub-Scenario Use-Cases</td>
<td>A1.2</td>
</tr>
</tbody>
</table>

**Process Step 2 (S2): Task Knowledge Specification and Verification**

Within the second process step, the TPO are first formally specified within the DBGUI, which administers the task knowledge database and assures a logically consistent and user-friendly management of all task representing elements (TRE).

**S2a - Specification of Abstract Process-Structures ($PS_A$):** For each sub-scenario identified within the previous process-step, an abstract process-structure according to an AND/OR net syntax is set up. The nodes of these networks are object constellations (OCs), built up from TPOs. OCs are connected via composed operators (COPs) to describe the action flow on abstract programming level. A new aspect introduced by the FRIEND::Process are pictographic $PS_A$s, which visualize object constellations with the help of pictograms. The required TRE on $PS_A$-level are: System, Items, COPs and Facts. The process step S2a is finished with the specification of Activity-Diagrams for each $PS_A$ ($PS_A$-AD) to exemplify a possible COP sequence in the assigned sub-scenario (usually the most probable action flow).

<table>
<thead>
<tr>
<th>Tools</th>
<th>Rhapsody, DBGUI, PSA-Designer</th>
<th>T1+2+3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repositories</td>
<td>Model Repository, Task Knowledge Database, Process-Structure Repository</td>
<td>R1+2+3</td>
</tr>
<tr>
<td>Artifacts (IN)</td>
<td>TPO, Sub-Scenario Use-Cases</td>
<td>A1.1+2</td>
</tr>
<tr>
<td>Artifacts (OUT)</td>
<td>System, Items, COPs, Facts, $PS_A$s, $PS_A$-ADs</td>
<td>A2.1–6</td>
</tr>
</tbody>
</table>

**S2b - Specification of Elementary Process-Structures ($PS_E$):** The FRIEND::Process introduces special function block networks (FBN) for programming of elementary process-structures. They contain elementary executable operators (EEOPs) and facts as main building blocks. For each COP that resulted from the previous process-step, an FBN is specified. The Items are the COP’s parameters and since an FBN represents system level programming, a certain system as identified before is assigned to it. Besides $PS_E$s, the new development artifacts from this process-step are EEOPs, which are formally introduced in the TRE database. Step 2b finishes with the conversion of an FBN-$PS_E$ into a Petri net equivalent. On Petri net level some remaining verifications take place, which cannot be solved on FBN-level, such as the build-up of a reachability graph to assure the reachability of target states and exclusion of deadlock situations.

<table>
<thead>
<tr>
<th>Tools</th>
<th>DBGUI, PSE-Designer</th>
<th>T2+4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repositories</td>
<td>Task Knowledge Database, Process-Structure Repository</td>
<td>R2+3</td>
</tr>
<tr>
<td>Artifacts (IN)</td>
<td>System, Items, COPs, Facts</td>
<td>A2.1–4</td>
</tr>
<tr>
<td>Artifacts (OUT)</td>
<td>EEOPs, $PS_E$s</td>
<td>A2.7+8</td>
</tr>
</tbody>
</table>
A. Compact Overview on the Process Model "FRIEND::Process"

Process Step 3+4 (S3+4): Skill Development and Test

Based on the task knowledge specification, the elementary functionalities of the system (skills) are developed and tested. For each EEOP from $PSE$-level a skill is developed. This is done in a model-centric approach, supported by the UML CASE tool Rhapsody and further test and visualization tools. Due to executable models, the development is done in an iterative process, with skill simulation and execution from the initial skill design on.

**S3a - Skill Design:** Based on the EEOP specification of step 2b the skill development starts with capturing Skill Use Cases and Requirements. Further on, the Skill Skeleton and its Behavior is set up according to a uniform flowchart template. For each $PS_A$ a Skill Sequence Diagram ($PS_A$-Skill-SD) is created to show a sample sequence of skills as they will be usually executed in a certain sub-scenario. Based on the use cases, the Skill Test Sequences and the environmental information as required for simulative executions of a skill are set up in the Skill Test Data Repository. Test sequences are configured with the Skill Tester and the environmental data is set up with the World Model Browser application. Consequently, the first skill tests can be executed, as indicate by the connecting arrow between S3a and S4a.

<table>
<thead>
<tr>
<th>Tools</th>
<th>Rhapsody, Skill Tester, World Model Browser</th>
<th>T1+5+6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repositories</td>
<td>Task Knowledge Database, Model Repository, Skill Test Data Repository</td>
<td>R2+1+4</td>
</tr>
<tr>
<td>Artifacts (IN)</td>
<td>EEOps</td>
<td>A2.7</td>
</tr>
<tr>
<td>Artifacts (OUT)</td>
<td>Skill-UCs, -Requirements, -Skeleton, -Skeleton-Behavior, $PS_A$-Skill-SDs, Skill Test Sequences, World Model Test Data</td>
<td>A3.1–7</td>
</tr>
</tbody>
</table>

**S3b - Skill Implementation:** In this development step the skill implementation is completed. This takes place in an iterative manner within the skill implementation and test cycle.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Rhapsody</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repository</td>
<td>Model Repository</td>
<td>R1</td>
</tr>
<tr>
<td>Artifacts (IN)</td>
<td>Skill-UCs, -Requirements, -Skeleton, -Skeleton-Behavior</td>
<td>A3.1–4</td>
</tr>
<tr>
<td>Artifacts (OUT)</td>
<td>Complete Skill Behavior</td>
<td>A3.8</td>
</tr>
</tbody>
</table>

**S4a - Skill Test:** Skill test sequences are executed with the Skill Tester. Several skill simulation and execution levels are used, starting with a probabilistic simulation of the skill core functionality to check the inter-module communication. Subsequently, manipulative skills are execution with motion simulation, to inspect in a virtual 3D environment, whether the skill shows the desired behavior. Also, hardware simulation can be performed. Finally, the skill is executed on the target system. The artifacts produced here are Skill Test Sequence Diagrams, which are generated automatically by Rhapsody during skill execution and which are stored in the Model Repository for documentation purposes.

<table>
<thead>
<tr>
<th>Tools</th>
<th>Skill Tester, World Model Browser, Helper Tools, Rhapsody</th>
<th>T5+6+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repositories</td>
<td>Task Knowledge Database, Skill Test Data Repository, Model Repository</td>
<td>R2+4+1</td>
</tr>
<tr>
<td>Artifacts (IN)</td>
<td>EEOPs, Skill Test Sequences, World Model Test Data</td>
<td>A2.7, A3.6+7</td>
</tr>
<tr>
<td>Artifacts (OUT)</td>
<td>Skill Test Sequence Diagrams</td>
<td>A4.1</td>
</tr>
</tbody>
</table>
**S4b - Scenario Test:** The Scenario Test is the final system testing. Here, a complete task as well as sequences of tasks are tested. The Sequencer plans the tasks based on process-structures. Task activation and supervision is done from the SequencerGUI. As in Step 4a the test execution generates sequence diagrams, which are again stored in the Model Repository.

<table>
<thead>
<tr>
<th>Tools</th>
<th>Repositories</th>
<th>Artifacts (IN)</th>
<th>Artifacts (OUT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequencer, SequencerGUI, Rhapsody</td>
<td>Task Knowledge Database, Process-Structure Repository, Model Repository</td>
<td>System, Items, COPs, Facts, EEOPs, $PS : A, PS_E$s</td>
<td>Scenario Test Sequence Diagrams</td>
</tr>
</tbody>
</table>

### A.2. Repositories

- **R1**: Model Repository
- **R2**: Task Knowledge Database
- **R3**: Process-Structure Repository
- **R4**: Skill Test Data Repository

### A.3. Development Artifacts

- **A1.1**: Task participating objects (TPO)
- **A1.2**: Sub-scenario use-cases
- **A2.1**: System specification
- **A2.2**: Item specifications
- **A2.3**: Composed operator (COP) specifications
- **A2.4**: Fact specifications
- **A2.5**: Abstract process-structure ($PS_A$)
- **A2.6**: $PS_A$ activity-diagram
- **A2.7**: Elementary executable operator (EEOP) specifications
- **A2.8**: Elementary process-structure ($PS_E$)
- **A3.1**: Skill use-case specifications
A. Compact Overview on the Process Model "FRIEND::Process"

- A3.2: Skill requirement specifications
- A3.3: Skill skeleton
- A3.4: Skill skeleton behavior
- A3.5: $PS_A$ skill sequence diagrams
- A3.6: Skill test sequences
- A3.7: World model test data
- A3.8: Complete skill behavior
- A4.1: Skill test sequence diagrams
- A4.2: Scenario test sequence diagrams

A.4. Development Tools

- T1: Rhapsody
- T2: DBGUI
- T3+4: PS-Designer with PSA-Designer and PSE-Designer
- T5: SkillTester
- T6: WorldModelBrowser
- T7: Sequencer
- T8: SequencerGUI
- T9: ConfigGUI
Composed Operators (COPs) in AMaRob
Scenario 'Fetch Meal'

In the following, some graphical COP function block networks, also called elementary process-structures (PS\textsubscript{E}s), are presented. Due to their structural similarity, only a subset of networks is given, i.e. those that have not been illustrated before and that are used in the sub-scenario 'Fetch meal', whose abstract process-structure can be found in Figure 9.3 on page 175.

All PS\textsubscript{E}s are constructed, with respect to the shared-control paradigm, according to the following rule: First, an autonomous skill is executed, e.g. a manipulative skill. If this skill returns a Failure, the respective user-interaction skill is activated. In all PS\textsubscript{E}s, the required sensor data has to be acquired with suitable sensorial skills (e.g. machine vision skills). Subsequently, the COPs core functionality can be executed. The different colors of the EEOPs blocks in the networks show the different skill servers like Machine-Vision Skill Server, Manipulator Skill Server or User-Interaction Skill Server.
B. Composed Operators (COPs) in AMaRob Scenario 'Fetch Meal'

B.1. DepartFromContainer(Manipulator, Object, Container)

B.2. GetObjectOutside(Manipulator, Object, Container, InsertLoc)
B.3. MoveObjectToRelLoc(Manipulator, Object, Container, InsertLoc)

B.4. MoveObjectFromRelLoc(Manipulator, Object, Container, InsertLoc)
B. Composed Operators (COPs) in AMaRob Scenario 'Fetch Meal'

B.5. PlaceObjectInside(Manipulator, Object, Container, InsertLoc)
Sample Protocol of Task Execution

In the following the execution protocol for the task *fetch_meal_getoutsideonly* is given. The abstract process-structure related to this task is depicted in Figure 9.3, showing that the retrieval of a meal tray from a fridge is meant. The task’s postfix *getoutsideonly* denotes that the meal tray is not put down on a platform due to subsequent task execution (insertion into microwave oven). The numbers on the left of the execution protocol indicate time stamps. Only seconds and milliseconds are shown to reduce the required page space.

```
13.832: SequencerCore::EnqueueHighLevelCommand: RECEIVE TASK:
        fetch_meal_getoutsideonly

13.835:

SequencerCore: Receive task request: fetch_meal_getoutsideonly

13.836:

SequencerCore: Use task execution preparation: YES
SequencerCore: Use task execution finishing: YES

14.429: SequencerCore: Trying to connect WM server with naming context: WorldModel [Success]

14.433: Adding to ssWM: Fr.1.EEL.ModelingState = CMODELING_STATE_T::DETAILED
14.434: Adding to ssWM: MP.1.EEL.ModelingState = CMODELING_STATE_T::DETAILED
14.434: Adding to ssWM: Mt.1.EEL.ModelingState = CMODELING_STATE_T::DETAILED
14.434: Adding to ssWM: Tr.1.EEL.ModelingState = CMODELING_STATE_T::DETAILED
14.435: SeqCore::SetSit4Planning: IsInsideContainer( Mt.1, Fr.1 ) = True
14.436: SeqCore::SetSit4Planning: HasFreeStoringSpace( Fr.1 ) = False
14.437: SeqCore::SetSit4Planning: ContainerAccessible( Fr.1 ) = True
14.438: SeqCore::SetSit4Planning: HoldsNothing( MP.1 ) = True
14.438: SeqCore::SetSit4Planning: IsInFreePos( MP.1 ) = True

SequencerCore: Task execution preparation
```
C. Sample Protocol of Task Execution

15.985:

Init SkillExecutor

| SkillExecutor: WorldModel:  | Available |
| SkillExecutor: Calculation: | Available |
| SkillExecutor: MachineVision: | Available |
| SkillExecutor: Manipulator:  | Available |
| SkillExecutor: Tray:         | Available |
| SkillExecutor: Microwave:    | Available |
| SkillExecutor: UserInteraction: | Available |

SKILL SIM/EXEC LEVEL: PROBABILITY_SIM

15.986: SkillExecutor::Init: (Re) Init CORBA connections now ...

15.988: InitSkillExecutor: Trying to connect VM server with naming context: WorldModel [Success]

15.990: InitSkillExecutor: Trying to connect Manipulator skill server with naming context: Manipulator [Success]

15.992: InitSkillExecutor: Trying to connect Tray skill server with naming context: Tray [Success]

15.994: InitSkillExecutor: Trying to connect MachineVision skill server with naming context: MachineVision [Success]

15.996: InitSkillExecutor: Trying to connect Calculation skill server with naming context: Calculation [Success]

15.998: InitSkillExecutor: Trying to connect Microwave skill server with naming context: Microwave [Success]

16.000: InitSkillExecutor: Trying to connect UserInteraction skill server with naming context: UserInteraction [Success]

16.724: SkillExecutor::Init: (Re) Init simulation and execution data now ...


23.285: SkillExecutor::InterpretEEOP: Get max exec time from DB: 0 ms

23.285: SkillExecutor::InterpretEEOP: MaxExecTimeout: 120000 ms

23.286: SkillExecutor::InterpretEEOP: SIMULATE SKILL ...

23.286: SkillExecutor::InterpretEEOP: Prepare simulation params: Success 100 Failure 0

23.286: SkillExecutor::SetSkillExecutionModeOnAllSkillServers: PROBABILITY_SIM

23.286: SkillExecutor::SetSkillExecutionModeOnAllSkillServers: Calculation ...

23.287: SkillExecutor::SetSkillExecutionModeOnAllSkillServers: MachineVision ...

23.288: SkillExecutor::SetSkillExecutionModeOnAllSkillServers: Manipulator ...

23.288: SkillExecutor::SetSkillExecutionModeOnAllSkillServers: Tray ...

23.289: SkillExecutor::SetSkillExecutionModeOnAllSkillServers: Microwave ...

23.307: SkillExecutor::SetSkillExecutionModeOnAllSkillServers: UserInteraction ...

23.308: SkillExecutor::InterpretEEOP: Skill execution successfully started 

23.308: SkillExecutor::PollSkillResults: Manipulator.GoToReadyPose CB not updated ...

23.508: SkillExecutor::PollSkillResults: Receiving return val ( Manipulator. GoToReadyPose ): ——> Success

23.508: SkillExecutor::ReleaseCallBackServant: Manipulator.GoToReadyPose: Release CB proxy

23.508: SkillExecutor::ReleaseCallBackServant: Manipulator.GoToReadyPose: Delete CB servant

23.589:
SequencerCore: Start task execution

23.589: SequencerCore: m_pEEOPExecutor->activate()
23.589: SequencerCore: m_pTaskPlanner->activate()
23.593: ProcessHLC: CreateCopsFusedPN
23.631: ProcessHLC: CreateLowLevelPlan
23.648:

Execute COP: GraspObjectInContainer.1( MP.1, Mt.1, Fr.1 )

23.648: Enter ExecuteCopStep
23.649: GraspObjectInContainer.1::IT.SPLIT_0()
23.649: Ignore transition (but fire Eeop): IT.SPLIT_0
23.649: GraspObjectInContainer.1::MachineVision.AcquireObjectBySCam( Bumblebee, Fr .1, Mt.1.EEL.Loc, Mt.1.EEL.SCub )
23.651: Skip Eeop — out data already exists: MachineVision.AcquireObjectBySCam

23.651: GraspObjectInContainer.1::Manipulator.CoarseApproachToObjectInContainer( Robotarm, Fr.1.EEL.Loc, Mt.1.EEL.SCub, Mt.1.EEL.SCub )
23.654: Planner: ExecuteCopStep: Manipulator.CoarseApproachToObjectInContainer
23.654: CTaskPlanner::PollAndHandleExecResult: Waiting for 1 parallelly executed Eeops

23.659: SkillExecutor::InterpretEEOP: Getting from planner: GraspObjectInContainer .1::Manipulator.CoarseApproachToObjectInContainer( Robotarm, Fr.1.EEL.Loc, Fr.1.EEL.SCub, Mt.1.EEL.Loc, Mt.1.EEL.SCub )
23.659: Planner: ExecuteCopStep: Manipulator.CoarseApproachToObjectInContainer
23.654: CTaskPlanner::PollAndHandleExecResult: Waiting for 1 parallelly executed Eeops

23.660: SkillExecutor::InterpretEEOP: Skill execution successfully started ...
23.667: SkillExecutor::PollSkillResults: Manipulator.
23.656: SkillExecutor::PollSkillResults: Receiving return val (Manipulator.
23.656: CoarseApproachToObjectInContainer CB not updated.
23.656: SkillExecutor::PollSkillResults: Receiving return val (Manipulator.
23.656: CoarseApproachToObjectInContainer): ---> Success
23.656: SkillExecutor::ReleaseCallBackServant: Manipulator.
23.656: CoarseApproachToObjectInContainer: Release CB proxy
23.656: SkillExecutor::ReleaseCallBackServant: Manipulator.
23.656: CoarseApproachToObjectInContainer: Delete CB servant
23.656: EEOPExecutor::ProcessReceivedEeop: Send EEOP back to planner: Manipulator.
23.656: CoarseApproachToObjectInContainer
23.657: EEOPExecutor::ProcessReceivedEeop: Enqueue message to task planner: 10: Success
23.154: CTaskPlanner::PollAndHandleExecResult: 10: Success

23.155: GraspObjectInContainer.1::IT.Trigger_20( )
23.155: Ignore transition (but fire Eeop): IT.Trigger_20
23.156: GraspObjectInContainer.1::UserInteraction.AdjustToObject( Robotarm, Mt.1 )
23.156: Planner: ExecuteCopStep: UserInteraction.AdjustToObject
23.156: CTaskPlanner::PollAndHandleExecResult: Waiting for 1 parallelly executed Eeops
23.159: SkillExecutor::InterpretEEOP: Getting from planner: GraspObjectInContainer .1::UserInteraction.AdjustToObject( Robotarm, Mt.1 ). Planned return value: Success
C. Sample Protocol of Task Execution

26.88: SkillExecutor::PollSkillResults: Receiving return val (UserInteraction. AdjustToObject): —> SystemTakeOver
26.89: SkillExecutor::ReleaseCallBackServant: UserInteraction. AdjustToObject: Delete CB servant
26.89: EEOExecutor::ProcessReceivedEeop: Send EEO back to planner: UserInteraction. AdjustToObject
26.89: EEOExecutor::ProcessReceivedEeop: Enqueue message to task planner: 16: SystemTakeOver
26.86: TaskPlanner::SendStopExecMsg (MB_USER) to EEOExecutor
26.86: Quit ExecuteCopStep
26.89: ProcessHLC::CreateLowLevelPlan
26.89: Execute COP: GraspObjectInContainer.1( MP.1, Mt.1, Fr.1 )
26.89: Enter ExecuteCopStep
26.89: GraspObjectInContainer.1:: Manipulator.FineApproachToObjectInContainer( Robotarm, Fr.1.EEL.Loc, Fr.1.EEL.SCub, Mt.1.EEL.Loc, Mt.1.EEL.SCub ). Planned return-value: Success
26.89: SkillExecutor::InterpretEeop: Getting from planner: GraspObjectInContainer.1:: Manipulator.FineApproachToObjectInContainer( Robotarm, Fr.1.EEL.Loc, Fr.1.EEL.SCub, Mt.1.EEL.Loc, Mt.1.EEL.SCub ). Planned return-value: Success
26.44: SkillExecutor::PollSkillResults: Receiving return val (Manipulator. FineApproachToObjectInContainer): —> Success
26.44: SkillExecutor::ReleaseCallBackServant: Manipulator. FineApproachToObjectInContainer: Release CB proxy
26.44: SkillExecutor::ReleaseCallBackServant: Manipulator. FineApproachToObjectInContainer: Delete CB servant
26.44: EEOExecutor::ProcessReceivedEeop: Send EEO back to planner: Manipulator. FineApproachToObjectInContainer
26.44: EEOExecutor::ProcessReceivedEeop: Enqueue message to task planner: 19: Success
26.48: CTaskPlanner::PollAndHandleExecResult: 19: Success
26.48: GraspObjectInContainer.1:: Manipulator.CloseGripper( Gripper )
26.48: Planner::ExecuteCopStep: Manipulator.CloseGripper
26.48: CTaskPlanner::PollAndHandleExecResult: Waiting for 1 parallely executed Eeops
26.53: SkillExecutor::InterpretEeop: Getting from planner: GraspObjectInContainer.1:: Manipulator.CloseGripper( Gripper ). Planned return-value: Success
[...]
SequenceCore: Task execution finishing
C. Sample Protocol of Task Execution

29.067: SkillExecutor::InterpretEEOP: Getting from planner: ::Manipulator.
GoToReadyPose(Robotarm, fetch_meal_getoutsideonly.FINISH). Planned return-value: Success

29.273: SkillExecutor::PollSkillResults: Receiving return val (Manipulator.
GoToReadyPose): -> Success

29.273: SkillExecutor::ReleaseCallbackServant: Manipulator.GoToReadyPose: Release CB
proxy

29.273: SkillExecutor::ReleaseCallbackServant: Manipulator.GoToReadyPose: Delete CB
servant

29.378: SequencerCore: Successfully finished task execution. Changing to READY STATE
D.1. Extraction of Valid Situations

The necessary prerequisite for task planning based on process-structures is the determination of the initial situation $Sit_I$ in a given process-structure ($PS$). This determination process is done on the base of the set of valid situations $S_{Sit}$ within $PS$. The situations from $S_{Sit}$ are potential candidates for $Sit_I$. However, to determine $S_{Sit}$ the following assumptions have to be true:

Each situation in $PS$ has to contain all items of $PS$, i.e.

$$\forall Sit \in S_{Sit} : \text{Items}(Sit) = S_I \quad (D.1)$$

Furthermore, the reachability of $Sit_T$ within $PS$ via an arbitrary but finite sequence $\text{Seq}_{OP} = \{\{OP_0, OP_1, \ldots, OP_m\}| OP_i \in S_{OP}, OP_i \neq OP_j, i,j = 1..m, i \neq j\}$ with $S_{OP}$ as set of all operations that are used in one process-structure has to be guaranteed for all $Sit \in S_{Sit}$. The verification of reachability and validity of situations can be done on the base of a situation graph $SG$ that covers the whole process-structure. A sample situation graph for a process-structure for the task *Pour-in a drink* is depicted in Figure D.1 and the flowchart of the algorithm to expand $SG$ is shown in Figure D.2. The algorithm takes as inputs one potential candidate situation $Sit_{pot}$ that is already valid according to Eq. D.1 as well as the set of ICs$^1$ $S_{IC}$ and the set of operations $S_{OP}$. First $Sit_{pot}$ is copied to $S_{Sit}$ that has been empty so far. Subsequently, all preceding and descending arcs from the current amount of situations (initially only $Sit_{pot}$) are expanded (i.e. operations from $S_{OP}$ are taken) and new

$^1$IC = item constellation, is the same as OC (object constellation).
D. Situation Graph

Figure D.1. Situation graph for process-structure *Pour-in a drink.*

situations are added to $S_{Sit}$. If no ICs with $IC \in S_{IC}$ remain unconsidered in $S_{Sit}$ after the complete expansion, a valid situation graph was found and all valid situations are contained in $S_{Sit}$. To make planning reasonable

$$Sit_i \neq Sit_T$$

has to be assumed. That means, $S_{Sit}$ is reduced for $Sit_T$ and the reasonable candidate situations for $Sit_i$ are contained in $S_{Sit,cand} = S_{Sit} \setminus Sit_T$.

D.2. Determination of Unique Sets of Facts

The determination of $Sit_i \in S_{Sit,cand}$ can now be done on the base of the facts that are assigned to the ICs contained in each candidate situation. A process-structure that is valid for monitoring therefore has to provide unique sets of facts for each situation. This means, the following has to be valid, assuming that $S_{Fact}(Sit_i)$ is the set of facts for one situation:

$$\forall Sit_i \in S_{Sit,cand} : \neg \exists Sit_j | S_{Fact}(Sit_i) = S_{Fact}(Sit_j), Sit_i \neq Sit_j$$

This verification is done during offline-programming of a process-structure an thus only verified process-structures are admitted for task planning respectively initial monitoring.
D.2. Determination of Unique Sets of Facts

Figure D.2. Flowchart of the algorithm to expand $SG$. 

---

$S_{IC}$, $S_{COP}$, $S_{pot}$

Start

$\forall S_{sit} \in S_{sit}$

$\forall IC \in S_{it}$

ExpandAllPrecedentArcs

ExpandAllDescendantArcs

$S_{IC} = S_{IC} \setminus \{IC\}$

$S_{sit} = S_{sit} \setminus \{S_{new}\}$

$S_{sit} = S_{sit} \cup \{S_{new}\}$

End
A sample manually drawn Petri net is shown here (for the COP *GraspObjectInContainer*) to show the complexity and loss of clarity even in rather simple networks. The Petri net shown here represents the same elementary process-structure as the function block network in Figure 6.51 on page 125. The Petri net has been composed with the tool HP-Sim [Anschuetz, 2009].
Figure E.1. Manually drawn Petri net GraspObjectInContainer.


Bibliography


Bibliography


Bibliography


# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Statistics on service robots for professional use.</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Statistics on service robots for personal/domestic use.</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>FRIEND III rehabilitation robot.</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Use case diagram of ADL support scenario <em>Preparation of a meal and eating</em></td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Special meal tray for handling a meal with a robot.</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>ADL sample scenario: The manipulator fetches an object from the refrigerator.</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>Use case diagram with library sub-scenarios.</td>
<td>14</td>
</tr>
<tr>
<td>2.6</td>
<td>Use case diagram with sub-scenarios in the workshop environment.</td>
<td>15</td>
</tr>
<tr>
<td>2.7</td>
<td>(a) Magazine for telephone keypads, (b) Test adapter for telephone keypads.</td>
<td>15</td>
</tr>
<tr>
<td>2.8</td>
<td>Overview on task processing based on process-structures.</td>
<td>16</td>
</tr>
<tr>
<td>2.9</td>
<td>Abstract process-structure related to the task <em>FetchCup</em>.</td>
<td>17</td>
</tr>
<tr>
<td>2.10</td>
<td>Elementary process-structure of COP <em>GraspObjectInContainer</em> as Petri net.</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>Typical defect introduction and detection in traditional software development.</td>
<td>25</td>
</tr>
<tr>
<td>3.2</td>
<td>Traditional way of software development.</td>
<td>26</td>
</tr>
<tr>
<td>3.3</td>
<td>SysML diagram taxonomy.</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>Example how to express a concept via model diagram.</td>
<td>27</td>
</tr>
<tr>
<td>3.5</td>
<td>Model Driven Development of software for embedded systems - an emerging trend.</td>
<td>29</td>
</tr>
<tr>
<td>3.6</td>
<td>Integrated Systems/Software Development Process HARMONY.</td>
<td>30</td>
</tr>
<tr>
<td>3.7</td>
<td>HARMONY-SE overview.</td>
<td>31</td>
</tr>
<tr>
<td>3.8</td>
<td>Functional analysis workflow.</td>
<td>32</td>
</tr>
<tr>
<td>3.9</td>
<td>Building the use case model.</td>
<td>32</td>
</tr>
<tr>
<td>3.10</td>
<td>SysML artifacts utilized in functional analysis.</td>
<td>33</td>
</tr>
<tr>
<td>3.11</td>
<td>System architecture design workflow.</td>
<td>34</td>
</tr>
<tr>
<td>3.12</td>
<td>Sample Task Tree.</td>
<td>37</td>
</tr>
<tr>
<td>3.13</td>
<td>Visual Design Tool.</td>
<td>38</td>
</tr>
<tr>
<td>3.14</td>
<td>Sample RoboGlyph Storyboard (&quot;Open a refrigerator door&quot;).</td>
<td>39</td>
</tr>
<tr>
<td>3.15</td>
<td>Sample Microsoft Visual Programming Language (VPL) diagram.</td>
<td>40</td>
</tr>
<tr>
<td>3.16</td>
<td>Function block according to IEC 61131-3.</td>
<td>43</td>
</tr>
<tr>
<td>4.1</td>
<td>Scheme of process model &quot;FRIEND::Process&quot;.</td>
<td>49</td>
</tr>
<tr>
<td>4.2</td>
<td>Process model steps of scenario analysis.</td>
<td>50</td>
</tr>
<tr>
<td>4.3</td>
<td>Kitchen environment with FRIEND III and distributed smart components.</td>
<td>51</td>
</tr>
<tr>
<td>4.4</td>
<td>Process model steps of task knowledge specification and verification.</td>
<td>52</td>
</tr>
<tr>
<td>4.5</td>
<td>Development artifacts of abstract process-structure specification.</td>
<td>53</td>
</tr>
<tr>
<td>4.6</td>
<td>Scheme of a pictographic representation of an abstract process-structure.</td>
<td>54</td>
</tr>
<tr>
<td>4.7</td>
<td>Products (development artifacts) of $PS_E$ programming.</td>
<td>56</td>
</tr>
<tr>
<td>4.8</td>
<td>$PS_E$ <em>GraspObjectInContainer</em> as function block network.</td>
<td>57</td>
</tr>
<tr>
<td>4.9</td>
<td>FRIEND::Architecture (also called MASSiVE architecture).</td>
<td>59</td>
</tr>
<tr>
<td>4.10</td>
<td>Skill execution from Sequencer.</td>
<td>60</td>
</tr>
</tbody>
</table>
### List of Figures

4.11. Reactive layer for FRIEND with intelligent environment. ........................................... 61
4.12. Process model steps for skill development and testing. ............................................. 63

5.1. Process Step 1a: Scenario Specification. ................................................................. 67
5.2. Products of scenario specification. ............................................................................. 68
5.3. Hierarchical ontology of task participating objects in AMaRob scenarios. ................. 70
5.4. Process Step 1b: Sub-Scenario Specification. ......................................................... 71

6.2. Activity diagram that defines the procedure of abstract process-structure specification. 75
6.3. Activity diagram of the specification procedure of task participating objects. ............. 76
6.4. Activity diagram of system specification procedure. .................................................. 76
6.5. Activity diagram of item specification procedure. ........................................................ 77
6.6. Activity diagram of COP specification procedure. ..................................................... 79
6.7. Activity diagram of fact specification procedure. ....................................................... 80
6.8. Activity diagram of graphical object constellation specification. .............................. 82
6.9. Activity diagram for adding a COP to a graphical abstract process-structure. ........... 83
6.10. Process of situation assignment in abstract process-structure. ................................ 84
6.11. Tools for specification of pictographic abstract process-structures. ......................... 86
6.12. Hierarchy of interface classes to access the task representing elements. ...................... 88
6.13. DBGUI with fact management view. ....................................................................... 88
6.14. DBGUI design overview. .......................................................................................... 89
6.15. DBGUI widget for COP management. ................................................................. 90
6.16. Architectural design to realize COP management. ................................................... 91
6.17. DBGUI widget for item management. ..................................................................... 92
6.18. DBGUI widget for systems management. ............................................................... 93
6.19. Activity diagrams for EEOP specification procedure. ............................................... 94
6.20. DBGUI widget for EEOP management. ................................................................. 96
6.21. DBGUI dialog for setting up sub-symbolic parameters. ........................................... 97
6.22. PSA-Designer use cases. .......................................................................................... 98
6.23. Refinement of use case $EditOC$. ............................................................................ 99
6.24. Preview on PSA-Designer GUI with pictographic $PS_A$ Fetch Bottle. ..................... 100
6.25. Large scale design of manager classes for $PS_A$ modeling. ................................ 101
6.26. Data structures that are managed by PSALogicManager. ........................................ 102
6.27. Data structures that are managed by EnvNSys. ....................................................... 103
6.28. Architectural design of PSA logic manager and related classes. ............................. 103
6.29. Core data structures for (picto)graphic $PS_A$ modeling. ...................................... 105
6.30. 3D visualization-related data structures for pictographic $PS_A$ modeling. ............. 105
6.31. Architectural design of graphic manager. ............................................................... 106
6.32. Statechart for 2D scene. ........................................................................................ 107
6.33. Statechart of graphic manager. .............................................................................. 108
6.34. Statechart of 3D object constellation manager. .................................................... 109
6.35. Dialog for selection of task participating objects (TPO). ......................................... 110
6.36. 3D scene for editing an object constellation. .......................................................... 110
6.37. COP selection and connection of pictographic object constellations. ...................... 111
6.38. Complete pictographic $PS_A$ with situation navigator and initial and target situations. 112
6.40. Process of $PS_E$ function block network specification as activity diagram. ............ 114
6.41. Hierarchy of function block network elements. ...................................................... 114
6.42. Activity diagram of adding procedure for EEOP blocks. ........................................... 115
6.43. Procedure of adding a link in a function block network. ........................................ 117
6.44. Verification and saving of a $PSE$. ................................................................. 118
6.46. Main use cases of PSE-Designer. ................................................................. 120
6.47. Large Scale Design of PSE-Designer. ....................................................... 121
6.48. $PSE$ model manager with FBN data structures ........................................... 122
6.49. Core data structures and controller for graphical layer. ................................ 123
6.50. Simplified statechart of PSE-Designer. ....................................................... 124
6.51. PSE-Designer front-end with simplified version of COP $GraspObjectInContainer$ .... 125
6.52. Class for function block to Petri net conversion and related classes ................ 126
6.53. Flowchart of the complete conversion algorithm. .......................................... 128
6.54. Flowchart for generating and linking EEOP transitions .................................. 129
6.55. (a) Fact generation, (b) AND generation, (c) OR generation ......................... 129
7.1. Process Step 3a: Skill Design. ........................................................................ 134
7.2. Development artifacts of skill design procedure. ............................................ 134
7.3. Skill use cases and requirements for the skill $CoarseApproachToObjectInContainer$ .... 135
7.4. Parameter Manager and related classes ....................................................... 138
7.5. Top-level flowchart of skill skeleton for the skill $CoarseApproachToObjectInContainer$ .......... 139
7.6. Main infrastructure flowchart of a skill skeleton ............................................ 140
7.7. Example for preparation of proxy objects. .................................................... 141
7.8. Sample flowchart for extraction of sub-symbolic data from the world model ......... 142
7.9. Flowchart to handle standard incoming skill messages STOP and KILL ............ 143
7.10. Top level sequence diagram for the skill $CoarseApproachToObjectInContainer$ .... 144
7.11. Sub-sequence diagram for the action $CheckNPReapreConnections$ ............... 145
7.12. Sub-sequence diagram for the action $SetAndClearWorldModelData$ ............. 146
7.13. Sequence diagram with possible skill sequence for fetching a meal from a fridge .... 147
7.15. Process Step 3b: Skill Implementation. ....................................................... 150
7.16. First level of main implementation of skill $CoarseApproachToObjectInContainer$ .... 151
8.1. Process Step 4a: Skill Testing. ......................................................................... 153
8.2. Tools for skill testing. .................................................................................. 154
8.3. Requirements for skill tester tool. ................................................................. 155
8.4. Overview on design of skill tester. ............................................................... 156
8.5. Screenshot of skill tester. ............................................................................. 157
8.6. Screenshot of world model browser. ............................................................ 158
8.7. Process Step 4b: Scenario Testing. ............................................................... 159
8.8. Tools for scenario testing. ............................................................................ 160
8.9. Large scale architectural design of the Sequencer. ......................................... 161
8.10. Overview on architectural design of task planner. ........................................ 162
8.11. Overview on architectural design of skill execution modules. ......................... 163
8.12. Architectural design of CORBA skill executor. ............................................. 164
8.13. Task processing in Sequencer, first part. ..................................................... 166
8.14. Task processing in the Sequencer, second part. .......................................... 167
9.1. Sample products of scenario specification. .................................................... 170
9.2. Use case diagram of ADL support scenario Preparation of a meal and eating .... 171
9.3. Sub-scenario 1: Fetch meal from fridge. ....................................................... 175
9.4. Sub-scenario 2: Cook meal. ......................................................................... 177
List of Figures

9.5. Sub-scenario 4: Take away lid. ..................................................... 180
9.6. Sub-scenario 5: Meal eating support. ............................................. 181
9.7. Sub-scenario 6: Take away mealtray. ............................................. 183
9.8. Function block network for COP GraspObjectInContainer. ................. 186
9.9. Screenshots of motion simulation. ................................................... 191
9.10. Animated sequence diagram, resulting from simulative skill execution. .......... 192
9.11. Automatic comparison of sequence diagrams. .................................. 194
A.1. Scheme of process model "FRIEND::Process" .................................. 206
D.1. Situation graph for process-structure Pour-in a drink. .......................... 222
D.2. Flowchart of the algorithm to expand SG. ........................................... 223
E.1. Manually drawn Petri net GraspObjectInContainer. ............................ 226
List of Tables

1.1. Number of expanded nodes and calculation time in breadth-first graph search. . . . . . 5
2.1. Composed operators of the Fetch-cup process-structure. . . . . . . . . . . . . . . . . . . 17
3.1. SysML artifacts in functional analysis. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
4.1. Summarized process requirements. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
4.2. Summarized tool requirements. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
6.1. Examples of sub-symbolic data. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
6.2. Counterpart elements of FBN and PN representation. . . . . . . . . . . . . . . . . . 126
9.1. Sub-scenario specifications for ADL scenario. . . . . . . . . . . . . . . . . . . . . . . 172
9.2. System specification of FRIEND-III. . . . . . . . . . . . . . . . . . . . . . . . . . . . 173
9.3. Further TPO in ADL scenario. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 174
9.16. Complete list of COP prototypes in AMaRob ADL scenario. . . . . . . . . . . . . . . . . 185
9.17. Complete list of EEOP prototypes in AMaRob ADL scenario. . . . . . . . . . . . . . . . . 188
9.18. Sample skill test sequence for grasping a meal tray (Mt.1) in a fridge (Fr.1). . . . . . 190
9.19. World model test data for grasping a meal tray (Mt.1) in a fridge (Fr.1). . . . . . . 190