Copyright Statement

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

This thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author’s right to be identified as the author of this thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from their thesis.

To request permissions please use the Feedback form on our webpage. [http://researchspace.auckland.ac.nz/feedback](http://researchspace.auckland.ac.nz/feedback)

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the [Library Thesis Consent Form](http://researchspace.auckland.ac.nz) and [Deposit Licence](http://researchspace.auckland.ac.nz).

Note : Masters Theses

The digital copy of a masters thesis is as submitted for examination and contains no corrections. The print copy, usually available in the University Library, may contain corrections made by hand, which have been requested by the supervisor.
On Migration of Scan Cycle Based PLC Programs to Distributed Component-Based Event Driven Software Architecture with Semantic Correctness Assurance

Wenbin Dai

Supervisor: A/Prof. Valeriy Vyatkin
Co-Supervisor: Dr. Partha Roop
July 2012

Abstract

In current automation world, majority of systems are designed using programmable logic controllers (PLC) under the IEC 61131-3 standard. The IEC 61131-3 standard PLCs are struggling with increasing demand for reconfigurability and flexibility in distributed control systems. The IEC 61499 standard is considered as the key of enabling distributed and intelligent control into industrial automation. However, the use of the IEC 61499 standard in the automation industry is still minimal. Although advantages of replacing legacy systems with function block controlled systems are substantial, the learning curve is quite steep and the cost of required research and development is high. Introducing the IEC 61499 standard into the IEC 61131-3 based systems provides flexibility and reconfigurability as well as a better graphical view of system designs. Therefore it is important to provide an easy migration path for existing PLC programs into IEC 61499 compliant platforms as the first step towards widespread adoption of the new standard.

This research presented in this thesis proposes a new methodology of migration from IEC 61131-3 PLCs to IEC 61499 function blocks. The aim of this migration process is to recreate IEC 61131-3 applications in IEC 61499 implementations with equivalent execution behaviour. The formal model of the IEC 61131-3 standard and formal cyclical execution model is included. This method also creates a foundation for correct-by-design development tools and automatic migration between the IEC 61131-3 and the IEC 61499 standards. Formal migration rules based on ontology mappings, restoring execution model including tasks and programs scheduling and variables mapping with different access levels are also provided. A transformation engine for import PLC code in XML format, mapping from PLC ontology model to Function Block model and code generation is implemented based on an ontological knowledge base and semantic query-enhanced web rule language. This research also proposes a new approach for semantic analysis using multiple-layered ontological knowledge base and rule-based configurable engine. The semantic rules of the IEC 61499 standard are proposed.
Acknowledgements

First of all, I would like to express my acknowledgement to my supervisor Prof. Valeriy Vytakin. Without his invaluable ideas and suggestions, constant encouragement and guidance, it will be impossible for me to complete the research and thesis.

Secondly, I would like to thank Dr. Partha Roop, my co-supervisor, shares his experiences with me.

Also I want to give my acknowledgement to Dr. Victor Dubinin who is the co-author of several my research papers. He provided me great helps on the semantic analysis and the formal modelling techniques.

I would appreciate Bernie Meredith, former high-level control manager from baggage handling system integrator Glidepath limited. He shared lots of useful ideas for the distributed control systems and experiences in distributed supercomputer applications.

I also give my appreciation to senior software engineer Ian Meyle from Glidepath Limited as well. He taught me lots of baggage handling systems knowledge as well as proper PLC programming techniques.

I’d like also to thank all members from our research group – Cheng Pang, Dr. John Yang, Jeffrey Yan, Gulnara Zhabelova and Majid Sorouri. The discussions between research seminars are really helpful. The practise seminars are helpful for improving my presentation skill.

Finally, very special thanks go to my family, especially my wife Weimin Lu for their constant support through my whole study.
Contents

1. Introduction ............................................................................................................................................................................. 1
   1.1 Introduction to PLC, IEC 61131-3 Standard and Challenges .......................................................... 2
   1.2 Introduction to Components-based Architecture and IEC 61499 Standard ..................... 5
   1.3 Motivation of Migration from PLC to Component-Based Architecture Control .................. 7
   1.4 Migration Process Overview .................................................................................................................. 9
   1.5 Contribution of the Research .................................................................................................................. 11
   1.6 Thesis Structure ........................................................................................................................................ 12
2. Literature Review ....................................................................................................................................................................... 14
   2.1 IEC 61131-3 PLC Design Patterns ........................................................................................................... 16
   2.2 IEC 61131-3 PLC Design Recovery ...................................................................................................... 18
   2.3 Component-Based Architecture Using IEC 61499 Function Blocks ........................................... 20
   2.4 IEC 61499 Reconfiguration and Code Generation ...................................................................................... 22
   2.5 Existing Migration Approaches .............................................................................................................. 24
   2.6 IEC 61499 Execution Semantics ............................................................................................................ 26
   2.7 Ontological Knowledge Base and Ontology Mapping ........................................................................... 28
   2.8 Summary ....................................................................................................................................................... 30
3. Redesign Patterns ........................................................................................................................................................................ 32
   3.1 Introduction ....................................................................................................................................................... 33
   3.2 Object-Oriented Design Pattern ................................................................................................................... 35
      3.2.1 Redesign Finite State Machine to Basic Function Block ECC ........................................... 35
      3.2.2 Reuse PLC code in Basic Function Block Algorithm .......................................................... 39
   3.3 Class-Oriented Design Pattern ...................................................................................................................... 41
   3.4 Data Handling in the Redesign Process .............................................................................................................. 43
      3.4.1 Data Handling in Object-Oriented Approach .............................................................................. 46
      3.4.2 Data Handling in Class-Oriented Approach ................................................................................. 47
   3.5 Redesign Approaches Performance Estimation ....................................................................................... 49
   3.6 Comparison between Approaches and Limitations ................................................................................. 52
   3.7 Summary ....................................................................................................................................................... 53
4. Migration Models and Rules ..................................................................................................................................................... 55
   4.1 Introduction ....................................................................................................................................................... 55
8  Case Study and Results.................................................................149
  8.1  Case Study Example on Baggage Handling System..................149
  8.2  Results from Redesign Approaches for the BHS......................151
    8.2.1  Object-Oriented Method Results........................................152
    8.2.2  Class-Oriented Method Results...........................................154
  8.3  Automatic Migration Results of the BHS.................................156
  8.4  Examples on Migration between IEC 61499-Compliant Tools........159
  8.5  Semantic Analysis Results of the BHS.......................................161
  8.6  Performance Measurement.....................................................163
  8.7  Summary................................................................................165
9  Conclusions and Future Work ....................................................166
  9.1  Efficiency and Impact of the proposed migration method...........166
  9.2  Conclusions.............................................................................169
  9.3  Extension to the SQWRL..........................................................171
  9.4  Recommendations and Future Work..........................................173
Appendix A  IEC 61499 Background Information...............................174
Appendix B  Ontology Definitions....................................................177
  Ontology Definition.......................................................................177
  Description Logic Notation Definition...........................................179
Appendix C  Publications based on this Thesis.................................180
References.....................................................................................181
List of Figures

FIGURE 1.1: MIGRATION BETWEEN IEC 61131-3 TOOLS AND IEC 61499 TOOLS ......................... 9
FIGURE 1.2: MIGRATION BETWEEN IEC 61131-3 TOOLS AND IEC 61499 TOOLS USING KNOWLEDGE
BASE .............................................................................................................. 10
FIGURE 1.3: THESIS STRUCTURE DIAGRAM ........................................................................ 12
FIGURE 3.1: PLC NETWORK DESIGN FOR CENTRALIZED AND DISTRIBUTED SYSTEM ........ 33
FIGURE 3.2: REDESIGN PROCESS FOR THE OBJECT-ORIENTED RE-DESIGN PATTERN ........ 38
FIGURE 3.3: REDESIGN PROCESS FOR THE OBJECT-ORIENTED REUSE PATTERN .......... 39
FIGURE 3.4: REDESIGN PROCESS FOR THE CLASS-ORIENTED PATTERN ......................... 42
FIGURE 3.5: A) IEC 61131-3 DATA FLOW; B) DIRECT MAPPING TO IEC 61499; C) PUBLISH/SUBSCRIBE
IEC 61499 VERSION. [82] .................................................................................... 45
FIGURE 3.6: PERFORMANCE COMPARISON BETWEEN IEC 61131-3 AND IEC 61499 .............. 49
FIGURE 3.7: PERFORMANCE ESTIMATION BETWEEN IEC 61131-3 AND IEC 61499 ............ 51
FIGURE 4.1: IEC 61131-3 SOFTWARE MODEL [1] ................................................................ 56
FIGURE 4.2: IEC 61131-3 EXECUTION SEMANTICS TIMING DIAGRAM ................................. 58
FIGURE 4.3: IEC 61499 SCHEDULER SYSTEM FOR IEC 61131-3 EXECUTION SEMANTICS .... 63
FIGURE 4.4: MAIN SCHEDULER INTERFACE, EXECUTION CONTROL CHART (ECC) AND SCHEDULING
ALGORITHMS (FOR PREEMPTIVE PLC EXECUTION SEMANTICS) ....................... 64
FIGURE 4.5: TASK SCHEDULER INTERFACE, ECC AND ALGORITHMS .............................. 66
FIGURE 4.6: REUSE PLC CODE IN IEC 61499 FUNCTION BLOCKS FOR LD, IL AND ST ........ 70
FIGURE 4.7: IEC 61131-3 SFC MAPPING TO IEC 61499 ECC ........................................... 71
FIGURE 4.8: IEC 61131-3 FBD MAPPING TO IEC 61499 COMPOSITE FB ......................... 72
FIGURE 4.9: GLOBAL/LOCAL VARIABLE COMPOSITE FUNCTION BLOCK [82] .................. 73
FIGURE 5.1: LAYERED IEC 61499 ONTOLOGY DESIGN .................................................. 77
FIGURE 5.2: FLEXIBLE SEMANTIC ANALYSIS USING LAYERED ONTOLOGY ..................... 78
FIGURE 5.3: IEC 61499 ONTOLOGY MODEL OVERVIEW ............................................... 82
FIGURE 5.4: IEC 61499 DATA PROPERTY OF FBTYpe .................................................... 84
FIGURE 5.5: IEC 61499 OBJECT PROPERTY OF FBTYpe ................................................ 85
FIGURE 5.6: FBTYpe CONCEPT ONTOLOGY DIAGRAM ................................................ 86
FIGURE 5.7: SYSTEM CONCEPT ONTOLOGY DIAGRAM ................................................. 86
FIGURE 5.8: DEVICE CONCEPT ONTOLOGY DIAGRAM ................................................. 87
FIGURE 5.9: BASICFB CONCEPT ONTOLOGY DIAGRAM ................................................ 87
FIGURE 5.10: EXAMPLE OF OWL INDIVIDUAL FOR THE ESTOP FBTYpe ......................... 90
FIGURE 5.11: IEC 61131-3 ROCKWELL OBJECT PROPERTY EXAMPLE OF TASK ................ 91
FIGURE 5.12: IEC 61131-3 ROCKWELL DATA PROPERTY EXAMPLE OF TASK NAME .......... 92
FIGURE 5.13: TASK CONCEPT ONTOLOGY DIAGRAM – ROCKWELL VERSION ................... 92
FIGURE 5.14: EXAMPLE OF OWL INDIVIDUAL FOR THE PLC TASK WHICH HAS A PROGRAM .................................................................................................................. 93
FIGURE 5.15: IEC 61131-3 PLCOPEN OBJECT PROPERTY EXAMPLE OF TASK ................. 94
FIGURE 5.16: TASK CONCEPT ONTOLOGY DIAGRAM – PLCOPEN VERSION ................... 95
FIGURE 5.17: EXAMPLE OF OWL INDIVIDUAL FOR THE PLC TASK WHICH HAS A POU .......... 96
FIGURE 6.1: MIGRATION PROCESS FROM IEC 61131-3 PLCs TO IEC 61499 FBs ............... 98
FIGURE 6.2: KNOWLEDGE BASE DEFINITION FOR MIGRATION MAPPING RULES ............ 101
FIGURE 6.3: STEP ONTOLOGY INDIVIDUAL TEMPLATE ................................................. 102
FIGURE 6.4: ACTION ONTOLOGY INDIVIDUAL TEMPLATE ............................................ 102
FIGURE 6.5: SOURCE ONTOLOGY INDIVIDUAL TEMPLATE ........................................... 102
FIGURE 6.6: SCRIPT ONTOLOGY INDIVIDUAL TEMPLATE ............................................ 103
FIGURE 6.7: TEMPLATE ONTOLOGY INDIVIDUAL TEMPLATE ....................................... 103
FIGURE 6.8: QUERY ONTOLOGY INDIVIDUAL TEMPLATE ............................................ 104
FIGURE 6.9: IEC 61131-3 OBJECT PROPERTY EXAMPLE OF TASK .................................. 105
FIGURE 6.10: IEC 61131-3 DATA PROPERTY EXAMPLE OF TASK NAME ............................ 105
Figure 6.11: Example of OWL individual for the PLC task which has a program. .......... 106
Figure 6.12: Step example of mapping PLC project to system configuration. ................. 109
Figure 7.1: DTD, textual and SWRL add-on rule syntax for basic FB definition. ............ 122
Figure 7.2: DTD, textual and SWRL add-on rule for EC action definition. .................... 123
Figure 7.3: DTD, textual and ontology syntax for event connection definition. ............... 124
Figure 7.4: Graphical view of FBT type SQWRL search for basic FB. ............................ 128
Figure 7.5: Dangerous event loop and short-circuit event. .................................... 137
Figure 7.6: Event flow through a basic FB. .......................................................... 138
Figure 7.7: Event flow through a SIFB. ............................................................... 139
Figure 7.8: Screenshot of a system configuration semantic analysis using the developed
semantic analysis tool. ................................................................. 145
Figure 8.1: The case study baggage handling system. ............................................. 149
Figure 8.2: The physical layout of the selected baggage handling subsystem. ............... 150
Figure 8.3: Conveyor control function converted in IEC 61499. ................................. 153
Figure 8.4: BHS example oo version in IEC 61499. .............................................. 154
Figure 8.5: BHS example co version in IEC 61499. .............................................. 155
Figure 8.6: Original PLC configuration structure. ................................................. 156
Figure 8.7: Generated function block configuration structure and main PLC scheduler
FB algorithm. .......................................................... 157
Figure 8.8: The overview of the transformed inbound system. .................................... 159
Figure 8.9: The transformed inbound system for nxtStudio. ................................. 160
Figure 8.10: Original and corrected ECC of basic function block conveyor control. .... 161
Figure 8.11: Dangerous event loop found in the emergency stop control part. ............ 162
Figure 8.12: Test results with different configuration. ......................................... 164
Figure 8.13: Time estimation for auto and manual migration process based on case study
in chapter 8. ................................................................. 166
Figure 8.2: Time estimation for changing migration rules in hard-coded and
configurable migration engine. .................................................. 167
Figure A.1: IEC 61499 system configuration [6]. .................................................. 174
Figure A.2: IEC 61499 event and data flow [6]. .................................................. 175
Figure A.3: IEC 61499 function block interface a) basic FB b) composite FB [6]. .......... 175
Figure A.4: IEC 61499 basic function block execution control chart [6]. ................. 175
List of Tables

TABLE 1: LITERATURE REVIEW OVERVIEW .................................................................................. 14
TABLE 2: MAPPING FUNCTIONS BETWEEN ECC AND FSM. .................................................. 37
TABLE 3: MAXIMUM REACTION TIME COMPARISON BETWEEN PLCs AND FBs......................... 163
<table>
<thead>
<tr>
<th>Term</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFB</td>
<td>Basic Function Block</td>
</tr>
<tr>
<td>BHS</td>
<td>Baggage Handling system</td>
</tr>
<tr>
<td>CFB</td>
<td>Composite Function Block</td>
</tr>
<tr>
<td>DL</td>
<td>Description Logic</td>
</tr>
<tr>
<td>DTD</td>
<td>Document Type Definition</td>
</tr>
<tr>
<td>EC</td>
<td>Execution Control</td>
</tr>
<tr>
<td>ECC</td>
<td>Execution Control Chart</td>
</tr>
<tr>
<td>FB</td>
<td>Function Block</td>
</tr>
<tr>
<td>FBD</td>
<td>Function Block Diagram</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>HMI</td>
<td>Human Machine Interface</td>
</tr>
<tr>
<td>IEC</td>
<td>International Electrotechnical Commission</td>
</tr>
<tr>
<td>IL</td>
<td>Instruction List</td>
</tr>
<tr>
<td>InOut</td>
<td>Input and Output</td>
</tr>
<tr>
<td>LD</td>
<td>Ladder Diagram</td>
</tr>
<tr>
<td>OO</td>
<td>Object Oriented</td>
</tr>
<tr>
<td>OWL</td>
<td>Ontology Web Language</td>
</tr>
<tr>
<td>PLC</td>
<td>Programmable Logic Controller</td>
</tr>
<tr>
<td>POU</td>
<td>Programming Organization Unit</td>
</tr>
<tr>
<td>SFC</td>
<td>Sequential Function Chart</td>
</tr>
<tr>
<td>SIFB</td>
<td>Service Interface Function Block</td>
</tr>
<tr>
<td>SOA</td>
<td>Service-Oriented Architecture</td>
</tr>
<tr>
<td>SQWRL</td>
<td>Semantic Query-Enhanced Web Rule Language</td>
</tr>
<tr>
<td>ST</td>
<td>Structure Text</td>
</tr>
<tr>
<td>SWRL</td>
<td>Semantic Web Rule Language</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XSD</td>
<td>XML Schema</td>
</tr>
</tbody>
</table>
1. Introduction

Majority of industrial automation systems are designed using programmable logic controllers (PLC). In the first part of this section, a brief introduction to PLCs, its' international standard IEC 61131-3 and challenges of applying PLCs for large distributed systems are stated. Next, the background of IEC 61499 standard and why it is suitable for large distributed systems are provided. In the Section 1.3, motivations and objectives of this research work are given. An overview of the proposed methodology is illustrated in the section 1.4. Finally, contributions of this research work are listed followed by an outline of the thesis structure.
1.1 Introduction to PLC, IEC 61131-3 Standard and Challenges

PLCs are dominating the automation industry for decades. PLCs are dedicated for the industry environment with its wider working temperature range, immunity to electrical noise and robust against vibration and impact. Various types of PLCs are deployed in material handling systems, processing control, assembly lines and many other areas.

The PLC execution semantics, programming organization units and programming languages are defined in the IEC 61131-3 standard [1]. The PLC program execution is based on a scan cycle. In a scan cycle, PLCs read all inputs at the beginning of execution scan cycle, then go through every active task and update all outputs at the end of each scan. There are four standard programming languages defined in the IEC 61131-3 standard. Those programming languages can be divided into two types: graphical languages and textual languages. The graphical languages part consists of ladder diagram (LD) and function block diagram (FBD). The textual languages include structure text (ST) and instruction list (IL). Each language has its own advantage, for instance, ladder diagram is similar to electric circuitry drawings and is easy for the maintenance electricians to understand the logics; The syntax of structure text is similar to high-level programming languages like Pascal which is easily adopted by developers with high-level programming experiences. Also the IEC 61131-3 standard provides a graphical state machine specification format called sequential function chart (SFC) which is frequently used for programming control of devices and machines.

The performance of PLC program execution is usually estimated by measuring the scan time. For many real-time applications, the PLC scan time determines whether the control system target is achieved or not. Recently, rapid reconfiguration of existing systems, easy integration with existing systems and downtime-less redundancy become more essential design requirements for PLC based systems. In order to achieve those features, a combination of heterogeneous automation hardware and software is necessary. However, the increasing complexity of such heterogeneous systems poses new challenges to achieving reconfigurability, interoperability and portability.
Introduction

A large system typically comprises a number of devices. For example, a large size airport baggage handling system may have hundreds of conveyors in conjunction with many other devices. For performance and design complexity reasons, it is not possible to control such a highly distributed system by using only one controller. Instead, the entire system has to be divided into a number of subsystems that each subsystem consists of a group of devices. Multiple PLCs are connected to each other via fieldbuses or dedicated controller link interfaces.

However, this distributed PLC architecture has several critical limitations. One of those issues is due to the write access ownership of outputs between multiple PLCs. In IEC 61131-3 PLCs, each analogue or digital output module must be owned by one PLC only. The purpose of this design was to ensure synchrony of the data and protect PLCs from non-deterministic behaviour. But the downside of this is that at the design time a produce and consume pair of PLC variables needs to be pre-determined manually in the source and target PLC, which decreases the flexibility at run-time. Since no more than one PLC can write to a single output module, extra communication interface between various PLCs is necessary to be added for pass the information including program data and I/O module data. This increases hardware costs and the required extensively engineering effort.

In addition, the system performance and reliability will be reduced significantly by copying data between PLCs over fieldbuses or dedicated controller link interfaces. For time critical systems like airport BHS, the reaction time of a PLC scan cycle must be under a certain threshold to ensure all input data changes are detected by PLCs during the system operation. Adding a new PLC to the network increases the reaction time of the entire system heavily. Since the extra PLC has to communicate with all existing PLCs in the network, the portion of time for updating data over the field bus therefore increases significantly. Also, the entire reliance of inter-PLC communications on the field bus or the controller link interface reduces system’s reliability: if any of the connection is lost, the entire system cannot continue operating until the connection is fixed. Therefore, a single connection point failure in the system can lead to long downtime. This will result in big economic and confidence losses for end-users. Even some Ethernet-based fieldbuses supports dual-ring redundancy now [2], systems still could be down if two points on the network are failure.
Another important issue is the compatibility between different PLC vendors. Although all vendors claim their PLCs are compliant with the IEC 61131-3 standard, none of them is compatible with other PLCs in reality. It is feasible to communicate between PLCs from multiple vendors, but this requires both ends support the same type of the fieldbus or the controller link interface with identical protocol. So it is almost impossible to validate the entire distributed systems with PLCs from different vendors.

Finally, the existing PLC tools are not capable for moving a POU instance from one PLC to another automatically. Although POU like functions or function blocks provided by the IEC 61131-3 standard could be reused, lots of human efforts are still required. Also the entire produce/consume variables required by that POU from the original and target PLC must be swapped manually. This leads to reschedule the fieldbus or the network of a live system which is extremely dangerous.

This discussion shows the limitations of using the classic PLC-based technology in the highly distributed automation system implementations. The distributed nodes based on PLCs would not be able to operate independently unless all related nodes are healthy as some critical information for this PLC to make decisions relies on the data from other PLCs. A single PLC connection failure in the system will end up with the entire system out of order.

As stated previously, current PLCs are naturally designed for centralized control systems, with increasing demands of distributed control in the industry, existing generation of PLCs are limited by the IEC 61131-3 standard. Although several new features are introduced into the next revision of the IEC 61131-3 such as object-oriented programming and methods [3], the most important factors including flexibility, reconfigurability and portability of distributed control systems are still missing.
1.2 Introduction to Components-based Architecture and IEC 61499 Standard

Software engineers are facing more challengeable scenarios to integrate products from multiple vendors into the same system with increasing complexity of automation system. In addition, there are always some dissimilarities between projects which requires a lot of re-work time. Modular design concept is often utilized in automation system to reduce the design complexity [4]. Interoperation between multiple PLCs in a single system is significantly important for design and development engineers. Component-based software architecture provides a great improvement on the reusability and flexibility to the control system design. The software components are suitable for integration in large scale distributed systems [5]. Major benefits for using the component-based design in distributed systems are substantial. First, the reduced dependency and highly deployability allows components to be distributed across the system easily. Secondly, the commonality and easy integration of reusable components increase the reusability of software codes. By using the component-based architecture, engineering time and cost can be reduced significantly. Finally, the component based design gives an opportunity to attach semantic categories to each component during design processes. The next generation tools for PLC development would be able to support programming in terms of semantically rich operations, for example applying service-oriented architecture (SOA) into PLCs, rather than just function blocks.

The IEC 61499 standard [6] published in 2005 is based on the concept of component-based architecture. The IEC 61499 standard is purposed to address specific requirements of distributed systems and to complement the IEC 61131-3 standard. The IEC 61499 standard targets for improving flexibility, reconfigurability and interoperability which are currently missing from the current PLC standard. Main features of the IEC 61499 architecture include event-driven execution paradigm and block diagram approach to application design. Main design artefact in this standard is an event-driven function block. This fits to the requirements of distributed control systems much better. The IEC 61499 standard is also a proper technology for replacing the PLC’s cyclic program execution semantics. From the design perspective, IEC 61499 provides a view from system-level and increases portability of event-driven components. Substituting the PLC cyclic scan execution with event-
driven invocation offers a shorter reaction time. The IEC 61499 technology can be very helpful to promote the distributed control design paradigm into the PLC systems [7]. IEC 61499 function blocks can be used to encapsulate control algorithms, deploy them on decentralized networking controllers, and guarantee their performance. Intelligent distributed nodes which are self-organized, self-diagnosed and interoperable with other nodes in the distributed systems could be achieved by using new features of the IEC 61499 standard.

Better flexibility is provided by new features introduced by the IEC 61499 standard which is essential for distributed control systems implementation. One such feature is an open device management protocol defined in the IEC 61499 standard, which enables start, stop, add and delete function blocks during the operation by sending the management commands from software tools or another peer node. When an IEC 61499 controller has detected another controller is overloaded, some tasks configured in this node can be automatically transferred to another node that has capacity for accepting new tasks. Also the system redundancy is improved by enabling the runtime reconfigurability and the task re-scheduling in the IEC 61499 standard. When a failure occurs in an IEC 61499 distributed node, other nodes in the same system would take over the tasks originally assigned to that node and reconfigure with the existing tasks. If a new controller is inserted into this distributed system, existing nodes will be able to detect the new hardware and offload part of tasks with data to the new processor. This new paradigm seeks how to do the necessary reconfiguration when requirements are changed faster and easier, even without stopping the operation [8]. Overall, the challenges of applying PLCs in the distributed systems are overcome by applying the IEC 61499 standard.
1.3 Motivation of Migration from PLC to Component-Based Architecture Control

The migration of software from the currently dominating PLC software architectures to the newly emerging component-based architecture of IEC 61499 is a very important issue for the success of the latter. To meet the requirements from automation and manufacturing industries, migration from IEC 61131-3 to IEC 61499 is considered as an optimized solution to provide better interoperability and redundancy to the existing the PLC systems.

(1) First of all, the success of this new technology depends heavily on the completeness and correctness of migrate the legacy PLC software into the IEC 61499 framework. The impact of the IEC 61499 standard is still minimal due to less support from both academic community and industry. The main reason is the cost to introducing the new technology is very high. The cost involves a large amount of research and development on new hardware and software, learning curve for both PLC programmers and maintenance workers and also promoting this new technology to the PLC industry. To improve on this situation, an automatic transformation from IEC 61131-3 PLCs to IEC 61499 function blocks is necessary as an intermediate step of introducing the distributed control into the PLC industry.

(2) Secondly, the IEC 61499 standard introduces the distributed control into the centralized PLC control world. After a PLC program is migrated to the function block version, the result system configuration could be distributed to multiple function block controllers without any major modification. The performance lag caused by the data ownership issue is solved by the migration process.

(3) Next, all features of the IEC 61499 standard including flexibility, reusability and reconfigurability are enabled in the PLC by applying the migration process [7]. All benefits stated in the last section are also brought into the existing PLC controlled systems. Performance issue on communication overhead and reusable component are also overcome by migrating from the IEC 61131-3 PLC to the IEC 61499 function blocks.

(4) Also the IEC 61499 provides a better system design overview in a graphical representation. The Function block network of the IEC 61499 gives a high-
level abstract of the software design. The function block network also provides the ability of applying computer software design paradigm such as object-oriented programming. IEC 61499 function block network could reflect physical system layout on software design tools. A function block instance in the network could represent a physical device exists in the system.

(5) Finally, this work is motivated by no existing automatic migration approach available. There is no clear guideline on how to redesign various brands of PLC systems into function block systems. A generic set of migration rules as well as an approach without any human efforts are essential for the migration process.
1.4 Migration Process Overview

The transformation process proposed here is not trying to replace the IEC 61131-3 standard with the IEC 61499 standard. The IEC 61499 standard is more suitable as the top level design entity and the actual algorithms inside each IEC 61499 standard function blocks are still using the IEC 61131-3 programming languages. This complies with the vision of many industry practitioners (e.g. [9]) who see IEC 61499 as a complement rather than a substitute to IEC 61131-3. Also in order to improve the accuracy and efficiency of the migration process, original PLC programs should be maximum utilized.

As stated previously, none of PLCs from various vendors is compatible with each other. On the other hand, IEC 61499 IDEs are all following the standard, there are still some implementation variations which lead to incompatibility between them. As illustrated in Error! Reference source not found., a separate implementation is required for every PLC tool to every function block tool. Although the generic set of migration rules could be defined, implementations still vary between various tools due to different code representations of the IEC 61131-3 standard and the IEC 61499.

![Diagram showing migration between IEC 61131-3 Tools and IEC 61499 Tools.](image)

Figure 1.1 : Migration between IEC 61131-3 Tools and IEC 61499 Tools.
Instead, a novel approach is proposed as shown in Figure 1.2. The objective of the migration process covers from importing PLC codes to generating final function block system configurations. The first step is to convert IEC 61131-3 PLC source codes into a platform independent knowledge base. Here, the hypothesis is made that original source codes must be able to be extracted into an XML format file. Then this knowledge base is mapped to IEC 61499 knowledge base using mapping rules defined in the S(Q)WRL ontology query language. Finally, the IEC 61499 system can be extracted automatically from the knowledge base. Also target IEC 61499 IDEs must support all instructions and libraries used in original PLC codes.

![Diagram](image)

**Figure 1.2:** Migration between IEC 61131-3 tools and IEC 61499 tools using knowledge base.

The main advantage of this approach is that the knowledge base is a self-documented high-level description language. Extra transformation rules can be added easily without any programming skills required. This approach provides a human maintainable solution to end users. The cost of the migration process on the man hours is reduced to a minimum level with provided knowledge base definitions. This approach is also applicable for migration between IEC 61499-compliant tools. After migration process, result IEC 61499 system configurations are required to be checked syntactically and semantically to ensure the correctness. In order to performing the semantic analysis of the IEC 61499 standard automatically, a generic set of semantic rules for IEC 61499 needs to be defined.
1.5 Contribution of the Research

The target of this research is to perform an automatic migration process from IEC 61131-3 PLCs to IEC 61499 function blocks to address the challenges described in the first section of using IEC 61131-3 PLCs in distributed control systems. The main contributions of this thesis are:

1. A novel migration method. This migration method includes a summary of patterns for redesigning PLC programs in the form of IEC 61499 function blocks. Based on the analysis of all patterns, a pattern that best suits the automatic migration can be selected. An IEC 61131-3 formal model and an IEC 61499 formal model are defined. In the IEC 61131-3 formal model, the execution semantics of the IEC 61131-3 is also included. A scheduling system in function blocks form is designed to achieve the identical PLC execution semantics in the IEC 61499. Finally, the generic mapping rules between two formal models are provided.

2. A configurable transformation engine using ontological knowledge base. The web ontology language is selected as the representation of knowledge base used in the migration process. Both IEC 61131-3 and IEC 61499 ontological knowledge base definitions are created automatically from their XML definitions. A transformation engine is implemented for perform migration by processing mapping rules. The mapping rules are human maintainable which can be configured without any modification to the transformation engine. Also the migration time is reduced significantly by using this transformation engine.

3. Ensured semantic correctness. An automatic semantic analysis engine is built to ensure the semantic correctness of migrated IEC 61499 system configurations. The semantic rules for the IEC 61499 standard are defined in the form of the SQWRL language as well. The semantic analysis engine is not required to be changed if new semantic rules to be added or changes to existing semantic rules needs to be performed.

4. Interoperability between platform dependent tools. The transformation engine is also capable for migrating between IEC 61499 platform dependent tools with a different set of mapping rules.
1.6 Thesis Structure

The thesis consists of nine chapters as shown in Figure 1.3.

Figure 1.3: Thesis Structure Diagram.

In Chapter 2, all related works are reviewed and discussed. Current PLC design methodologies and design recovery from PLC programs are analysed. Publications regarding to all IEC 61499 execution semantics, reconfiguration using IEC 61499 and automatic code generation are reviewed. Next, existing migration approaches are discussed in details. Ontology mapping techniques are revised at last.

In Chapter 3, several design patterns such as object-oriented design patterns are provided. The data handling efficiency of all patterns is compared. Performance estimation of the redesign patterns is defined. The design pattern which suits best for the automatic migration is then selected for the automatic migration process from PLCs to function blocks.

The formal model of the IEC 61131-3 standard is defined in Chapter 4. In order to achieve cyclical execution behaviour of the original PLC applications in the generated IEC 61499 systems, the formal IEC 61131-3 execution model is developed. The
equivalent execution model in IEC 61499 is built according to the PLC execution model. In addition, the formal model of the IEC 61499 standard is also defined. Finally, the mapping rules between two models are defined in this Chapter.

Chapter 5 introduces the migration foundation - ontological knowledge base definition of IEC 61131-3 and IEC 61499. For the IEC 61131-3 knowledge base, definitions are generated from the XML definition (DTD or XSD). Two IEC 61131-3 knowledge base examples (Rockwell from DTD and PLCOpen from XSD) are given. The details of the IEC 61499 knowledge base are also provided.

Following that, Chapter 6 describes the definition of the mapping rule template, followed by illustration of the transformation engine implementation. Then the mapping rules, defined in the Chapter 4, are implemented in the template using the SQWRL language. Last but not the least, the migration process between IEC 61499-compliant tools is provided.

In Chapter 7, rules of semantic correctness for basic function block, composite function block, service interface function block and system configuration are formally defined. Then those semantic rules are implemented using the SQWRL language. Finally, the automatic semantic analysis engine is described.

A case study of an inbound airport baggage handling system is given in Chapter 8. All redesign approaches are applied to this example programmed in the PLC. The automatic migration from PLC to FB, migration from NxtStudio to FBDK and automatic semantic analysis are applied to this example. The performance is measured on both PLC and function block implementation.

In the last Chapter 9, the proposed extension to the S(Q)WRL language is provided. The efficiency and impacts of the proposed migration method are given. The thesis is concluded and limitations with recommendations of future works are listed.
2 Literature Review

The aim for the literature review is to find out existing migration approaches with their advantages and disadvantages. The need for migration from IEC 61131-3 to IEC 61499 in distributed PLC control systems has been recently recognized by several researchers [10]. In this chapter, the existing research works related to the IEC 61131-3 PLC, the IEC 61499 function blocks and the migration are reviewed. The review order and structure is given as:

Table 1 : Literature Review Overview

<table>
<thead>
<tr>
<th>Section/Literature Review Area</th>
<th>Review Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 IEC 61131-3 PLC Design Patterns</td>
<td>Determine existing PLC design patterns and find out which could be used for automatic migration</td>
</tr>
<tr>
<td>2.2 IEC 61131-3 PLC Design Recovery</td>
<td>Investigate existing approaches for first step of migration - PLC design recovery from sources code</td>
</tr>
<tr>
<td>2.3 Component-Based Architecture in IEC 61499 Function Blocks</td>
<td>Review existing works on using IEC 61499 function blocks as component-based architecture</td>
</tr>
<tr>
<td>2.4 IEC 61499 Reconfiguration and Code Generation</td>
<td>Research on results on reconfigure industrial system using IEC 61499 and automatic code generation for function blocks which is useful for last step of migration</td>
</tr>
<tr>
<td>2.5 Existing Migration Approaches</td>
<td>Study all existing migration approaches and investigate possibility of reuse some existing research results</td>
</tr>
<tr>
<td>2.6 IEC 61499 Execution Semantics</td>
<td>Investigate all IEC 61499 execution semantics to find out a generic solution to ensure execution semantics are not changed after migration</td>
</tr>
<tr>
<td>2.7 Ontological Knowledge Base and Ontology Mapping</td>
<td>Find out research works on using ontological knowledge base in the industry and methodologies of ontology mapping which could be used in the migration process</td>
</tr>
</tbody>
</table>

In the first part of this chapter, the work involving the IEC 61131-3 PLC generic design methods and program patterns are reviewed. Those PLC code styles are quite useful for summarize all PLC design patterns for migration. Other than the design patterns, design recovery from the PLC code is also an important part in the migration
process. This section will provide feasible methods of recognizing PLC code patterns from the “raw” PLC programs.

Following sections related to PLCs, researches based on using IEC 61499 function blocks to achieve component-based design are analyzed. Also existing works on function block design patterns are reviewed. Those patterns could be used as the target design pattern in the migration process. Then, the papers related to the reconfigurability and the code generation of the IEC 61499 standard are discussed. Next, existing migration approaches and case studies are reviewed. Works related to the known IEC 61499 execution semantics are presented in order to propose a solution to recover the cyclical execution semantic in the function block execution. Some works regarding the ontology in the automation industry and ontology mapping are provided as references of the knowledge base definition of the IEC 61131-3 and the IEC 61499 standard. Finally, this section is concluded with a summary of all reviews.
2.1 IEC 61131-3 PLC Design Patterns

Estevez et al. [11] suggest an automatic generation of PLC automation project using component-based models. Those component-based models are defined using a proposed language industrial control systems (icsML) in the form of the XML. This markup language is further transformed to the PLCOpen XML format by using the XSLT translation table. This approach provided a guideline on how to generate a PLC program from a component-based model. The concept is similar to use ontology to generate function block programs.

Recently the object-oriented programming concept from high level languages is introduced into PLC world. Benitez Pina et al. [12] discuss the possibility of introducing objects into all PLC programming languages. Private, protected and public data members and functions encapsulated in objects are proposed. Also polymorphism was considered. This is included in the next version of the IEC 61131-3 standard and already implemented by CoDeSys [13].

Ferrarini et al. [14] present an object-oriented modeling techniques to design and development of PLC programs. Their object-oriented model was represented in the UML and a Petri net model was built for a formal analysis. However, the actual implementation was built on a soft PLC based on the RTAI Linux operation system instead of using the real PLCs. Also, in their later research work [15], the object-oriented modeling was implemented in the IEC 61499 function blocks. Finally, this function block system configuration was converted to the AWL executable code automatically.

Bonfe et al. [16] describe a development of using object-oriented model in the IEC 61131-3 standard. Object-oriented concept was introduced into IEC 61131-3 function blocks. By adding features of object-oriented programming into the function block, the PLC would have the option to do object-oriented programming.

The limitation of applying object-oriented programming in the PLC is discussed by Ma et al. [17]. Instead an abstract object was proposed. The abstract object is based on task-oriented programming. This idea is the same as service-oriented architecture in the programming.

A comparison between sequential function chart and object-modeling PLC programs is provided by Hajarnavis et al. [18]. This paper illustrate an example of an
assembly line controlled by Rockwell PLCs. Two code styles using SFC and object-oriented concept in the PLC are discussed. The benefits of using both approaches are listed. The conclusion from this paper is that each code style suits a certain type of the application. There is no clear boundary between two approaches presented in this paper.

Bonfatti et al. in [19] illustrate the process for the development of PLC software. The design paradigm of object-oriented with model based approach was introduced in the PLC code development to achieve a higher degree of modularity and reusability. The importance of the design consistency criteria was discussed to ensure a smooth final integration of the software components.

An object oriented programming paradigm approach on the IEC 61131-3 standard is proposed in the paper [20] by González et al. This proposed approach includes adding methods into an IEC 61131-3 function block, inheritance and polymorphism of the function block, access control of private, protected, public and friend. This approach offered a clear separation between the standard POUs and the OO extension proposed by the authors.

From existing approaches, the most popular design pattern for PLCs is the object-oriented approach.
2.2 IEC 61131-3 PLC Design Recovery

The main goal of this section is to find out existing methodologies of capture PLC design pattern from PLC codes – Design Recovery. Here, different design recovery methodologies are reviewed.

Several design recovery methodologies are illustrated by Falcione et al. [21]. Authors focus on recovery the ladder logic (LD) and SFC program from programmable logic controllers (PLC). The design recovery algorithms for LD and SFC are provided. The results are represented in a graphical way which is further to be expressed in math equations. But this approach is not suitable for code generation as the result of this approach is not using an open standard format which could be easily interpreted.

Another design recovery approach is provided by Lim et al.[22]. For the re-engineering process of design recovery, authors suggest a formal specification language concept mapping language (CML) as a storage media. Terminology, expressions and grammar definition for the CML are provided. The expression definition of CML is similar to description logic used by ontology but transformations are still done manually. To achieve automatic transformation, this approach is also not applicable.

There is a body of several approaches which are not particularly designed for the automation industry. A method [23] is described to assist design recovery and understanding programs by recognizing design pattern instances. This method is designed for high-level programming languages and the pattern defined is for object-oriented programming. This method is only semi-automatic and some manual efforts must be put in during the design recovery process. A similar approach is proposed by Dean et al. [24] for solving the year 2000 problem of the computer program. Biggerstaff [25] proposes a design recovery model for maintenance and reuse the code programmed in C language. The original C program was divided into groupings of data abstractions and modules. The recovery process was performed on top of the abstractions. Malton et al. [26] describe a simple technique of design recovery from some text. The text is divided into two collections: independent texts and interrelated texts. The design recovery is achievable by separating the source code into lexically standard sub-streams and further indicated by the markup.
Jahnke et al. [27] review a number of tools developed for recovering the design of legacy software. The limitation of those tools was pointed out. They are all designed for linear transformation process and less support for iteration, recursion and incremental changes. The solution for solving those issues is proposed. Another review of the reverse engineering and design recovery techniques for the object-oriented programming is provided by Sadiq et al. [28]. The limitations and advantages are also indicated.

All approaches for design recovery from PLC code are not straight forward. Extra efforts must be put in to achieve an acceptable accuracy level for PLC design recovery.
2.3 Component-Based Architecture Using IEC 61499 Function Blocks

There are several researchers focus on applying the component-based paradigm in the PLC software development. Thramboulidis et al. [29] propose a control and automation design framework based on the idea of service-oriented architecture. Each feature required during the manufacturing process is defined as a service. The entire framework consists of multiple services which supports the development process for engineers.

Stromman et al. [30] provide guidelines of using IEC 61499 in the process automation domain. The original PLC code written in the structure text is proposed to be reused inside the function blocks. Also a converted version of the structure text in the ECC was also provided. Those ideas are in line with the ideas proposed in this research work.

Zoitl et al. [31] propose a hierarchical component-based approach using IEC 61499 language. A new domain-specific language MONACO was introduced by the authors to model the hierarchical automation solutions. The components then decoupled into the hierarchical function block layers. The concept of using the IEC 61499 sub-application to simplify the design is suggested.

An approach of implementing intelligent mechatronic component (IMC) model using the IEC 61499 standard is presented by Pang et al. [32]. The concept of applying model-view-control design pattern in the IMC model and the IEC 61499 is proposed. Also some semantic extensions to the IMC model are provided to helping on automatic reconfiguration and generation of the function block systems.

Black et al. [33] propose a multi-agent based airport baggage handling control design using the IEC 61499 standard. The distributed control version of the converted baggage handling system is proved to be scalable, reconfigurable and fault tolerant. The benefits and effectiveness using distributed control are demonstrated.

Yan et al. [34] also prove fully distributed baggage handling systems using IEC 61499 function blocks. Each device is encapsulated into a single re-usable function block in this design approach. The flexibility, reusability and fault tolerance are also demonstrated.
Sunder et al. [35] present an design architecture based on the reference model the IEC 61499 standard. The proposed architecture took extra consideration the distributed systems. By applying the IEC 61499 SIFB communication blocks, the system programmed in the IEC 61499 function blocks were easily distributed over the network.

Component-based architecture is perfect in line with the concept of reusing functions or function blocks from PLC in IEC 61499 function blocks.
2.4 IEC 61499 Reconfiguration and Code Generation

A lot of publications have already proved the reconfigurability could be achieved by using the IEC 61499 standard. Hagge et al. [36] propose a component based modeling language derived from the Petri nets. This graphical language CNet avoids the possibility of error-prone designs in modeling languages due to data inconsistency. The interface of CNet model is capable to be generated automatically from the IEC 61499 function blocks. However, the method is using a graphical language which is not suitable for automatic conversion.

Orozco et al. [37] apply IEC 61499 for distributed multi-agent control systems. A process control hardware platform is selected as a testbed. The algorithms of the process control are encapsulated in function blocks according to IEC 61499. The coordination control between multiple agents is done via exchange of messages modelled by events and data. The paper also shows that reconfigurability of the test bed on the fly is feasible by using the IEC 61499 standard management commands.

Brennan et al. [38] propose an approach for dynamic reconfigurable real-time distributed control systems using the IEC 61499 function blocks. This approach is based on agent-based control addressing the issue of real-time reconfigurations in distributed control systems. Also fault detections and automatic recovery can be achieved by applying this approach.

Hegny et al. [39] provide a three-layer automation architecture with IEC 61499 on the low level control part and agents on the high level control end. A knowledge base of the system is used as the top level design where agents gain the system information. This structure enables reconfiguration during the operation and simplifies code deployment to distributed control systems due to modularity of the design.

Khalgui et al. [40] [41] propose a reconfiguration protocol for multi-agent controlled system using the IEC 61499 function blocks. A reconfiguration agent and coordination agent were inserted into each IEC 61499 device as an enabler of the reconfigurability. Both agents are verified by the net condition-event systems (NCES) which is an extension of petri nets.
Also there are some successful experiences on how to generate IEC 61499 code automatically.

Goh and Dint in [42] describe an approach to code generation for IEC 61499 based on the iterative knowledge base. The iterative knowledge base is represented in the form of XML and Extended Backus-Naur Form (EBNF). The goal of that approach is to eliminate any additional script language to be used in the code generation. Also the translation rules are configurable and reusable to improve the accuracy of translation rules. In order to achieve this goal, rule-based blocks are built for each IEC 61499 XML element. During the code generation process when the pre-defined rules are satisfied, the related block of code will be generated and data types and connections will be also inserted. However, this XML and EBNF based approach is not convenient for configuring the IEC 61499 systems manually. Besides, the code template must be pre-defined in the knowledge base manually and cannot be easily modified when the code template is changed.

An approach to generate FB based designs automatically from requirements using Semantic Web technologies is proposed by Ryssel et al. [43]. A generic approach is developed based on some specific design patterns. Different function block types (not IEC 61499) are used but no FB ontology is presented.

The paper by Lepuschitz et al. [44] is more focused on reconfigurability of the control system during the real-time execution for multi-agent systems (MAS). To achieve reconfiguration of a manufacturing system on the fly, the IEC 61499 standard best suits the requirements. The approach provided for MAS is the combination of ontological representation for low level control and a UML type format for high level control. The reasoning can be applied to low level control part which provides flexibilities to manufacturing system as well as predicting ability to reconfigure the entire system during the processing. This concept is further combined with ontology-based knowledge base to provide intelligence to an agent using the ontology [45]. Similar approach is proposed [46] to use agent with ontology to detect failure and perform automatic recovery.

Most of existing approaches for reconfiguration are based on implementing agents using the IEC 61499 standard. General concept of code generation for IEC 61499 format is to store codes in a unique format which can be interpreted easily.
2.5 Existing Migration Approaches

There are several researchers published their works related to the migration between IEC 61131-3 PLCs and IEC 61499 function blocks.

An initial migration case study on a FESTO MPS didactic plant is presented by Hussain and Frey [47]. This paper however relies on the assumption of PLC code being preliminary re-engineered into Petri nets, and considers only one state-machine conversion to ECC of function block. To deploy distributed control in this plant model, each station is controlled by a different processor. The paper indicates that the event-driven execution time of IEC 61499 is shorter than the scan cycle execution in PLC. It also proves feasibility of the conversion from state machines to ECC. However the whole picture of state machine to ECC conversion was not presented in that work.

Peltola et al. [48] has provided an example on manual migration from IEC 61131-3 PLCs to IEC function blocks. PLC control code for a batch process was converted to the IEC 61499 function blocks with integrated HMI views. The paper has only proven that using IEC 61499 function blocks replacing for IEC 61131-3 PLCs is feasible.

The papers by Hirsch et al. [49] and by Gerber et al. [50] propose the integration solution for IEC 61499 standard into the IEC 61131-3 framework based on both centralized and distributed approaches. A distributed heterogeneous system testbed of Netmaster controllers running IEC 61499 and Siemens PLCs is demonstrated. A communication interface between PLCs and IEC 61499 controllers is designed. Also the Model-View-Control (MVC) concept is introduced for including the HMI part of the testbed into the IEC 61499 solution. However the redesign process is done manually without summarizing any systematic migration rules.

Wenger et al. [51] propose a model driven migration approach to convert existing IEC 61131-3 into IEC 61499 standard. The original IEC 61131-3 PLC code, stored in the standard PLCOpen XML format, is transferred to the IEC 61499 XML format. However, as admitted by the authors, this approach cannot guarantee that the generated IEC 61499 XML system can be executed directly in the IEC 61499 function block runtime without further manual modifications. A model-driven based transformation and semantic correction from IEC 61131-3 to IEC 61499 is also proposed by Wenger et al [52]. Authors proved that with IEC 61131-3 and IEC 61499 libraries available, it is possible to transform source IEC 61131-3 model into an
internal E-core model and then write it in the target IEC 61499 XML model. Recreating IEC 61131-3 execution order in IEC 61499 is feasible but how to connect events in order automatically is still not completely solved. However, this approach focuses on the translation of the PLC code sentence by sentence. No entire PLC program structure is considered.

Shaw et al. [53] focus on migration from the Rockwell PLC ladder logic by translating it to C code with synchronous semantics. A semi-automated translation process from IEC 61131-3 ladder logic to IEC 61499 function blocks is established. This approach admittedly enables migration of IEC 61131-3 PLC code to IEC 61499 architecture without loss of previous design efforts. But this implementation is limited to a particular PLC platform and cannot be used as a generic guide for the migration process.

Sunder et al. [54] propose a solution for converting IEC 61131-3 automation projects into IEC 61499 standard. Differences between two standards are carefully compared. The mapping between two software models is also provided. The variables with various level of access are considered. However, mapping of global variables which are commonly used by IEC 61131-3 to an IEC 61499 format is not solved and automatic transformation is not implemented using this approach.

Basile et al. [55] show how object oriented programming could implicitly make the event-based PLC software behaviour. Authors propose a solution which is the combination of two existing approaches in introducing object oriented programming into PLCs: adding object-oriented support in the IEC 61131-3 standard or adopting the IEC 61499 standard. The IEC 61131-3 programming language SFC is used to develop a proper object-oriented approach. This is just a concept and no complete rule is defined.

Overall, there are several existing migration approaches. But all of them lack part of either correctness or completeness. None of them could be used for automatic migration between multiple PLC platforms and FB IDEs.
2.6 IEC 61499 Execution Semantics

There are various execution semantics as the execution semantics is not clearly defined in the IEC 61499 standard.

Hagge et al. [56] propose a handler-based execution model to the IEC 61499 standard especially for basic and composite function blocks. This execution model HB-XM provides a complete dynamic event scheduling system. The deadlines are set to the event handlers to meet the requirements of the real-time constraints.

Catalan et al. [57] propose an execution model on the application level to ensure the execution behavior of the function block design system at the design stage. This model defines the status of the execution including initialization, normal and failure.

Wang et al. [58] propose an integrated approach for collaborative manufacturing using IEC 61499 function blocks. This approach supports dynamic scheduling on runtime. Each manufacturing process is embedded into an IEC 61499 function block. This allows machine-level adaptive decision-making and provides intelligence by using function blocks.

Strasser et al. [59] discuss several design and execution issues related to the aforementioned problems for IEC 61499 distributed control systems. This paper focuses on the design from the execution models perspectives rather than the application view.

A cyclic execution semantic for the IEC 61499 runtime is developed by Tata et al. [60]. The runtime is programmed using the Java language and execute the function block in a sequential way which only activate one function block at one time. On the other hand, Li [61] develops a FB to C compiler using the synchronous execution semantics. The author claims the performance advantage of using synchronous execution semantics compared to other execution semantics.

Cengic et al. publish a series of work on the IEC 61499 formal models and execution semantics. Some preliminary researches on the formal model of the IEC 61499 standard is presented [62] [63] followed by a complete formal model of the IEC 61499 standard [64]. Also the formal model of the execution semantics are also defined in [65] [66]. Execution models including buffered sequential, non-preempted multithreaded and cyclical buffered are defined.

Dubinin et al. [67] provide a refactoring of the ECC in basic function block of the
IEC 61499 standard. The advantage of the refactoring process is to remove arcs without event inputs and get rid of potential deadlock states. The concept of model-driven engineering is introduced.

A new event-chain concept is introduced by Zoitl et al. [68] [69]. The runtime issue of optimize the event-chain is discussed. Based on this concept, execution models of function block network, device and resource execution model are provided.

There are three common execution semantics implemented for the IEC 61499 standard. All of them are using quite different algorithms and can lead to a FB program behaviors differently.
2.7 Ontological Knowledge Base and Ontology Mapping

An idea of applying ontological mechanisms for semantic analysis is appeared in the work by Arakawa [70], but in a completely different domain: it proposed a method for analyzing natural language texts using ontologies. Ontologies are widely used in the software engineering domain [71], [72]. However, there is no previous work about the using of ontologies for the semantic analysis of programming and domain specific languages. The authors [73] also proposed a reverse engineering tool to understand the program and design recovery using the ontology. The definition of the ontology used in this paper is designed for the high level programming languages, not particularly for PLCs. Valiente et al. [74] also suggest an integration between the ontology domain and the IT service management domain. The software implementation processes and models were defined in the ontology. A set of rules which allows the specification of the underlying constraints and the eventual inference mechanisms were defined using the SWRL.

The feasibility of applying semantic web and service-oriented architecture into automation industry has been discussed by Jammes and Smit [75]. The work is motivated by the challenges of interoperability, scalability, plug-and-play connectivity and seamless integration. The service-oriented architecture using the Web services standards is applied to automation systems. That paper proves that the use of SOA and Web services standards can enhance the intelligence of automation systems.

A similar knowledge base approach for semantic description has been demonstrated [76] for describing Petri Nets. The Petri Nets Markup Language (PNML) is used to describe the vocabulary and the functional relations between the concepts. However, PNML cannot be used to describe the Petri Nets semantics. In this paper, a layered ontology for modeling high-level Petri Nets is proposed. Data types, knowledge representation and high-level Petri nets are described in the hierarchical ontology structure which is suitable for analysis and checking the behavior of Petri Nets.

Orozco and Lastra [77] illustrate the idea of how semantics can be added to the Function Blocks reference models of the standard IEC 61499 by using ontologies. But the intention of this paper is to use semantic descriptions of FBs for automatic searching and discovery of FBs in applications based on the web services. Also the
FB ontology model presented in this paper is not detailed enough for semantic analysis.

An ontology based reconfiguration system is also using IEC 61499 [78]. To achieve rapid reconfiguration during the manufacturing process, the agent is built based on an ontological model. This model contains all knowledge about the manufacturing process. With any change during the physical environment, the agent will react without any human intervention. This automatic reaction is based on ontology query and reasoning. Once the reaction is decided from the agent, the system configuration will be reconfigured immediately to respond the physical change.

This thesis extends the idea proposed by Dubinin et al. [79]. A technique is presented for semantic analysis of IEC 61499 based on ontological model which can be regarded as a direct predecessor of this research. While the concept has been soundly demonstrated in this paper, it was based on a manually created ontological model. It is hard to create software tools implementing it on systematic basis since instance data have to be entered manually into the ontological knowledge base. The new approach [80] of the present paper uses automatically generated ontological models with data instances which pave the way to implementation of software tools. Also this paper presents new execution semantic rules of execution control chart (ECC) in basic function blocks and other complicated rules, not covered in this paper.

An ontology for semantic analysis of IEC 61499 compliant systems is proposed. The function block type ontology model includes basic, composite and service interface function blocks and system configurations. The ontology model for any function block type includes a model of its interface. Along with that, the ontology model of a basic function block includes the Execution Control Chart (ECC) ontology model. In the ECC ontology model, EC state, EC algorithm, EC Transition and EC Transition conditions are defined. The composite function block ontology model includes references to the component function block instances along with models of event and data connections. Finally, the system configuration ontology model contains devices, resources, applications, connections, mappings and network segments and their object and data properties. That paper provides simple examples of semantic analysis for IEC 61499 files using description logic and SWRL. A semantic analysis tool using a Protégé plug-in is developed for automatic semantic checking.
2.8 Summary

In the first part of the literature review, the existing IEC 61131-3 PLC design patterns and design recovery methodologies are analyzed. The PLC contains two basic code styles: state-machine style and long-hand style. Recently object-oriented concept is introduced into the IEC 61131-3 standard but no actual application is using that concept yet.

Following the PLC design, reviews of the existing function block design patterns, reconfigurability and code generation are performed. The object-oriented pattern is also popular in function block design. Also component-based approach is suggested for some automation domain. However, there is no clear guide on how to redesign PLC code styles as well as mapping PLC codes following the object-oriented pattern into function block formats. The reconfigurability is majorly relied on the implementation of agent-based control in the function block network although some suggests achieving that on the runtime level. The migrated system configuration will be able to achieve the reconfigurability by introducing those agents.

Then existing migration approaches and execution semantics are presented. An automatic transformation between PLCOpen XML and IEC 61499 FB is proposed by Wenger et al. [51, 52]. This transformation process is driven by a meta-model defined in the standard XML format. This work is limited to the import from the PLC code programmed in FBD using PLCOpen XML format. The approach does not provide the mapping between other IEC 61131-3 languages to ICE 61499 as well as is not generic which could be applied to other IEC 61131-3 XML formats. The mapped function block systems using this approach cannot automatically connect event connections and associate with proper data variables. This requires human efforts on manually correct those connections. From time and cost perspective, a fully automatic process is preferred. A careful comparison between two standards is completed. Some of the proposed mapping rules are similar to the results proposed by this thesis like mapping from IEC 61131-3 project to IEC 61499 system configuration. However, those mapping rules are not formally defined. Some elements from the IEC 61131-3 could be mapped to multiple elements in the IEC 61499 standard. More importantly, the execution orders of PLC tasks and programs are not considered in the mapping rules. Overall, Shaw et al. [53] propose a conversion from Rockwell PLC code into the function block with C code as the algorithm using the synchronous execution
semantics. However, this approach is based on sentence-by-sentence transformation and rules are all hard-coded. It is dedicated to one PLC platform and one function block execution semantic. There is no systematic approach on automatic migration between an IEC 61131-3 PLC and IEC 61499 function blocks available.

Also none of these existing approaches offers an identical IEC 61131-3 execution model for all IEC 61499 execution semantics including sequential [60], parallel [68, 69] and synchronous [61]. An IEC 61499 system configuration could result in different statuses using various runtime environments. This work aims at addressing these issues. To achieve the PLC execution semantics in the function blocks, a unique solution for all execution semantics must be proposed.

Finally, some ontology-related researches are analyzed for gathering some ideas for mapping the IEC 61131-3 and the IEC 61499 ontological knowledge base. One of the works proposed to use the SWRL rule in the computer programming is a feasible solution in this case. Dubinin et al. [79] proposes to use the ontology web language combined with the SWRL rule to perform semantic analysis for IEC 61499 system configurations. This idea is extended with using SQWRL language and automatic import from the IEC 61499 code.

The proposed ontology model in this thesis is based on the XML schema and proposed semantic rules are defined in the SQWRL query which uses the XML schema as the source. This improves the efficiency and enables automatic semantic analysis which is not supported in the original work. Besides that, the semantic rules defined in the original work are more related to usage of events with data variables rather than execution semantics. For example, there is no ECC related rules defined as well as deadlocks in the event connections. This idea is also applied to the automatic migration process that the mapping rules are defined in SQWRL queries as well.

The main conclusion to be derived from the existing migration approaches is that a proper balance between domain-specific and universal approaches needs to be found in order to achieve an automatic migration process which could be easily configured for various PLC and function block platforms. The mapping rules between IEC 61131-3 elements and IEC 61499 elements provided by Sunder et al. [54] could be utilized.
3 Redesign Patterns

In this section, several design patterns for IEC 61499 are illustrated. The section starts with an introduction on current design techniques of IEC 61131-3 PLC programs. Two re-design methods based on the object-oriented approach are proposed in the next section. In Section 3.3, the class-oriented approach is introduced. Besides redesign approaches, another critical aspect of the redesign process is the data handling efficiency. A mapping between the original PLC design pattern and those redesign patterns is also provided. The data handling efficiency between various redesign approaches are provided in Section 3.4. Followed by that, the performance estimation for those approaches is discussed. Finally, this section is concluded with a comparison and discussion between approaches as well as limitations for each approach. A design pattern is selected for the automatic migration process based on the comparison results.
3.1 Introduction

Before starting migration process, the first step is to recognize the PLC design patterns and provide a unique IEC 61499 design pattern could handle all of the PLC design patterns. In more general approaches [51], the distribution of the resulting code is problematic since it is almost equivalent to the original PLC code. The function block instances in the result IEC 61499 system do not reflect any system level design information such as functionalities or physical layout like objects in the system. It is difficult and time-consuming to recognize what does each function block do in the resulting code. It is impossible to enable scalability for the result function block system if the design pattern could not be recognized. On the other hand, it is hard to apply results of the narrowly focused case studies in other domains. In this Chapter, the PLC patterns are reviewed and three re-design patterns of function blocks are proposed and investigated. Each of the proposed methods is to be applied under different assumptions on the original PLC code structure. Through studies on those re-design methods, a human maintainable and scalable method is used for the automatic migration process.

The PLC-based hardware and software architectures need to be specified before discussing the redesign approaches. There are two typical PLC design architectures: distributed and centralized. In the distributed architecture, each PLC is located in a separate control cabinet in rack with I/O modules and connected to other PLCs by fieldbuses. In the centralized way, all PLCs are placed together in a single rack in the main control cabinet and connect to remote I/O modules via fieldbuses. Those two approaches are shown in Figure 3.1.

Figure 3.1: PLC network design for centralized and distributed system.
In a PLC, function blocks or functions are created as a reusable software component performing a particular functionality or controlling a particular type of devices in the system. During each scan, the PLC reads all inputs from local and remote rack, executes every scheduled program in the predefined order and updates all outputs at the end of each scan. Inside a scheduled PLC program, algorithms are presented either in a sequential order or as a finite state machine written in one of the five IEC 61131-3 programming languages (Graphical: ladder diagram, sequential function chart and function block diagram. Textual: structure text and instruction list). The finite state machine style of the PLC code suits better for representing control logic of devices and machines as each state could reflect a physical status of a device or machine [81]. Functions such as data processing, calculation and communication interface are better implemented in the sequential logic style of the PLC code. PLC codes defined in either FSM or sequential style are able to be encapsulated in functions and function blocks for reuse. New instances of functions and function blocks can be invoked in the program. An additional part of a PLC program for distributed systems is the data communication between PLCs: a PLC must produce periodically the data required by other PLCs over the network, which must be received by those PLCs prior to the beginning of the next scan cycle.

To redesign a distributed IEC 61131-3 system written in either state machine style or long-hand style in IEC 61499, there are two basic design patterns can be distinguished: the object-oriented pattern and the class-oriented pattern. The former approach is observed in many works proposing the design of IEC 61499 applications from scratch, as well as in some migration works, e.g. [47, 50, 51, 53], while the latter has been introduced [82].
3.2 Object-Oriented Design Pattern

The object-oriented approach is the most popular design pattern in the IEC 61499 design process. The basic function block type is used as a fundamental software component that contains all related functionalities, program variables and interfaces of a single physical device (object). Similar to the concept of object-oriented programming in the high-level programming, a separate instance of this function block type would be created for each physical device in the system in the IEC 61499 context. After all instances are created, those FB instances are connected to each other according to the physical layout position order. Some characteristics of the object-oriented programming concepts are supported in IEC 61499 function blocks such as encapsulation of data and methods and class instantiation. However, data abstraction, polymorphism and inheritance are not available in the current edition of the IEC 61499 standard. There are two re-design methods for the object-oriented pattern. One is based on the transformation of the PLC code into ECC of the function block, and the other on the direct re-use of the PLC code into an algorithm within a basic FB.

3.2.1 Redesign Finite State Machine to Basic Function Block ECC

The first option of the object-oriented design method is to present the finite state machine (FSM) type PLC POUs (function, function block or program) as the execution control chart (ECC) in basic function blocks. In order to redesign the FSM, the original PLC code must be designed in any form of a state machine. There are two types of FSM: Moore type whose output actions are associated with the state and Mealy type which the transition is bounded to the output actions [83]. Also a FSM could be deterministic or non-deterministic. The assumption made here is that the original PLC program must have a deterministic state machine.

Firstly, a formal model describes the finite state machine (FSM) is needed. A FSM is defined as:

$$FSM = (Σ, Γ, S, S₀, δ, ω) [83] (3.1)$$

Where $Σ$ is a finite set of input alphabet and $Σ ≠ ∅$; $Γ$ is a finite set of output alphabet and $Γ ≠ ∅$; $S$ is a finite set of states and $S ≠ ∅$; $S₀$ is the initial state and $S₀ ∩ S ≠ ∅$; $δ$ is the state transition function $δ : S × Σ → S$; $ω$ is the output function.
The FSM consists of states and state transitions which jump between states based on the changing values of state conditions rely on the PLC inputs. In terms of the precursor FSM, the execution of state machine based PLC code can be represented as the following sequence of steps (repeated again once the POU is scheduled to be executed by the PLCs):

1) Calculate Boolean expression in the state machine transition conditions;
2) Determine the next state; (if there are concurrent states then the next state is decided based on the priority and the order of those concurrent states)
3) Execute actions associated with the next state.

Alternatively, the basic function block is defined as:

\[ BFB = (Interface, ECC, Alg, IntData), \]  \[ (3.2) \]

Where \( Interface \) is the function block interface (including inputs/outputs event and data); \( ECC \) is representing an execution control chart in the basic function block; \( Alg \) is a set of algorithms associated with EC states; \( IntData \) is a set of internal variables only used in this basic function block.

Next the formal model of the function block interface is designed as:

\[ Interface = (EI, EO, DI, DO, EI\tau, EO\tau) \]  \[ (3.3) \]

Where \( EI \) is a finite set of event inputs; \( EO \) is a finite set of event outputs; \( DI \) is a finite set of data inputs; \( DO \) is a finite set of data outputs; \( EI\tau \) is the event input association function \( EI\tau \subseteq EI \times DI \); \( EO\tau \) is the event output association function \( EO\tau \subseteq EO \times DO \).

And the formal model of the ECC is given as:

\[ ECC = (ECstate, ECTrans, ECTCond, ECAction, ECstate_0) \]  \[ (3.4) \]

Where \( ECstate \) is a set of EC states; \( ECTrans \) is a set of EC transitions and \( ECstate \times EI \times DI \rightarrow ECstate \times EO \times DO \); \( ECTCond \) is a set of EC transition conditions; \( ECAction \) is the EC state actions; \( ECstate_0 \) is the initial EC state.
After both formal models are defined, now the redesign process starts with the mapping functions between ECC and FSM. The mapping table is given as following:

Table 2: Mapping functions between ECC and FSM.

<table>
<thead>
<tr>
<th>FSM Element</th>
<th>ECC Element</th>
<th>Mapping Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Event and Data Input</td>
<td>ζ_{input} : Σ → EI × DI</td>
</tr>
<tr>
<td>Output</td>
<td>Event and Data Output</td>
<td>ζ_{output} : Γ → EO × DO</td>
</tr>
<tr>
<td>State</td>
<td>EC State</td>
<td>ζ_{state} : S → ECstate</td>
</tr>
<tr>
<td>Initial State</td>
<td>Initial EC State</td>
<td>ζ_{initstate} : S_0 → ECstate_0</td>
</tr>
<tr>
<td>State Transition</td>
<td>EC Transition</td>
<td>ζ_{tran} : δ → ECTrans</td>
</tr>
<tr>
<td>Output Function</td>
<td>EC Action</td>
<td>ζ_{action} : ω → ECAction</td>
</tr>
</tbody>
</table>

A PLC input could be a variable which is accessible directly from the PLC global memory. In the converted function block design, those inputs are listed as the data inputs in the interface. The solution for storing global variables in function blocks is suggested in the section 4.4.

The state machine can be designed per the PLC code if the original program is written in SFC. Or if the PLC code is written in a textual language, the state machine can be reverse engineered from it.

The ECC of the function block created for the PLC FSM is formed as below:

1) Map items one by one according to the Table 2;
2) An extra input event REQ shall be created, associated with all data inputs and used to “clock” the event part of the EC transition conditions.
3) An extra output event CNF shall be created and triggered by all states, associated with all data outputs and used to trigger the event in the EC state output event.

When this REQ event occurs, all input data variables will be updated first. Then the ECC will be evaluated. The event part of the EC transition condition must be activated and the Boolean expression of the data part also must be fulfilled in order to switch to the corresponding state in ECC. Finally, each state entry action of the PLC
code is directly copied into the corresponding EC state algorithm. The redesign process is illustrated on converting a function block in Figure 3.2.

![Figure 3.2: Redesign process for the object-oriented re-design pattern.](image)

The original PLC function block represents a simple state machine written in structure text as shown on the right hand side. The function block has two inputs IN1 and IN2 and two outputs OUT1 and OUT2. There is a FSM with three states (START/S1/S2) inside this function block. The state S1 and S2 are associated with an action OUT1 and OUT2 respectively. A Boolean variable (IN1/IN2) is used as the transition condition for the state S1 and S2. The redesigned IEC 61499 FB version with an equivalent ECC is shown in the left hand side. Two event inputs (INIT/REQ) and two event outputs (INITO/CNF) are introduced in the interface for the new function block. The event pair INIT and INITO is for clear all data inputs and outputs during the initialization stage of this function block. After execution starts, those data inputs are only updated when the event input REQ is raised. After the execution is completed, the event output CNF is emitted with all refreshed data outputs. Each FSM state in the original FB is mapped to an EC state in the new function block. The actions of state S1 and S2 are transformed to the EC state action S1_REQ and S2_REQ. The logics inside the actions switching OUT1 and OUT2 on and off are directly put into the S1_REQ and the S2_REQ. Different from the original FB, the
new EC state transition conditions are bounded with the input event REQ. The event REQ is triggered by a service interface function block source. This SIFB could be an E_CYCLE SIFB for cyclical polling or input module SIFB which generates events when an input value is toggled.

### 3.2.2 Reuse PLC code in Basic Function Block Algorithm

Also there is another approach for the object-oriented approach. In this method, both finite state machine and sequential logic style PLC programs tend to be reused directly. The entire PLC code inside a POU like function block or function, is encapsulated into a single EC state algorithm. The mapping between PLC inputs/outputs and function block event and data inputs/outputs is still the same. When the input event REQ is triggered by a SIFB, the basic function block will transition into the state REQ and process the algorithm associated. After the algorithm is executed, the event output CNF is emitted and data outputs shall be updated. The FB interface is identical with that of the previous model as presented in Figure 3.3.

![Redesign Pattern Diagram](image)

**Figure 3.3:** Redesign process for the object-oriented reuse pattern.

But there are only two EC states - START and REQ inside the ECC. When data inputs are updated from the I/O module, this FB wakes up from the START state and
executes the REQ algorithm associated with the REQ state. Once the process is completed, the FB returns to the START state where it remains idle until the REQ event is triggered again. Figure 3.3 illustrates the reuse solution with the same PLC function block.

The new IEC 61499 reused version of the function block has an identical interface compared to the converted version. Inside the new function block, all the algorithms defined in ST in the original FSM are copied into the REQ state algorithm. The data outputs are emitted in accordance with the current state.
3.3 Class-Oriented Design Pattern

After the object-oriented design pattern is introduced, another design pattern – class-oriented is presented. The class-oriented design pattern is inspired by the concept of Service-Oriented Architectures (SOA). SOA comprises unassociated, loosely coupled units of functionality that have no calls to each other and groups in the same service [84]. The basic function block is still used as a fundamental programming unit in the class-oriented approach. Different from the object-oriented approach, here the function block represents a class or group of devices performing the same service. In the class-oriented approach, a single instance of this function block encapsulates all the devices of this class or a group of such devices. In other words, all data and algorithms required for providing one particular service are encapsulated in one function block type. This model ensures each function block is able to provide a service without invoking any other function blocks or accessing to the global memory.

In the class-oriented design pattern, only one function block instance is created for representing a service instead of creating an instance for each device as in the object-oriented model. This service covers a number of devices which provide the same functionalities or all devices performing together the same service. Considering individual devices as the basic component in the object-oriented approach, a group of devices contributing to the same service is considered as the basic component in the class-oriented approach. The function block interface is similar to the object-oriented model. The difference is that each data input is defined as an array as shown in Figure 3.4 (with 32 elements in this particular example). When the I/O Service Interface Function Blocks (SIFB) capture the new values for those inputs, those values will be grouped and sent to the class-oriented function block instance. Inside this FB, there is one additional EC state named PROCESS compared to the object-oriented reuse version. When the REQ event raises, the ECC moves its current state to the REQ state and updates all data inputs’ values. Once the update process is completed, a list of device instances which have updated input data values is created. ECC executes the REQ state in a loop for each instance in the list. In an iteration of the loop, the REQ state algorithm is executed for a device instance data located in the function block’s internal memory. This is achievable in two methods: copy the instance data in the PROCESS state and then move to the REQ state or use an array for device instance,
pass the index number to the REQ state. Provided that the PLC code has been already designed in a modular way and there is no direct use of variables of other devices within the code of a single device [85] (instead they are referred to via a kind of reference table), the original PLC code can be reused in the REQ state algorithm with only minor modifications for inserting the array indexes.

![Diagram](image)

**Figure 3.4:** Redesign process for the class-oriented pattern.
3.4 Data Handling in the Redesign Process

Other than the system architecture and algorithms conversion, data handling efficiency is another critical aspect of the redesign process. In the IEC 61131-3 standard, there are two catalogues of data types: predefined data types including Bit, String, Integer, Real and etc and user defined data types. The predefined data types can be used for scalar variables or for elements of arrays. User defined data types are customized based on the combination of predefined data types and other user defined data types. Variables must be declared in the symbol list with a proper scope (associated with controller or program) before being used. The global scope variables are accessible anywhere in PLCs, while local variables are only accessible in a certain scope (program or function block) where they are declared. I/O variables are a special type of global variables directly mapped to actual inputs and outputs modules. Global variables are usually designed for constants and variables that must be shared in multiple tasks, programs, functions or function blocks. Although it is convenient for PLC developers to list all variables in global memory, the use of local variables represents a better programming style. It also helps to avoid data being corrupted accidentally by other tasks in the same PLC.

The IEC 61499 standard complies with all predefined data types which are defined in IEC 61131-3 as well as supports user define data types. The concept of an adapter type is suggested in IEC 61499 to gather various data and event links in a group to avoid “spaghetti” connections in FB diagrams. Due to the orientation on distributed control system, there is no any form of central database or global variables in the IEC 61499 standard. All data variables in IEC 61499 function blocks must be declared in a basic or stored in a service interface function block. In an IEC 61131-3 PLC function block implementation, a higher level variable from global or program memory could be declared as InOut Parameter (input and output) or an address reference which allows variables pass as pointers into lower level function blocks, modifying values of these variables inside the function blocks and returning back to the global or the program memory. Such arrangement is not allowed in the IEC 61499 standard. So the corresponding variables shall be separated as a pair of an input variable and an output variable. In IEC 61499 function block networks, variables’ values could be hold in basic or service interface function blocks. When a specific data variable is requested by another function block, this block shall hand over the value of this variable to the
destination function block by passing an update event signal. After the value is modified by that function block, the new value must be passed back to the owner function block using events.

Another widely used artefact in the PLC programming is timers. IEC 61499 provides a predefined service interface FB E_DELAY for timers. When time is count up to a pre-set value, E_DELAY will emit an event to indicate the pre-set time is elapsed. However, this FB cannot be used in a basic function blocks’ algorithm. One of the solutions is to provide a timer instruction in format of the IEC 61131-3 programming languages which could be invoked in the basic function block algorithm. Alternatively, a global timestamp can be applied to the basic function blocks. Instead of providing a “done” event of the timer, a timestamp in numeric format is updated when this function block is triggered. The minimum scale of this timestamp depends on the actual FB runtime can produce. When any input event rises, the value of the global timestamp is passed into the target basic function block. By manipulating the counter value from the start timestamp, handle timers inside basic function blocks can be achieved. If there is a requirement of time synchronization in the distributed FB system, the IEEE 1588 standard could be used as a solution to synchronize all timestamps of all nodes connected via the Ethernet [86].
In Figure 3.5, both PLC subroutines A and B require a global variable – INT stored in the global memory. This tag is manipulated in sequence by those subroutines. There are two possible mapping solutions in the IEC 61499 version. First is to mapping global memory to a special function block DB. This special DB FB is implemented as a central storage for global variables and constants. Due to the fact that in IEC 61499 it is not allowed to connect more than one data arc to a data input of
Redesign Patterns

a function block, the DB FB has two separate “channels” for the value setting. Two data inputs INT1 and INT2 and two event inputs SET1 and SET2 are implemented. On the other hand, if these FBs were communicating using the pairs of PUBLISH-SUBSCRIBE FBs, it would be possible to have just one setting channel in DB. This option suits systems distributed across several devices better. The need for explicit insertion of communication FBs is sometimes criticized for cluttering FB design. Despite these simplifications, explicit implementation of global variables in IEC 61499 could be still complicated. It requires several messages to request a variable and return its modified value back to the storage FB. However, one should note that there is no better solution achievable in a distributed system. Hiding this complexity from the user is the best option a smart software tool could be done.

In the following subsections, the global and local data handling are considered in both the object-oriented and class-oriented approaches.

3.4.1 Data Handling in Object-Oriented Approach

In the high level object-oriented programming languages (e.g. C++ or Java), variables are defined in a proper scope (either in a specific class or in global). Each variable could be given an access level including public, private and protected [87]. The overall time for read/write variables in a PLC is given as:

$$T_{PLC RW} = T_{PLC alg}$$

Where $T_{PLC alg}$ is the time for process the variable.

In the IEC 61499 terms, data variables must be encapsulated inside a basic or service interface function block, and those variables will be read or written via internal algorithms, are only accessible via function block interfaces. For example, to modify the value of a variable inside another FB, an event is required from both ends of the function blocks. The overall time $T_{RW}$ for a read/write process for a variable is given as:

$$T_{RW} = T_{event} \times 2 + TT_{alg} + TO_{alg}$$

Where $T_{event}$ is the event propagation time between two function blocks; $TO_{alg}$ is the time required for executing the algorithm in the source function block; $TT_{alg}$ is the time required for executing the algorithm in the target function block;
More complex data structures are less efficient. Thus, for two- or more-dimensional arrays with indirect addressing for the array index in PLC application, the execution time is significantly longer than that of a flat data structure since for each array element its offset in the array needs to be calculated. This time delay increases exponentially with more dimensional arrays. This is also applied to the object-oriented design in the IEC 61499. The nested structures are applied through the entire IEC 61499 applications which will also seriously affect the execution time \( T_{Oalg} \) and \( TT_{alg} \).

For each global variable in the PLC, an equivalent variable is required to be implemented in a FB which is connected to wherever requests for read. Also whenever a write request rises, an event is generated to this function block which owns the variable. This is not only required for global variables but also applies to any variable in the nested function block networks.

Compared to the PLC, the difference between times required for read/write a variable is given as:

\[
T_{\text{diff}} = T_{RW} - T_{PLC_{RW}}
\]

\[
T_{\text{diff}} = T_{\text{event}} \times 2 + TT_{alg} + T_{Oalg} - T_{PLC_{alg}}
\]

Assume the target is to read/write variable faster than PLC does,

\[ T_{\text{diff}} < 0 \]

\[ T_{\text{event}} \times 2 + TT_{alg} + T_{Oalg} - T_{PLC_{alg}} < 0 \]

\[ T_{\text{event}} \times 2 + TT_{alg} + T_{Oalg} < T_{PLC_{alg}} \]

If both controllers run at a similar frequency, the time required for process the read/write request will be similar. So assume \( TT_{alg} + T_{Oalg} \approx T_{PLC_{alg}} \) now the time difference is \( T_{\text{event}} \times 2 \) which is obviously greater than zero. So unless the FB controller could run faster than the PLC, the PLC reads/writes variables faster than a FB controller using the object-oriented approach. This is experimented in the case study presented in section 8.6.

### 3.4.2 Data Handling in Class-Oriented Approach

In the class-oriented design pattern, all reusable and interoperable functions are grouped into a single function block. No data handover between blocks as the object-oriented approach is needed. All data required by a group of devices or services is
kept locally in the corresponding function block. The requirements of having global variables and constants are minimal. As function blocks are grouped by functionalities, the number of read/write variable between function blocks is reduced. This event will only be emitted when a variable required by another service or a device in another group. The overall time $TCO_{RW}$ for process read/write request is given as:

$$TCO_{RW} = TCO_{alg}$$

Where $TCO_{alg}$ is the time required for process the read/write request.

Again, assume the PLC and the FB controller has similar clock frequency, the overall processing time for two controllers is almost the same where $TCO_{RW} \approx TPLC_{RW}$.

However, when this concept applies to composite function blocks, performance will drop to the level of the object-oriented approach. As no internal memory is available in composite function blocks, a basic or service interface function block is needed for storing the variables in this CFB. Data still have to be passed around function blocks inside the composite function blocks. Debugging of FBs designed according to the class-oriented approach is more difficult than of their object-oriented counterparts. The instance data of a particular device is not available directly to the user for viewing. However, the class-oriented model handles data more efficient compared to the object-oriented model.
### 3.5 Redesign Approaches Performance Estimation

An important question is what will be the performance gain or loss if redesign PLC systems into function blocks. The performance can be measured using the reaction time to an external event as illustrated in Figure 3.6 for both the IEC 61131-3 PLC and the IEC 61499 FB. The worst case reaction time is observed when an input event triggers one PLC but the output event is generated by another PLC. The main performance gain in the IEC 61499 case comes from eliminating scan cycles used in PLCs: assuming actual algorithm execution times and propagation times of both versions are identical, the PLC version has to wait until next scan before the data from another PLC can be updated.

![Diagram](image)

**Figure 3.6:** Performance comparison between IEC 61131-3 and IEC 61499.
The maximum reaction time (MRT) of an application distributed across over the PLC network is defined as:

\[ T_{total} = \sum_{i=1}^{n} (TIN_i + TSCAN_i + TOUT_i + TPROG_i + TFB_i) \times 2 \]  

where \( TIN_i = \{TIN1, TIN2, \ldots, TINn\} \) is a set of individual PLC input update times; \( TSCAN_i = \{TSCAN1, TSCAN2, \ldots, TSCANn\} \) is a set of individual PLC task execution times; \( TOUT_i = \{TOUT1, TOUT2, \ldots, TOUTn\} \) is a set of individual PLC output update times; \( TPROG_i = \{TPROG1, TPROG2, \ldots, TPROGn\} \) is a set of individual propagation times across the field bus; \( TFB_i = \{TFB1, TFB2, \ldots, TFBn\} \) is a set of individual field bus scan input/output times.

The MRT of the equivalent application in IEC 61499 is derived as:

\[ T_{total} = \sum_{i=1}^{n} (TSCAN_i + TPROG_i) \]  

where \( TSCAN_i = \{TSCAN1, TSCAN2, \ldots, TSCANn\} \) is a set of individual IEC 61499 device execution times; \( TPROG_i = \{TPROG1, TPROG2, \ldots, TPROGn\} \) is a set of individual propagation times across the field bus. From these formulae one sees that the PLC reaction is greater by \((TIN_i + TOUT_i) \times 2\).

Communication over field buses or inter-PLC network is also based on the scan cycles \((TFB_i)\) therefore an extra scan time has to be waited until the data can be copied into the PLC. As the IEC 61499 standard is based on the event-triggered communication, it is assumed that execution of FBs will start immediately whenever events arrive. Finally as most of the data is stored locally in function blocks of IEC 61499 applications, no access to global memory is required. This will also provide a performance gain. However, encapsulating global data into FBs is quite time-consuming job.
Redesign Patterns

The estimated execution time of the PLC program and 3 approaches of the FB programs based on the formula (3.1) and (3.2) using the case study baggage handling system presented in the section 8.1 are illustrated in Figure 3.7. From the figure, when more distributed nodes are introduced into the system, the PLC reaction time is increasing larger than the reaction time of the function block version.

![Estimated Execution Time on Distributed Nodes](image)

Figure 3.7: Performance Estimation between IEC 61131-3 and IEC 61499.
3.6 Comparison between Approaches and Limitations

In this section, all three approaches including object-oriented reuse approach, object-oriented converted approach and service-oriented approach are compared regarding several factors: redesign efforts, reaction time and robustness.

From the redesign effort point of view, the object-oriented convert approach needs lots of human efforts on converting FSM type PLC code into the ECC. Redesigning FSM to ECC manually may insert some human errors into the result function block system if the original PLC program is not fully understood by developers. The object-oriented reuse approach allows the original PLC to be reused completely without understand the original PLC code at all. This could be done automatically without any human effort. The class-oriented approach is sit between two object-oriented approaches, some human efforts are required.

Secondly, compare from the reaction time perspective, the three approaches has better performance compared to the PLC if the system distributed on more than two nodes as discussed in the previous section. From the data handling perspective, the class-oriented pattern avoids the issue of stores variables in a separate function block which communicates with every block needed. For the overall performance, the class-oriented approach should have the best performance.

From the robustness view, the object-oriented reuse version is the best option due to the original PLC code is reused without any modification. Although the class-oriented pattern only requires minimal changes, it is still possible to introduce some issues during the manual transformation.

Finally, there are some limitations of the proposed approaches. For the object-oriented reuse approach, the libraries and built-in instructions used in the PLC program must exist in the target function block development environment. Secondly, converting an IEC 61131-3 FSM to IEC 61499 ECC using sentence by sentence code translation may cause the behaviour change due to various IEC 61499 execution semantics. For the class-oriented approach, the PLC code can only be reused inside basic function blocks. To reuse PLC code in composite function blocks, manual translation from IEC 61131-3 programming languages to IEC 61499 FB diagram is required.
3.7 Summary

To summarize the redesign patterns of applying the IEC 61499 standard in distributed control systems, some mappings of terms for the migration process are proposed, for example, FSM to ECC. Both predefined and user-defined data types of IEC 61131-3 are reused in IEC 61499 without any change except global variables.

Three redesign approaches proposed in this paper cover two PLC code styles. Each approach suits a certain PLC code style. The general rules of when to apply each approach are provided as follows.

In both object-oriented design approaches, each physical device is represented by a function block instance. Those approaches fit well to the vision of distributed control systems. Each part of the physical machinery could be equipped with its own embedded controller running the corresponding controller FB as a distributed node. Interconnections between the object-oriented nodes reflect the system physical layout. From the algorithm perspective, if the FSM style code is recoverable from the original IEC 61131-3 program, it can be converted to an ECC in the IEC 61499 system configuration. Otherwise, the original PLC code could be copied directly into the EC state algorithms. All IEC 61131-3 programming languages (FBD, LD, IL and ST[1] as well as SFC) can be reused in IEC 61499 basic FB algorithms if those languages are supported by target IEC 61499 IDEs. Alternatively, FBD can be transformed into the composite FB format if the original FBs from the FBD are available in the library of the target IEC 61499 tool. The SFC code can be converted into an ECC by inserting a basic FB. The combination of redesign and reuse PLC code provides a shorter learning curve for PLC developers with minimum time and cost. This approach is more suitable for application in process control, machine/robot control systems and motion control systems which are relied on state-machine based control.

The class-oriented pattern accommodates a distributed node that is responsible for a group of devices with same functionality. The PLC code is reused as algorithm in BFB of new system configuration but some minor modifications are required. The overall structure of function blocks network is much tidier compared to the object-oriented architecture. The class-oriented design clearly represents distribution of system functionalities which simplifies their reuse. This design pattern suits systems which have large quantities of devices performing a similar service (for example, conveyors) or any systems requiring massive data processing better.
Redesign Patterns

In order to achieve automatic migration process, the object-oriented reuse design pattern is the best option as it requires least human efforts and is most robust solution overall.
4 Migration Models and Rules

4.1 Introduction

This Chapter defines the migration models and generic mapping rules based on the patterns introduced in the previous Chapter. The migration process proposed in this research work is not trying to replace the IEC 61131-3 standard with the IEC 61499 standard entirely. This complies with the vision of many industry practitioners (e.g. [88]) who see IEC 61499 as a complement rather than a substitute to IEC 61131-3. Instead, the IEC 61499 Standard could be used as the top level design POU in conjunction with low level control codes written in the programming languages defined in the IEC 611131-3 standard. The aim for introducing automatic migration process is to reduce the redesign time significantly. In order to achieve automatic migration, entire IEC 61131-3 PLC programs are encapsulated in the IEC 61499 function blocks and reused in the new IEC 61499 system configuration according to the object-oriented reuse pattern proposed in the previous section. In order to achieve cyclical execution behaviour of the original PLC applications in the generated IEC 61499 systems, the formal IEC 61131-3 model is required. The IEC 61499 formal model used here is very similar to the work presented by Dubinin et al. [89]. In order to distinguish between the IEC 61131-3 and the IEC 61499 models, the symbols used in [89] to present IEC 61499 formal models are renamed in this chapter. The equivalent execution model in IEC 61131-3 is also needed according to the PLC execution model. Once both the IEC 61131-3 standard and the IEC 61499 standard models are defined, the mapping rules between those two models can be provided.

The rest of the section is allocated as follows. In the section 4.2, models for migration of both the IEC 61131-3 standard and the IEC 61499 are provided. A scheduling function block set for achieving cyclical execution in the IEC 61499 is described. Finally, the mapping rules between two models are given in the section 4.4.
4.2 Models for Migration

The models defined in this section are majorly for migration. As a result, the IEC 61131-3 formal model proposed here is not covering every element defined in the IEC 61131-3 standard. Some of the features are not relevant for migration will be ignored. Also for the IEC 61499 formal model, only the portion of the model used in the mapping rules is provided. The complete formal IEC 61499 model can be found in [89].

4.2.1 IEC 61131-3 Formal Model

First, the fundamental elements of the IEC 61131-3 must be declared in the formal model. The IEC 61131-3 basic high level language elements and their relationships are illustrated in Figure 4.1.

In an IEC 61131-3 system configuration, each resource corresponds to a physical device – PLC CPU. Every PLC may be scheduled with one or more tasks which contains one or more programs. Inside each program, functions and function blocks can be invoked from any program of any task in the same resource. Programs, functions and function blocks are fundamental programming organization units (POUs) of IEC 61131-3. Nested structures of POU calls are also supported. Five programming languages are defined in IEC 61131-3. Graphical languages includes ladder logic diagram (LD), function block diagram (FBD) and sequential function chart (SFC). Structure text (ST) and instruction list (IL) are textual based languages. POUs are written in one of those languages.
The transformation is applied to PLC project which is defined as a set of resources. For the purposes of formal definition of transformations, an IEC 61131-3 resource belongs to the project is defined as a tuple:

\[ Res = (Task, GlobalVar, UseTG, UpdTG) \]  

Where \( Task = \{task_1, task_2, ..., task_n\} \neq \emptyset \) is a non-empty set of tasks scheduled in the PLC; \( GlobalVar = \{globalVar_1, globalVar_2, ..., globalVar_m\} \) is a set of global variables of the resource; \( UseTG \subseteq Task \times GlobalVar \) is a relation of using the global variables in the tasks (for reading); \( UpdTG \subseteq Task \times GlobalVar \) is a relation of changing the global variables in the tasks.

Each task is defined as a tuple:

\[ task = (Program, LocalVar, UsePL, UpdPL) \]

Where \( Program = \{program_1, program_2, ..., program_p\} \) is a non-empty set of programs scheduled in a PLC task; \( LocalVar = \{localVar_1, localVar_2, ..., localVar_s\} \) is a set of local variables used in one or more PLC program of this task; \( UsePL \subseteq Program \times LocalVar \) is a relation of using the local variables in the programs (for reading); \( UpdPL \subseteq Program \times LocalVar \) is a relation of changing the local variables in the programs. There must be some variables defined in the resource. This can be expressed as \( GlobalVar \cup \bigcup_{i} LocalVar^i \neq \emptyset \).

For \( task_i \) and \( task_j \) \((i \neq j)\), program must be associated to only one task can be represented as: \( Program^i \cap Program^j = \emptyset \). Here the upper indexes \( i \) and \( j \) are used to distinguish between various tasks.

The next level of the PLC code hierarchy is routines, functions and function blocks inside a program. Function or function blocks can be invoked along with algorithms inside a PLC program:

\[ Program = (Alg, FB, Func) \]

Where \( Alg \) is representing the actual PLC algorithms consists of statements and operators written in one of the five programming languages in this particular program not including hierarchy levels of functions or function blocks invoked; \( FB = \{FB_1, \)
Migration Models and Rules

\( FB_2, \ldots, FB_k \) is a set of function block instances used in the program; \( \text{Func} = \{ \text{Func}_1, \text{Func}_2, \ldots, \text{Func}_h \} \) is a set of functions instances used in the program.

A function block is defined as a 3-tuple:

\[
FB = (\text{Interface}, \text{AlgFB}, \text{Var}), (4.4)
\]

Where \( \text{Var} = \{ \text{var}_1, \text{var}_2, \ldots, \text{var}_x \} \) is a set of internal variables used in the function; \( \text{AlgFB} \) is an algorithm incorporated into \( FB \).

Function block interface is defined as:

\[
\text{Interface} = (\text{VI}, \text{VO}, \text{VIO}, \text{EN}, \text{ENO}), (4.5)
\]

Where \( \text{VI} = \{ \text{vi}_1, \text{vi}_2, \ldots, \text{vi}_r \} \) is a set of input variables; \( \text{VO} = \{ \text{vo}_1, \text{vo}_2, \ldots, \text{vo}_q \} \) is a set of output variables; \( \text{VIO} = \{ \text{vio}_1, \text{vio}_2, \ldots, \text{vio}_t \} \) is a set of in/out variables; \( \text{EN} \) is a Boolean type input variable to indicate the function block activation status; \( \text{ENO} \) is a Boolean type output variable to indicate the function block has operated successfully.

After the basic elements of IEC 61131-3 are defined, the execution function using those elements is also necessary for the migration process. The IEC 61131-3 PLC execution is based on scan cycles [1]. As shown in Figure 4.2, during each scan, a PLC runs the communication service first which reads all inputs variables from input modules, then executes through all tasks and updates all outputs data to the output modules finally. The execution order in this model is sequential. A new execution cycle starts immediately once the previous cycle is completed.

![SINGLE PLC SCAN CYCLE](image)

Figure 4.2: IEC 61131-3 Execution Semantics Timing Diagram.

The formal IEC 61131-3 execution model for a PLC resource is given as a tuple:

\[
M = (\text{Res}, \text{If}, \{ \text{Tf}_i \}_{i=1}^{n}, \text{Of}), (4.6)
\]

Where \( \text{If} \) is a function servicing input data from the input PLC modules; \( \text{Of} \) is a function updating data to the output PLC modules; \( \text{Tf}_i \) is a task execution function (\( i = 1, n \)) which consists of multiple program execution functions \( \text{Pf}_j (j = 1, p' \)).
The global variables can be divided into three classes:

\[ \text{GlobalVar} = \text{InputVar} \cup \text{OutputVar} \cup \text{UDVar}. \]

and

\[ \text{InputVar} \cap \text{OutputVar} \cap \text{UDVar} = \emptyset, \]

Where \( \text{InputVar} \) is a set of variables mapping from the PLC input modules; \( \text{OutputVar} \) is a set of variables mapping to the PLC output modules; \( \text{UDVar} \) is a set of variables defined by the users to be used in multiple programs.

The essence of function \( \text{If} \) is to sample the values of PLC data inputs into \( \text{InputVar} \). Function \( \text{Of} \) “copies” the values of variables from \( \text{OutputVar} \) back to PLC output modules.

There are two types of tasks for executing POU as defined in the IEC 61131-3 standard: periodic tasks and continuous tasks. Periodic tasks execute at a customizable non-zero time interval. Continuous tasks have a zero time interval keep looping through all associated programs until interrupted by periodic tasks. Each task also has a different preset priority level. There are also two types of PLC runtimes: preemptive and non-preemptive. In a preemptive PLC runtime environment, a lower priority periodic task will be interrupted by a higher priority periodic task when the execution of this lower priority task cannot be accomplished before the next round of the higher priority task starts. Once the higher priority task terminates, the lower priority task will resume execution at the place where it was interrupted. In a non-preemptive PLC runtime environment, the higher priority task will wait until the lower priority task terminates. If two tasks have same priority and all waiting for execution, the task with longer waiting time will have the priority. For the continuous tasks, the priority means the execution order. Higher priority continuous tasks always execute first followed by the lower priority continuous tasks. The execution of the continuous task will resume from the place where it was interrupted as well. Given an IEC 61131-3 resource \( \text{Resource} \) as defined in (4.1), the task execution function for \( \text{task}_i \in \text{Task} \) is defined as

\[ T_f^i = \bigcup_j P_f^i_j. \]
The execution function $Pf^i_j$ of the program $j$ in the task $i$ is defined as

$$Pf^i_j : \prod_{(\text{task}_i, \text{globalVar}) \in \text{UseTG}} \text{Dom(globalVar)} \times \prod_{(\text{program}_j, \text{localVar}) \in \text{UsePL}} \text{Dom(localVar)} \rightarrow \prod_{(\text{task}_i, \text{globalVar}) \in \text{UpdTG}} \text{Dom(globalVar)} \times \prod_{(\text{program}_j, \text{localVar}) \in \text{UpdPL}} \text{Dom(localVar)}$$

The algorithm for a PLC execution scan is given as:

1: for all task$_i \in $ Continuous Task do
2: \hspace{1em} for all program$_j \in $ Program$^i$ do
3: \hspace{2em} if task$_i$ is interrupted by task$_k \in $ Periodic Task then
4: \hspace{3em} for all program$_m \in $ Program$^k$ do
5: \hspace{4em} Compute Pf$^k_m$
6: \hspace{3em} end for
7: \hspace{2em} end if
8: \hspace{1em} Compute Pf$^i_j$
9: \hspace{1em} end for
10: end for

The scan time of a PLC cycle time is defined as:

$$T_{scan} = \sum_i (x_i \times TP_1) + \sum_j TC_j + T_{comm}$$

Where $T_{comm}$ is also the time of servicing communications and I/Os; $TP_i$ is $i$-th individual periodic task execution time; $TC_j$ is $j$-th individual continuous task execution time; $x_i$ is representing the number of times periodic tasks execute.

In every PLC cycle, all scheduled continuous tasks will be executed exactly once. The periodic tasks may execute zero or more times depends on duration of each PLC scan cycle time. If the scheduled continuous tasks execute fast enough, the periodic tasks may only be taken place every several scan cycles. Or if the scheduled continuous tasks execute extremely slow, the periodic tasks may execute several times before the scan terminates.
In a task, there are multiple programs executed in a sequential sequence. The total execution time of a task is:

\[ T_{task} = \sum_i T_{pro_i} \]

Where \( T_{pro_i} \) is the execution time of the \( i \)-th individual program.

### 4.2.2 IEC 61499 Formal Model

After the IEC 61131-3 model is defined, now it is the turn for the IEC 61499 standard. An IEC 61499 system configuration \( S \) is defined as a tuple:

\[ S = (\text{Dev}, \text{Seg}, \text{App}, \text{Map}) \] (4.7)

Where \( \text{Dev} = \{\text{Dev}_1, \text{Dev}_2, ..., \text{Dev}_n\} \) is a set of device instances; \( \text{Seg} \) is the network segment for this IEC 61499 system configuration; \( \text{App} = \{\text{App}_1, \text{App}_2, ..., \text{App}_n\} \) is a set of applications defined. \( \text{Map} = \{\text{Map}_1, \text{Map}_2, ..., \text{Map}_n\} \) is a set of mappings of function blocks between applications and devices.

After the system configuration is defined, the next level of IEC 61499 element is a device. An IEC 61499 device \( \text{Dev} \) is defined as a 2-tuple:

\[ \text{Dev} = (\text{Resr}, \text{FBN}) \] (4.8)

Where \( \text{Resr} = \{\text{Resr}_1, \text{Resr}_2, ..., \text{Resr}_y\} \) is a set of resources used in the device; \( \text{FBN} \) is a function block network.

An IEC 61499 function block network \( \text{FBN} \) is defined as a 3-tuple:

\[ \text{FBN} = (\text{FBI}, \text{EConn}, \text{DConn}) \] (4.9)

Where \( \text{FBI} = \{\text{FBI}_1, \text{FBI}_2, ..., \text{FBI}_n\} \) is a set of function block instances defined in a function block network; \( \text{EConn} \) is a set of event connections; \( \text{DConn} \) is a set of data connections.
There are three types of function blocks available in the IEC 61499: Basic, Composite and Service Interface. An IEC 61499 composite function block $CFB$ is defined as:

$$CFB = (\text{Interface}, FBN) \quad (4.10)$$

Where $FBN$ is a function block network.

An IEC 61499 basic function block (BFB) is defined as:

$$BFB = (\text{Interface}, ECC, Alg, IntData), \quad (4.11)$$

Where $\text{Interface}$ is the function block interface (including inputs/outputs event and data); $ECC$ is representing an execution control chart in the basic function block; $Alg$ is a set of algorithms associated with EC states; $IntData$ is a set of internal variables only used in this basic function block.

The IEC 61499 execution control chart $ECC$ inside a basic function block is defined as :

$$ECC = (\text{ECstate}, ECTrans, ECTCond, ECAction, ECstate_0) \quad (4.12)$$

Where $\text{ECstate}$ is a set of EC states; $ECTrans$ is a set of EC transitions and $ECstate \times EI \times DI \rightarrow ECstate \times EO \times DO$; $ECTCond$ is a set of EC transition conditions; $ECAction$ is the EC state actions; $ECstate_0$ is the initial EC state.

Now all IEC 61499 elements are ready for mapping.
4.3 Cyclical Execution Scheduling in IEC 61499 FBs

Before starting migration process, there is one more important issue needs to be resolved - the execution semantics. In the migration approach proposed in the earlier section, the execution semantics of PLC is recreated by means of IEC 61499. This includes not only cyclical execution model which is achievable by some function block runtimes (e.g. ISaGRAF [90]), but also other PLC key features such as priority of tasks and preemptive between periodic and continuously executed tasks. Different from the IEC 61131-3 standard, the IEC 61499 standard has a variety of execution semantics [64, 66, 91]. Three of them are most known as: sequential, parallel and synchronous. In the sequential semantics only one function block at one time is executed. The parallel semantics allow multiple function block chain to be activated simultaneously. In the synchronous semantics blocks are running based on the concept of tick. Each tick all activated function blocks will execute once. In order to apply the IEC 61131-3 execution model in all those runtime semantics, a scheduler system in function blocks is built on the application level. The hierarchy of this proposed scheduler system given in Figure 4.3.

![Figure 4.3: IEC 61499 Scheduler System for IEC 61131-3 Execution Semantics.](image)
This scheduler system ensures PLC cyclical behaviour in the function block without considering what execution semantics is used – only one function block of a program in a task is activated at one time. There are three layers of schedulers: a PLC main scheduler, a task scheduler and a program scheduler.

The PLC main scheduler controls the execution order of tasks. The main scheduler is implemented as a basic function block and its interface, execution control chart (ECC) and algorithms are illustrated in Figure 4.4 for a preemptive PLC scan.

![Main Scheduler Interface, Execution Control Chart (ECC) and scheduling algorithms (for preemptive PLC execution semantics).](image)

**REQ ALGORITHM**

```
TIMESTAMP = TIMESTAMP + 1;
if TIMESTAMP <= SCAN_TIME then
    if TIMESTAMP < SCAN_TIME
        then
            P1_ENABLE = NOT(P1_ENABLE) & NOT(P2_ENABLE) & NOT(P3_ENABLE) & NOT(P4_ENABLE);
            P2_ENABLE = TRUE;
            P3_ENABLE = TRUE;
            P4_ENABLE = TRUE;
    else
        END IF;
        END IF;
        END IF;
        END IF;
```

**P1_DONE ALGORITHM**

```
P1_ENABLE = FALSE;
```

**P2_DONE ALGORITHM**

```
P2_ENABLE = FALSE;
```

**INIT ALGORITHM**

```
TIMESTAMP := 0;
P1_ENABLE_ONS := FALSE;
P2_ENABLE_ONS := FALSE;
P1_ENABLE = FALSE;
P2_ENABLE = FALSE;
```

Figure 4.4: Main Scheduler Interface, Execution Control Chart (ECC) and scheduling algorithms (for preemptive PLC execution semantics).
There are three tasks in this example. P1 and P2 are both periodic tasks where P1 has a shorter interval time and higher priority. In this case, P1 has a faster interval time of 20 milliseconds (ms) and higher priority than P2 which executes every 35 ms. C1 refers to a continuous task. Here C1 could be preempted by P1 and P2 several times in each scan.

The PLC main scheduler is implemented for this example with the following structure. Its execution control chart (ECC) includes four states: INIT, REQ, P1_DONE and P2_DONE. The core part REQ state is triggered by the TICK event that is raised by an E_CYCLE service interface function block (SIFB) (say, every millisecond in FBDK, this can be shorter if microsecond is supported for E_CYCLE in other platforms). An internal timestamp is counted up by one every millisecond as well. This timestamp is cleared in the INIT state once the system started. It is continuously counting up to the preset PLC scan time value from SCAN_TIME input then wrapping around back to zero again. There is a dedicated enable signal output for each task in the main scheduler.

When a task is scheduled to be executed, the corresponding enable bit data output is set. For example, when timestamp is counted up to 20ms, the task P1 will be activated by set P1_ENABLE to true. The P1 will be enabled without any extra condition due to its highest priority in the entire PLC configuration and preemptive PLC scan. For a non-preemptive PLC scan, P1 will only be enabled when no other task is enabled. A terminate event will be emitted from the P1 task scheduler when process for P1 task is ended. This P1_DONE event will trigger the P1_DONE state which resets P1_ENABLE signal back to false. Another internal Boolean variable P1_ENABLE_ONS is set once the P1 task is triggered every scan. It is reset by the end of the scan to ensure no continuous task is triggered more than once in single scan. Similar task trigger and task termination feedback mechanisms are implemented for P2 as well. When the timestamp is counted up to 35ms, it will activate the P2 task scheduler by setting the P2_ENABLE data variable to true. However the P2 execution will be held if the task P1 is processing due to its lower priority. The continuous task C1 will be enabled as long as no periodic task is enabled. If more tasks are scheduled in the original PLC project, a dedicated pair of input and output event, an enable Boolean output data variable and task complete state algorithm should be inserted.
Also in the REQ ant INIT state algorithms, the new task must be inserted according to the type, interval and priority of this task.

On the next level of the scheduling system, the task scheduler is designed for scheduling the execution order of associated programs in a PLC task. The standard interface, ECC and algorithms of the task scheduler function block are illustrated in Figure 4.5.

When the main scheduler enables a task, an event input REQ is received with ENABLE signal set to true. It will immediately activate the PROGRAM1 state and PROGRAM1_ENABLE signal is raised to trigger the PROGRAM1 program scheduler. Once the PROGRAM1 execution is completed and this task is still enabled by the main scheduler, PROGRAM2 will start execution immediately. Same as the main scheduler, the TICK event input is connected to the E_CYCLE SIFB updated every one millisecond. If this task is interrupted by another higher priority task during processing the PROGRAM1, the ENABLE bit will be reset back to false. The task scheduler will complete the PROGRAM1 execution and remain at PROGRAM1 state. When this program is scheduled to be executed in the next scan cycle, the ENABLE
bit will be set again and the task will resume from executing the PROGRAM2. Once all programs in the chain are processed, a done signal is sent back to the main scheduler to switch to the next scheduled task. The task scheduler will remain idle until it is activated again in the next scan. If there is a new program associated with this task in the original PLC project, a dedicated pair of data input and output variable and a state with algorithm must be added. In the COMPLETED algorithm, the enable bit of this state must be reset.

On the lowest level of the scheduling system, the Program scheduler is responsible for arranging the execution order of routines invoking functions and function blocks inside a particular program. The program scheduler function block is only required when there is more than one routine in this program. The program scheduler function block is very similar to the task scheduler function block. Instead of scheduling programs, the program scheduler enables routines according to the order defined in the original PLC program. The ECC state will be named following routines rather than using program names in the task scheduler.

The sequential execution semantics of any IEC 61131-3 configuration can be implemented in any IEC 61499 execution semantics with this scheduling system. A similar approach has been demonstrated and validated in [80].

In order to increase system performance and responsiveness, all the schedulers can be implemented in the form of SIFB rather than as basic FB.
4.4 Migration Rules

After the execution semantics are sorted out, the final step for the migration process is to map elements between two models. Before start migration process, there are some assumptions need to be declared.

The premises of the migration rules are:

1. Assume the instructions, built-in functions and function blocks used in the original PLC program are supported by the target IEC 61499 development environment.

2. Assume the original PLC program can be exported into the XML file format.

**Mapping Rule 1**

A system configuration \( S \) (4.7) is created for each PLC project.

\[
M_\alpha: P \rightarrow S
\]

The mapping function \( M_\alpha \) will be constructively defined through the following rules.

**Mapping Rule 2**

An IEC 61131-3 resource \( Res \in P \) (4.1) is mapped to an IEC 61499 device \( Dev \in S \) (4.7):

\[
M_\beta: Res \rightarrow Dev
\]

Each IEC 61131-3 project may have one or more PLCs configured. Every PLC controller module is considered as an individual resource. In the IEC 61499 domain, the proper mapping element is a device which also refers to a physical controller.

The mapping function \( M_\beta \) is defined through definition of the mapping function \( M_\gamma \) as follows.
Mapping Rule 3

An IEC 61131-3 task \( Task \in Res \) (4.2) is mapped to an IEC 61499 resource \( Resr \in Dev \) (4.8):

\[
M_\gamma: Task \rightarrow Resr
\]

In an IEC 61131-3 resource, there could be one or more tasks scheduled. Also in the IEC 61949 standard, each physical device can have one or more resources allocated. Here, every PLC task refers to a FB resource.

Now, the architecture transformation of an IEC 61131-3 application to an IEC 61499 system is completed. However, there are several issues on the code level still need to be resolved in the generated system.

The mapping function \( M_\gamma \) is defined through \( M_\delta, M_\epsilon, M_\zeta \) and \( M_\eta \) as follows:

Mapping Rule 4A

An IEC 61131-3 program \( Program = (Alg, FB, Func) \) (4.3) is mapped to an IEC 61499 composite function block type \( CFB = (Interface, FBN) \) (4.10):

\[
M_\delta: Program \rightarrow CFB \text{ and } FB \cup Func \neq \emptyset.
\]

If there is one or more function or function block invoked in a program, a composite function block is needed to encapsulate the algorithms in the original program.

Mapping Rule 4B

An IEC 61131-3 program \( Program \) (4.3) is mapped to an IEC 61499 basic function block \( BFB = (Interface, ECC, Alg, IntData) \) (4.11) if there is no function or function block is used inside a program:

\[
M_\epsilon: Program \rightarrow BFB \text{ and } FB \cup Func = \emptyset.
\]

If there is no function or function block calls in the original function block design, then a basic function block is used to encapsulate the algorithm instead of a composite function block.
PLC programs can be written in one of the five programming languages defined in the IEC 61131-3 standard. For a program written in ladder logic diagram (LD), instruction list (IL) or structure text (ST), the code can be copied directly into an EC state algorithm of a basic function block. However, there is one exception – the case, when another IEC 61131-3 function or function block is invoked in this program. This requires calling an instance of a function block inside a basic function block. This is not supported by the existing IEC 61499 tools and not specified in the standard itself. The solution is illustrated in Figure 4.6 below. If there is no function or function block instance called in the program, the entire logic can be reused into a Basic FB. Otherwise those function or function block instances are mapped to an inherited CFB and logics around function or function block instances are mapped into separated BFBs.

Figure 4.6: Reuse PLC code in IEC 61499 function blocks for LD, IL and ST.
Mapping Rule 5

An IEC 61131-3 SFC program \textit{Program} (4.3) is mapped to an ECC inside the IEC 61499 basic function block \textit{ECC} (4.12) if the original program is written in SFC:

\[ M_\delta: \text{Program} \rightarrow \text{ECC} \quad \text{and} \quad \text{Program} \subseteq \text{SFC} \]

If the original program is written in the sequential function chart (SFC) language, the program can be converted to the ECC inside a basic function block. Each SFC step is mapped to an EC state. SFC transitions, transition conditions, step algorithms are mapped to EC transition, EC transition conditions and EC state algorithm correspondingly as showing in Figure 4.7 and defined in the Section 3.2.1. However, parallel SFC steps are allowed in IEC 61131-3 but no concurrent ECC is defined in the IEC 61499 standard. The solution is to have an ECC with the product of all parallel SFC steps as proposed by Riedl et al. [92].

Figure 4.7: IEC 61131-3 SFC mapping to IEC 61499 ECC.
Mapping Rule 6

An IEC 61131-3 FBD program \( \text{Program} \) (4.3) is mapped to an IEC 61499 composite function block \( \text{CFB} \) (4.10) if the original program is written in FBD:

\[ M_{\eta} : \text{Program} \rightarrow \text{CFB} \text{ and } \text{Program} \subseteq \text{FBD} \]

If the original program is written in function block diagram (FBD) language, it can be converted to the function block network in the composite function block. Each function block in the FBD of IEC 61131-3 is converted to an IEC 61499 function block and connections between function blocks are easily established. All function blocks converted from the original PLC version must be in a single event chain in order to be executed once in each scan in sequential order as illustrated in Figure 4.8.

![Figure 4.8: IEC 61131-3 FBD mapping to IEC 61499 Composite FB.](image)

The other issue of the migration process is related to treating the global and program variables used in the PLC code. There are two levels of variables in the PLC program: controller (global) variables and program (local) variables. A controller variable can be accessed from anywhere in the PLC configuration. A program variable can only be accessed from a particular program. There is no global variable concept in the IEC 61499 standard as it is naturally designed for distributed systems control. The only place where a variable might be stored is a basic function block. The alternative choice is to build a service interface function block (SIFB) to access some external data sources. But the SIFB is implementation dependent which means a different SIFB must be created manually for every IEC 61499 platform. For the automatic migration process, using basic function blocks to store variables is more suitable.
In order to access global variables from any level in the IEC 61499 hierarchy, a pair of Publish and Subscribe function block is inserted. As indicated in Figure 4.9, a basic function block is used as the global variable storage. Each variable has a data input publishing its value and subscribing writing requests over the entire PLC configuration. The data storage function block is also suitable for program variables. To access a global variable from a BFB, a pair of PUBLISH and SUBSCRIBE SIFB need to be inserted between this BFB and the global data BFB.
4.5 Summary

This chapter has presented a formal model of the IEC 61131-3 standard for migration. This model not only describes the PLC software architecture but also includes the cyclical execution semantics. Two types of PLC cyclical execution semantics are modeled: the preemptive scan and non-preemptive scan. The formal model of the IEC 61499 standard is provided. To achieve the PLC cyclical execution in the IEC 61499 manner, a scheduling system implemented in function blocks is introduced. The scheduling system is based on the three-layer function blocks which are responsible for scheduling tasks, programs and subprogram/subroutines respectively. Finally, the mapping rules between the IEC 61131-3 model and the IEC 61499 model are provided. The mapping rules cover the entire program structure up to the algorithms inside the program. The object-oriented reuse redesign pattern is utilized in the mapping. The premises of using this redesign pattern is that the instructions and operands used in the program of the original PLC program must be supported in the target function block environment. The PLC variables with different scopes are also considered in the mapping rules. The solution of the PLC global variables issue in function blocks is also suggested. The mapping rules will be used as a guideline in the implementation of the automatic migration engine. By using the automatic migration engine, lots of redesign time could be saved.
5 Ontological Knowledge Base

5.1 Introduction

It has been demonstrated in the previous sections that IEC 61499 can implement modern software engineering design methods, for example object-oriented design patterns, much better than PLC tools and languages. This improves the design efficiency substantially, but on the other hand, the result FB network is quite complex. Checking their syntactic and semantic correctness becomes a challenge. This can be partially overcome by applying some “correct by design” methodologies, for instance, using the physical structure of the plant as a starting point for automatic code generation. This requires some “semantic glue” between the two worlds: physical object and its automation software.

It is also important to automate the re-use of existing IEC 61131-3 PLC code in the newly developed IEC 61499 function block solutions. The mapping rules proposed in the previous section also maximize the reusability of the original IEC 61131-3 PLC code. However all PLC code formats are quite different. In order to avoid lots of human efforts for transform all different types of PLC codes, a generic foundation for the migration process is required. A formal semantic link between the concepts of PLC programming and function block programming can simplify the migration task.

To provide such a multi-purpose semantic glue, a knowledge model for representing the related concepts of the function blocks architecture can provide great help. The ontology mechanism is considered as a proper representation of such a knowledge base. The word “Ontology” is from philosophical study of the nature of being, existence or reality originally and now is widely deployed into the semantic web programming and service-oriented architecture (SOA). Recently this concept is
introduced into the automation and control world [79]. Ontologies are majorly used to describing manufacturing and processing plants. For example, there are some researches on providing a standard ontology for describing manufacturing process, for instance, MASON [93]. This Chapter aims to provide a general ontology model for both the IEC 61131-3 and the IEC 61499 standard.

An international standard of ontology language for semantic web – OWL [94] was published by the W3C Consortium. OWL extends the RDF language (Resource Description Framework) which is considered as the fundamental data model of semantic web programming. Similar to RDF, OWL is also based on XML. There are three versions of OWL: OWL-Full, OWL-DL and OWL-Lite. The OWL-Lite is a subset of the OWL-DL and OWL-DL is a sublanguage of OWL-FULL as well.

Here, the OWL-DL (description logic) is used for describe the knowledge base [95]. The complexity of OWL-DL is sit between OWL-Full and OWL-Lite which is enough for handling the scenarios of semantic analysis and migration process. Reasoning can be applied to the ontology model of OWL-DL which is based on description logic (DL) [95]. This version of the OWL is sufficient for handling this transformation case and the additional ontology definition can be merged into any existing ontology model by using both the DL and the SWRL rules. The Semantic Web Rule Language (SWRL) [96] extends the set of OWL axioms to include Horn-like rules. It thus enables Horn-like rules to be combined with an OWL knowledge base.

The rest of the section is organized as following: Firstly in section 5.2, the concept of the layered ontology design is proposed. The methodology of automatic ontological knowledge base generation from XML definition is given in section 5.3. The IEC 61499 and IEC 61131-3 ontological knowledge base definition are provided in the section 5.4 and 5.5 respectively. One example from XML DTD definition (Rockwell XML) and another from XML XSD definition (PLCOpen XML) are illustrated. This section is concluded at the end. The ontology definitions could be found in the Appendix B.
5.2 Layered IEC 61499 Ontology Design

In the previous sections, some basic concepts of the ontology were introduced. Previously, the IEC 61499 ontology in [79] was developed manually leaving the questions of how adequate is it to the text of the standard and, especially, to particular implementations, which may slightly deviate from the standard or may extend its insufficiently defined parts. This shortage can be overcome by proposing a layered approach to structuring the ontology as shown in Figure 5.1.

![Diagram of Layered IEC 61499 Ontology Design](image)

Figure 5.1: Layered IEC 61499 Ontology Design.

The base level of the ontology is automatically generated from the IEC 61499 XML definition that captures most of syntactic properties and is used directly for syntactic analysis or code generation (Described further in the section 5.3). This approach promises to have less discrepancies between the code syntax supported by a tool and its semantic analyser. Moreover, the layered approach promises better extensibility of the ontology, or possibility to customize it for a particular dialect or design pattern.

The proposed IEC 61499 ontology includes three corresponding levels of its hierarchy. From nodes of those ontology levels, all items, defined in the original IEC 61499 XML DTDs, will be automatically transformed in a hierarchical structure (Described further in the section 5.3). At this stage, the generated ontology model is based only on syntactic rules with very limited semantic information that includes only the quantity of items that can exist in the system.

Also one should take into account possible contradictions in this syntax-based ontology model originated in the contradictions between the original DTDs and the text of the standard. For instance, a function block type can have maximum one basic FB or maximum one FB network and maximum one service simultaneously according to the original DTD file. In fact, only one of these three artefacts can exist in a single
FB type. Those contradictions must be clarified referring to the textual definitions in the IEC 61499 standard.

After the ontology model is capable of accurately representing the syntax, the next step is to add some semantic rules there. Three levels of rules are proposed to verify an IEC 61499 system as given in Figure 5.2.

![Figure 5.2: Flexible Semantic Analysis using layered ontology.](image)

(1) The first semantic layer consists of basic and simple rules to check that all the types (both function block types and data types) are matched correctly. For example, in implementations where typecasting is not supported, all data elements connected via arcs shall be identical. More generally, the implementation dependent typecasts can be represented as semantic rules. This ensures all events and data variables are correctly defined and properly used.

(2) The next layer of rules aims at achieving the correct execution semantics of the FB system or of a single FB. There are separate sets of rules for each execution semantic of IEC 61499 [91].

(3) The final category of the semantic rules is to check the compliance with particular design patterns or absence of known semantic problems.

The rules are defined in terms of the SWRL and can be verified by standard ontology reasoneres or using the Semantic Query-Enhanced Web Rule Language (SQWRL) [97] query engine.
5.3 Automatic Knowledge Base Generation from XML Definition

The first level of the layered ontology model refers to the actual PLC code structure, or in another word, the XML file. There are two types of XML definition: Document Type Definition (DTD) and XML Schema (XSL). Thuy et al. [98] proposed a solution to convert XML file schema to OWL automatically. A “DTD/XSD to OWL” transformation engine has been developed for the automatic generation of ontology from XML Schema using similar approach. Firstly, the mapping process for DTD to OWL is as follows:

(1) Each DTD document is considered as a domain in the ontology. DTD Elements can be grouped into sub-domains if required.

(2) Each DTD Element is mapped to an OWL class. In order to give a unique ID to each class, domain and sub-domain names must be added as prefixes for the class ID.

(3) The hierarchies of the DTD Elements are mapped to the object properties and if the Element has only constants, they are mapped to the data properties straight away by using the prefix “Has_ConstantValue_”.

(4) In a standard DTD document, there are some symbols to indicate occurrence of an element:

* Declaring Zero or More Occurrences of an Element;
+ Declaring Minimum One Occurrence of an Element;
? Declaring Zero or One Occurrences of an Element;

The OWL keyword owl:QualifiedCardinality is used to represent those occurrence symbols, for example:

*  \(Owl:\text{minQualifiedCardinality} = 0;\)
+  \(Owl:\text{minQualifiedCardinality} = 1;\)
?  \(Owl:\text{maxQualifiedCardinality} = 1;\)

(5) The attributes of an element are mapped to data properties.

There are two sorts of attributes: #REQUIRED and #IMPLIED. The “Required” attribute is mapped to \(Owl:\text{QualifiedCardinality}\) with quantity of exactly one. The “Implied” attribute means that the attribute is not necessary
Ontological Knowledge Base

to appear in the XML, which can be expressed by

\[ \text{Owl:}\text{maxQualifiedCardinality} = 1. \]

The mapping process for XSD to OWL is very similar to the DTD version. The entire process is given as:

1. Each XML schema (XSD) is considered as a domain in the ontology. XSD elements can be grouped into sub-domains if required.

2. Each XSD Element is mapped to an OWL class. In order to give a unique ID to each class, domain and sub-domain names must be added as prefixes for the class ID.

3. The hierarchies of the XSD Elements are mapped to the object properties and if the Element has an only simple type (e.g. \text{xs:string}), they are mapped to the data properties straight away by using the prefix “\text{Has_CONSTANTValue}”.

4. In a standard XSD document, there are some attributes of the XSD elements to indicate occurrence of an element:

   \[ \text{maxOccurs} = "\text{Integer}\" \]

   Declaring Maximum Occurrences of an Element;

   \[ \text{minOccurs} = "\text{Integer}\" \]

   Declaring Minimum Occurrence of an Element;

   The OWL keyword \text{owl:QualifiedCardinality} is used to represent those occurrence symbols, for example:

   \[ \text{maxOccurs} = "\text{Integer}\" \text{ is mapped to } \text{Owl:}\text{maxQualifiedCardinality} = \text{Integer}; \]

   And \[ \text{minOccurs} = "\text{Integer}\" \text{ is mapped to } \text{Owl:}\text{minQualifiedCardinality} = \text{Integer}; \]

5. The optional and required attributes of an element are mapped to data properties. Attributes are optional by default. If the use of attribute is set to required, the attribute is mapped to \text{Owl:}\text{QualifiedCardinality} with quantity of exactly one. The optional attribute means that the attribute is not necessary to appear in the XML, which can be expressed by \text{Owl:}\text{maxQualifiedCardinality} = 1.
5.4 IEC 61499 Ontological Knowledge Base Definition

The methodology of automatic conversion the definition of the XML in the previous section can be applied to the IEC 61499 knowledge base definition. The DTD files provided by the IEC 61499 standard are converted to the ontology T-Box. The T-Box is the knowledge base of all properties of the IEC 61499 standard and relationships between IEC 61499 concepts. According to the IEC 61499 standard, XML is used to define three classes of artefacts:

(1) Library Elements;

(2) Function Block Management Commands which define the protocol used for communication between the IEC 61499 devices;

(3) Data Types allowed in IEC 61499 artefacts.
The IEC 61499 XML format is specified in the form of 3 standard DTDs for Library Elements, Function Block Management Commands and Data types. Each DTD file is considered as a separate domain in the ontology and is converted and combined by the import engine into a single IEC 61499 ontological model. As described in the previous section, three major domains are considered as the root node of the ontology as shown in Figure 5.3.

Figure 5.3: IEC 61499 Ontology Model Overview.

All sub-domains are defined under the major domain:

*Library Elements* ≡ *Common Elements* ⊔ *FB Types* ⊔ *Adapter Types* ⊔ *Resource Types* ⊔ *System Elements* ⊔ *Sub-Application Types* ⊔ *Network Elements.*

And

*Function Block Management Commands* ≡ *Common Elements* ⊔ *FB Types* ⊔ *Data Types* ⊔ *Adapter Types* ⊔ *Requests* ⊔ *Responses*.

As no repeatable concept names is allowed in the ontology definition, the options are either to have all the IEC 61499 ontology concepts named with the domain and sub-domain name (e.g. `<DomainName>_<SubDomainName>_<ConceptName>`) or
store three root nodes in separate files without changing any concept name. Here, those root nodes are stored in separated files.

The object properties of this ontology model contain the hierarchy of the IEC 61499 code structure for semantic analysis. When using an object property to represent an element requiring another element, the name of this object property is defined as $\text{Has}_{-}\langle\text{ConceptName}\rangle$. To complete this object property, domains and ranges needs to be defined. The domains are the classes where this object property will be used from and ranges are the classes where this object property will be applied to. An object property can be used in multiple locations in the same ontology model.

Data properties are used to represent the attribute values of elements in IEC 61499. When using a data property to present a constant value of a data type in the attributes of elements, the data property is named as $\text{Has}_{-}\langle\text{ConceptName}\rangle_{-}\langle\text{AttributeName}\rangle$ or $\text{Has}_{-}\langle\text{ConstantValue}\rangle_{-}\langle\text{ConceptName}\rangle$ when the element itself is a constant value. Similar to the object property, domain and range are required as well. In the data property, domains are the locations where this data property will be used from and ranges are the pre-defined data types in the XML Schema and OWL.

The idea of the IEC 61499 properties ontology definition will be first illustrated on the concept of $\text{FBType}$ of IEC 61499. The keyword $\text{FBType}$ defines a function block type that can be basic, composite or service interface function block. The corresponding ontological definition comprises of BasicFB (Basic FB type) or FBNetwork (Composite FB) or Service (Service Interface FB) element associated with an interface. Beyond those essential parts, there might be some extra details including compiler information, version information, etc. For the data properties, a function block must have a name and may have some comments.

First step of creating a class is to create all data properties used in this class. For instance, $\text{FBType}$ must have a name of data type String (Characters) in description logic:

$$\exists \text{Has}_{-}\text{FBType}_{-}\text{Name}.\text{String}$$
And then convert into OWL format in Figure 5.4:

```xml
<owl:Restriction rdf:resource="#Has_FBType_Name"/>
   <owl:qualifiedCardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:qualifiedCardinality>
   <owl:onDataRange rdf:resource="#String"/>
</owl:Restriction>
```

Figure 5.4: IEC 61499 Data Property of FBType.

Next step is to create all object properties for this class. An FBType class either represents a Basic Function Block description or a Function Block Network or a Service Interface FB:

```plaintext
( =1 Has_FBNetwork.FBNetwork | =1 Has_BasicFB.BasicFB | =1 Has_Service.Service)
```
And then convert into OWL format in Figure 5.5.

```
<rdfs:subClassOf>
  <owl:Class>
    <owl:unionOf rdf:parseType="Collection">
      <owl:Restriction>
        <owl:onProperty rdf:resource="#Has_BasicFB"/>
        <owl:onClass rdf:resource="#BasicFB"/>
        <owl:maxQualifiedCardinality
          rdf:datatype="&xsd;nonNegativeInteger">1
        </owl:maxQualifiedCardinality>
      </owl:Restriction>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#Has_FBNetwork"/>
        <owl:onClass rdf:resource="#FBNetwork"/>
        <owl:maxQualifiedCardinality
          rdf:datatype="&xsd;nonNegativeInteger">1
        </owl:maxQualifiedCardinality>
      </owl:Restriction>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#Has_Service"/>
        <owl:onClass rdf:resource="#Service"/>
        <owl:maxQualifiedCardinality
          rdf:datatype="&xsd;nonNegativeInteger">1
        </owl:maxQualifiedCardinality>
      </owl:Restriction>
    </owl:unionOf>
  </owl:Class>
</rdfs:subClassOf>
```

Figure 5.5: IEC 61499 Object Property of FBType.
The Figure 5.6 is a simplified graphical version of the \textit{FBType} concept. One sees from Figure 5.6 that, a \textit{FBType} individual must have exactly one interface, name, maximum one service, identification, version information, compiler information and comments, and maximum one Function Block Network or Basic Function Block or Service description.

Figure 5.6: \textit{FBType} Concept Ontology Diagram.

Next, the system configuration concept \textit{System} is shown as Figure 5.7. A \textit{System} individual must have minimum one \textit{Device} and \textit{VersionInfo} object configured. All other object properties including \textit{Application}, \textit{Link}, \textit{Mapping}, \textit{Segment}, \textit{CompilerInfo} and \textit{Identification} are optional from the FB XML definition. For the data properties, a \textit{System} node must have exactly one \textit{Name} and can have maximum one \textit{Comment}.

Figure 5.7: \textit{System} Concept Ontology Diagram.
As a compulsory node in the System concept, a Device can have maximum one FBNetwork and optional number of Parameter and Resource configured. A Device also has several data properties including Name, Type, Comment, X and Y coordinates in the editor. A FBNetwork Node consists of maximum one occurrence of collection of AdapterConnection, DataConnection and EventConnection. It also can have zero or more FB instances scheduled in the function block network. A FBNetwork concept can also be used in a Resource, FBType, DeviceType or ResourceType node.

Figure 5.8: Device Concept Ontology Diagram.

In a FBType, BasicFB is representing a basic function block type. A BasicFB is comprised by one Execution Control Chart (ECC), some Algorithm and a collection of InternalVar as illustrated in Figure 5.9. An Algorithm is written in one of the IEC 61131-3 programming languages (LD, ST, FBD) or others. An ECC consists of at least one ECState and one ECTransition. The source and the destination of ECState in the ECTransition are declared as data properties as well as the EC Transition Condition.

Figure 5.9: BasicFB Concept Ontology Diagram.
The DTD to OWL engine developed according to the rules above is able to generate the complete IEC 61499 ontology T-Box including all other nodes.

Beside the T-Box, all actually implemented IEC 61499 system configurations and function blocks instances are modelled in A-Box. The next step is to automatically import all function block files (*.fbt), resources files (*.res), device files (*.dev) and system configuration files (*.sys) into the A-Box.

An OWL individual is created for each XML element in the IEC 61499 source file. The object property `Has_<ConceptName>` is created for all child nodes of this XML element. Finally, data properties in the form `Has_<ConceptName>_AttributeName` are created for all attributes of this XML element with the actual value stored in them. Figure 5.10 presents an interface and ECC of the basic function block “EStop” which controls the emergency stops. The ECC of the FB works as follows: at the input event REQ indicating emergency stop status change will be manipulated and then the ECC returns to the idle state.

```xml
<owl:NamedIndividual rdf:about="#FBType_EStop">
  <rdf:type rdf:resource="#FBType" />
  <Has_Identification rdf:resource="#FBType_EStop_Identification" />
  <Has_VersionInfo rdf:resource="#FBType_EStop_VersionInfo" />
  <Has_CompilerInfo rdf:resource="#FBType_EStop_CompilerInfo" />
  <Has_InterfaceList rdf:resource="#FBType_EStop_InterfaceList" />
  <Has_BasicFB rdf:resource="#FBType_EStop_BasicFB" />
  <Has_FBType_Name rdf:datatype="STRING">EStop</Has_FBType_Name>
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="#FBType_Estop_InterfaceList">
  <rdf:type rdf:resource="#InterfaceList" />
  <Has_EventInputs rdf:resource="#FBType_EStop_EventInputs" />
  <Has_EventOutputs rdf:resource="#FBType_EStop_EventOutputs" />
</owl:NamedIndividual>
```
<Has_InputVars rdf:resource="FBType_EStop_InputVars" />
<Has_OutputVars rdf:resource="#FBType_EStop_OutputVars" />
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="#FBType_EStop_EventInputs">
  <rdf:type rdf:resource="#EventInputs" />
  <Has_Event rdf:resource="#FBType_EStop_Event_INIT" />
  <Has_Event rdf:resource="#FBType_EStop_Event_REQ" />
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="#FBType_EStop_EventOutputs">
  <rdf:type rdf:resource="#EventOutputs" />
  <Has_Event rdf:resource="#FBType_EStop_Event_INITO" />
  <Has_Event rdf:resource="#FBType_EStop_Event_CNF" />
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="#FBType_EStop_InputVars">
  <rdf:type rdf:resource="#InputVars" />
  <Has_VarDeclaration rdf:resource="#FBType_EStop_Var_QI" />
  <Has_VarDeclaration rdf:resource="#FBType_EStop_Var_ES_OK" />
  <Has_VarDeclaration rdf:resource="#FBType_EStop_Var_RESET" />
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="#FBType_EStop_OutputVars">
  <rdf:type rdf:resource="#OutputVars" />
  <Has_VarDeclaration rdf:resource="#FBType_EStop_Var_QO" />
  <Has_VarDeclaration rdf:resource="#FBType_EStop_Var_ESFAULT" />
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="#FBType_EStop_BasicFB">
  <rdf:type rdf:resource="#BasicFB" />
  <Has_Algorithm rdf:resource="#FBType_EStop_Alg_INIT" />
  <Has_Algorithm rdf:resource="#FBType_EStop_Alg_REQ" />
  <Has_ECC rdf:resource="#FBType_EStop_ECC" />
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="#FBType_EStop_ECC">
  <rdf:type rdf:resource="#ECC" />
</owl:NamedIndividual>
To describe this function block in the ontological knowledge base, an `owl:NamedIndividual` item is created in the form of `<NodeType>_<NodeName>` as `FBType_EStop`. To specify the node type of this instance, a `rdf:type` is inserted as the first sub-node with a type of `FBType`. All sub-elements, such as: `Identification`, `VersionInfo`, `CompilerInfo`, `InterfaceList` and `Basic FB` of `FBType` (refer to Figure 5.10) are used to create object properties. Property `Has_<ConceptName>` that are linking to the owl individual of that particular sub-node. The attributes of this node are created as data properties. In the case here, the data property `Has_FBType_Name` is created and refers to data type `String`. It indicates the name of this function block instance is `EStop`. The function block interface `FBType_EStop_InterfaceList` is connected to the `FBType_EStop class by using the object property `Has_InterfaceList`. All event inputs (`INIT and REQ`) and outputs (`INITO and CNF`) as well as all data inputs (`QI, ES_OK and RESET`) and outputs (`QO and ES_FAULT`) are linked to the interface list. In the basic function block definition, all EC states, algorithms and transitions are attached. Similar approach is applied to all remaining nodes and properties to import the complete code into the knowledge base A-Box.
5.5 IEC 61131-3 Ontological Knowledge Base Definition

The approach used for the IEC 61131-3 knowledge base generation is identical to the IEC 61499 knowledge base. However, there is no standard XML definition defined in the IEC 61131-3 standard. Although not all vendors provide the XML definition of the PLC code, there are still various XML representations available on the market. Here, two commonly used platforms – Rockwell Automation [99] (in DTD) and PLCOpen [100] (in XSD) are given as examples.

5.5.1 Rockwell Automation XML Version from DTD

Rockwell Automation is one of the major PLC vendors in the world. Same as the IEC 61499 version, each XML element in the DTD definitions of the Rockwell Automation is mapped to an ontology class. The top level of the PLC programming organization unit Task is chosen as the example here. As given in Figure 5.11, an object property Has_ScheduledPrograms of a class named ScheduledPrograms is defined for representing a PLC task that has one or more programs associated. The OWL node minQualifiedCardinality is a quantity limitation to indicate the minimum occurrence of a program class is a task class.

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#Has_ScheduledPrograms"/>
  <owl:onClass rdf:resource="#Task"/>
  <owl:minQualifiedCardinality
    rdf:datatype="&xsd;nonNegativeInteger">1
  </owl:minQualifiedCardinality>
</owl:Restriction>
```

Figure 5.11: IEC 61131-3 Rockwell Object Property Example of Task.
Each PLC task has a unique name which is represented as a data property *Has_Task_Name* with of data type *String* attached to the task class. This data property in the OWL format is given in Figure 5.12.

```xml
<rdf:subClassOf>
  <owl:Restriction>
    <owl:onProperty rdf:resource="#Has_Task_Name"/>
    <owl:qualifiedCardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:qualifiedCardinality>
    <owl:onDataRange rdf:resource="#String"/>
  </owl:Restriction>
</rdf:subClassOf>
```

Figure 5.12: IEC 61131-3 Rockwell Data Property Example of *Task Name*.

The entire ontology definition for the *Task* node is illustrated in Figure 5.13.

![Task Concept Ontology Diagram – Rockwell Version.](image)

There is one collection of *ScheduledProgram* associated with the *Task*. There are several data properties also attached to the *Task* node for indicating name, type (periodic or continuous), priority, watchdog timeout value and other settings of the task. The same procedure is applied to all nodes in the original DTD file and all corresponding objects and properties are created automatically.
After the T-Box is settled, the actual PLC code needs to be imported as OWL individuals into the knowledge base A-Box. An example of a task named FastTask which has a scheduled program named MainProgram is provided in Figure 5.14. A rdf:type node is inserted to indicate the type of this instance. A program named MainProgram is attached with the object property “Has_ScheduledProgram”. The data property “Has_Task_Name” stores the name of this task instance.

```
<owl:NamedIndividual rdf:about="#Task_FastTask">
  <rdf:type rdf:resource="#Task" />
  <Has_ScheduledPrograms rdf:resource="#Task_FastTask_Programs" />
  <Has_Task_Name rdf:datatype="STRING">FastTask</Has_Task_Name>
  <Has_Task_Type rdf:datatype="STRING">Periodic</Has_Task_Type>
  <Has_Task_Priority rdf:datatype="STRING">10</Has_Task_Priority>
  <Has_Task_Watchdog rdf:datatype="STRING">500</Has_Task_Watchdog>
  <Has_Task_DisableUpdateOutputs rdf:datatype="STRING">0</Has_Task_DisableUpdateOutputs>
  <Has_Task_InhibitTask rdf:datatype="STRING">0</Has_Task_InhibitTask>
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="#Task_FastTask_Programs_MainProgram">
  <rdf:type rdf:resource="#ScheduledPrograms" />
  <Has_ScheduledProgram rdf:resource="#Task_FastTask_MainProgram" />
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="#Task_FastTask_MainProgram">
  <rdf:type rdf:resource="#ScheduledProgram" />
  <Has_Program_Name rdf:datatype="STRING">MainProgram</Has_Program_Name>
</owl:NamedIndividual>
```

Figure 5.14: Example of OWL Individual for the PLC Task which has a program.

By repeating this process, all XML based IEC 61131-3 Rockwell Automation PLC codes are imported into the ontological knowledge base.
5.5.2 PLCOpen XML Version from XSD

PLCOpen is an organization active in industrial control especially for PLC control. The PLCOpen XML file format is defined in the XML Schema (XSD). In this case, each XSD element in the PLCOpen XML format is mapped to an ontology class. The same node Task as the Rockwell version is selected as the example. As shown in Figure 5.15, an object property Has_pouInstance of a class named pouInstance is defined for representing a PLC task that has zero or more programming organization unit instances linked to a task. The quantity limitation of the pouInstance inside a task class is zero to infinite.

```xml
<rdfs:subClassOf>
  <owl:Class>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#Has_pouInstance"/>
      <owl:onClass rdf:resource="#Task"/>
      <owl:minQualifiedCardinality
        rdf:datatype="&xsd;nonNegativeInteger">0
        </owl:minQualifiedCardinality>
      </owl:Restriction>
    </owl:Class>
  </owl:Restriction>
</rdfs:subClassOf>
```

Figure 5.15: IEC 61131-3 PLCOpen Object Property Example of Task.
Each PLC task has a unique name which is represented as a data property. The completed ontology definition for the Task node of the PLCOpen version is provided in Figure 5.16. There is a set of pouInstnace, addData and documentation can be attached with the Task. There are also some compulsory data properties of the Task node for indicating name and priority. Other properties like single, interval and globalID are optional to the task. The rest of the nodes in the PLCOpen XSD file are converted into the OWL format as well following the same procedure.

![Task Concept Ontology Diagram – PLCOpen Version.](image)

Figure 5.16: Task Concept Ontology Diagram – PLCOpen Version.
The final step is to import PLCOpen XML codes into the A-Box. The same example of a task named *FastTask* which has a POU named *MainProgram* is provided in Figure 5.17. A *rdf:type* node is inserted to indicate the type of this instance. A program named *MainProgram* is attached with the object property “*Has_pouInstance***. The data property “*Has_Task_Name*** stores the name of this task instance and the priority of this task is linked by the data property “*Has_Task_Priority***.

```xml
<owl:NamedIndividual rdf:about="#Task_FastTask">
  <rdf:type rdf:resource="#Task" />
  <Has_pouInstance rdf:resource="#Task_FastTask_MainProgram" />
  <Has_Task_Name rdf:datatype="STRING">FastTask</Has_Task_Name>
  <Has_Task_Priority rdf:datatype="STRING">10</Has_Task_Priority>
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="#Task_FastTask_MainProgram">
  <rdf:type rdf:resource="#pouInstance" />
  <Has_pouInstance_Name rdf:resource="#STRING">MainProgram</Has_pouInstance_Name>
  <Has_pouInstance_typeName rdf:resource="#STRING">MainProgramScheduler</Has_pouInstance_typeName>
</owl:NamedIndividual>
```

Figure 5.17: Example of OWL Individual for the PLC Task which has a POU.

Using the same approach, the remaining PLC codes of the PLCOpen XML are imported into the ontological knowledge base automatically.
5.6 Summary

In this section, an ontology model of three layers is proposed. The details of the first syntactic layer in this ontology model are defined. The methodology of automatic creating ontology definition and importing ontological individuals is provided. The ontological knowledge base of the IEC 61499 standard and two implementations of IEC 61131-3 standard are generated from the standard DTD or XSD files. All instances are directly imported from the XML files and stored into the corresponding knowledge bases. This automates the process of knowledge base creating and simplifies implementation of the method. All human mistakes and typo errors during the conversion can be eliminated. This approach is independent from any specific editor tool and is compatible with all the tools as well as long as the XML Schema is provided.

The results described in this Chapter enable the use of the ontological models in the transformation engine of the migration process and the semantic analysis. These will be discussed in the following Chapters.
6 Ontology Based Implementation of Transformation Mechanism

6.1 Introduction

Before get into the details of the implementation, a clear picture of the migration process needs to be described. The proposed migration approach aims at automatic recreation of IEC 61131-3 PLC code behaviour in IEC 61499. The entire process includes importing of PLC programs into a knowledge base, automatic mapping from IEC 61131-3 knowledge base elements to IEC 61499 version, recreation of original execution models and code generation to IEC 61499 platforms. The sequence of steps for the automatic migration tool is illustrated in Figure 6.1.

![Figure 6.1: Migration Process from IEC 61131-3 PLCs to IEC 61499 FBs.](image)

On the left hand side of Figure 6.1, the source code from single PLC written in IEC 61131-3 languages are imported into the IEC 61131-3 knowledge base as ontology individuals. The IEC 61131-3 knowledge base is defined in the OWL. Recall from the previous Chapter, an ontological knowledge base consists of two parts: T-Box and A-
Box. In the ontological terms, definitions are belonging to the taxonomy box (T-Box) which contains definitions of the ontological concepts and properties. When this is applied to the IEC 61131-3 standard, a T-Box contains definitions of all IEC 61131-3 program elements in the XML format and program hierarchy between those IEC 61131-3 concepts. Those concepts are linked together by using object properties and described in the description logic which ontology is mainly based on. Actual data of concepts are stored as data properties. On the other hand, instances are stored into the assertion box (A-Box) which consists of knowledge that is specific to the individual system design. In the term of the IEC 61131-3 knowledge base, an A-Box retains all instances imported from PLC configurations. That information is saved as ontology individuals in the A-Box.

The source code of PLCs presented in the XML format is interpreted during the import process and stored into the IEC 61131-3 knowledge base as OWL individual which relies on XML as well. The import process consists of two parts. The first part is to recover PLC execution order including scheduled parameters of tasks and programs. This step also gathers PLC tasks execution order, priority and other useful information which are required later in the migration process. The second part is to copy the original algorithms encapsulated in function blocks, functions and subroutines written in IEC 61131-3 languages. Those algorithms are considered as reusable software components and will be utilized again according to the object-oriented reuse version redesign approach proposed previously.

After the knowledge base for the original PLC codes has been created, the next action is to transform all IEC 61131-3 elements into the IEC 61499 format according to the general migration rules defined in Chapter 5. The execution model of IEC 61131-3 is recreated by inserting scheduling function blocks into the IEC 61499 result system. Tasks and programs from PLC codes are rearranged according to the original order in the PLC with new introduced scheduling function blocks. The transformation engine is based on the Semantic Query-enhanced Web Rule Language (SQWRL) which is designed to extract information from OWL knowledge bases. Similar to SPARQL which queries RDF knowledge base, SQWRL is the query language for OWL. Mapping rules defined as complete SQWRL queries are taken into the transformation engine. The results of those queries provide the information required to construct desired IEC 61499 systems.
The last step in the chain is the code generation for IEC 61499. Based on the target platform, the respected code templates and related instances are combined by the code generation engine and output as IEC 61499 XML files. The generated code can be viewed and edited in any IEC 61499 IDE and immediately deployed using its mechanisms of compilation and deployment. The example platform is this work is FBDK [101] although there are many IEC 61499 platforms such as nxtStudio [102], 4DIAC-IDE [103], FBench [104] and CORFU [105].
6.2 Transformation Implementation using Ontological Models

6.2.1 Knowledge Base for Migration Mapping Rules

The scheduler function blocks introduced in the Section 4.3 provides a customizable template for scheduling function blocks in a sequential order like PLC. Although the function block pattern is generic, customization based on the number and the type of tasks as well as the number and the execution order of programs of each project is still required. It is essential to auto generate scheduling functions as well as all mapping other function blocks for both usability and generality purposes.

In order to create the scheduling function blocks and map PLC tasks and programs automatically, a set of generic migration mapping rules is purposed in Section 4.4. These generic migration mapping rules can be implemented using the ontology mapping of classes and properties between the IEC 61499 and IEC 61131-3 knowledge bases. The structure of knowledge base contains mapping rules is illustrated in Figure 6.2.

![Diagram](#)

Figure 6.2: Knowledge Base Definition for Migration Mapping Rules.
As defined in the knowledge base, each migration rule is stored as a step. Each step may have one or more actions. The step ontology individual template is given in Figure 6.3. All step individuals with name Step_Name have a type of Step and associated with one or more actions identified by Action_Name.

![Step Ontology Individual Template](image)

An action is responsible for mapping from one OWL class/object property/data property in the original knowledge base to another OWL class/object property/data property in the target knowledge base. An action consists of a source, a destination and a script. The template for the action is given in Figure 6.4.

![Action Ontology Individual Template](image)

The source refers to the original node in the IEC 61131-3 knowledge base. The name Source_Name of the original node and the mapping type (Class/Object Property/Data Property) are associated with the source.

![Source Ontology Individual Template](image)
Ontology Based Implementation of Transformation Mechanism

Similar to the source, the destination is linked to the target node in the IEC 61499 knowledge base with its node name and type (Class/Object Property/Data Property). In an action, there can be one or more scripts. The script is associated with a template and a query as shown in Figure 6.6.

```xml
<owl:NamedIndividual rdf:about="&xsd;Script_Name">
  <rdf:type rdf:resource="#Script" />
  <Has_Template rdf:resource="#Template_Name" />
  <Has_Query rdf:resource="#Query_Name" />
</owl:NamedIndividual>
```

Figure 6.6: Script Ontology Individual Template.

The template in Figure 6.7 keeps the content of the actual value to be written into the target node. Content is a context template which consists of a combination of string contents and SQWRL query target variables. The migration engine will pick up the content in the template then replace variable symbols with the SQWRL query results for those variables. For example, the content is defined as “System_?a”. ?a is a variable in the SQWRL query and VAR node for selecting the system name. The system name will be replaced in the content and finally result is shown as “System_Name”.

```xml
<owl:NamedIndividual rdf:about="&xsd;Template_Name">
  <rdf:type rdf:resource="#Template" />
  <Has_Template_Content rdf:datatype="#CDATA" >Content</Has_Template_Content>
</owl:NamedIndividual>
```

Figure 6.7: Template Ontology Individual Template.

Some values from the node in the original knowledge base may be required as a part of the value in the target knowledge base. A query engine for fetching relevant information from the ontological knowledge base would be useful. The SQWRL provides the ability to extract information from the ontological knowledge base. SQWRL is based on the Semantic Web Rule Language (SWRL) and extended with string processing, aggregation, counting and many other features. The SQWRL query (more precisely, its left-hand side) is stored in the SQWRL node and the result variable name(s) (that is after keyword "select" in SQWRL query) is saved in the VAR node. If there is more than one matched value in the result set, the variable will be replaced by the combined multiple results as one single text block. When the migration process starts, the migration engine fetches all steps with actions from a
procedure and execute through them. In each action, an OWL individual is created if the destination type is an OWL class. This individual name is provided by the content of the template which is filled by the SQWRL query from the Query node as shown in Figure 6.8. If the destination type is a relation, an object property or a data property is created and attached to that OWL class. The referred OWL individual instance from the object property or the value from the data property is selected by the query in the SQWRL node.

```xml
<owl:NamedIndividual rdf:about="&xsd;Query_Name">
  <rdf:type rdf:resource="#Query" />
  <Has_Query_SQWRL rdf:datatype="#CDATA">SQWRL_QUERY</Has_Query_SQWRL>
  <Has_Query_VAR rdf:datatype="#CDATA">Variable_List</Has_Query_VAR>
</owl:NamedIndividual>
```

Figure 6.8: Query Ontology Individual Template.

### 6.2.2 Migration Process with Transformation Engine

After the knowledge base for storing mapping rules are designed, the next step is to convert the instances in the PLC knowledge base into the function block knowledge base automatically. The transformation engine is built in Microsoft Visual C# and running on Microsoft .Net framework. Besides the transformation wizard, the transformation engine also has a built-in OWL API, a SWRL built-in functions library and a SQWRL query engine. The OWL API provides features of generate knowledge base definition from XML schema or DTD file, import from XML into the knowledge base in OWL and export from OWL back to the XML. All SWRL built-in functions defined in the W3C standard [96] are implemented in the SWRL library. Also a SQWRL query engine is also created for fetching individual data from the knowledge base. The SQWRL built-in functions are also included in the transformation engine.

The first step of the transformation wizard is to import PLC programs into the IEC 61131-3 ontological knowledge base. The knowledge base definitions are created automatically from the DTD definition of IEC 61131-3 PLCs. Although each PLC vendor has their own file format definition, as long as the file format is based on a proper text format like XML, those files can be imported automatically into the knowledge base.

Each XML element in the DTD definitions is mapped to an ontology class. There are two types of properties for each class as introduced previously: Object Property
and Data Property. Object properties refer to relationships between classes and data properties refer to actual data values associated with a class. To indicate a class is associated with another class, an object property named as \(<\text{Has\_TargetClassName}\>\) is inserted. To set a value to a class, a data property is named as \(<\text{Has\_OriginalClassName\_AttributeName}\>\). For example given in Figure 6.9, an object property \(\text{Has\_Program}\) of a class named \(\text{Program}\) is defined for representing a PLC task that has one or more programs associated. The OWL node \(\text{minQualifiedCardinality}\) is a quantity limitation to indicate the minimum occurrence of a \(\text{program}\) class is a \(\text{task}\) class.

![Image](image1.png)

Figure 6.9: IEC 61131-3 Object Property Example of Task.

Each PLC task has a unique name which is represented as a data property \(\text{Has\_Task\_Name}\) with of data type \(\text{String}\) attached to the task class. This data property in the OWL format is given in Figure 6.10.

![Image](image2.png)

Figure 6.10: IEC 61131-3 Data Property Example of Task Name.

All elements from the DTD definition of the original PLC code are stored into the ontological knowledge base T-Box. The next step is that the actual PLC code is imported as OWL individuals into the knowledge base A-Box. An example of a task named \(\text{FastTask}\) which has a program named \(\text{MainProgram}\) is provided in Figure
6.11. A rdf:type node is inserted to indicate the type of this instance. A program named *MainProgram* is attached with the object property “*Has_Program*”. The data property “*Has_Task_Name*” stores the name of this task instance.

```xml
<owl:NamedIndividual rdf:about="#Task_FastTask">
  <rdf:type rdf:resource="#Task" />
  <Has_Program rdf:resource="#Task_FastTask_Program_MainProgram" />
  <Has_Task_Name rdf:datatype="STRING">FastTask</Has_Task_Name>
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="#Task_FastTask_Program_MainProgram">
  <rdf:type rdf:resource="#Program" />
  <Has_Program_Name rdf:datatype="STRING">MainProgram</Has_Program_Name>
</owl:NamedIndividual>
```

Figure 6.11: Example of OWL Individual for the PLC Task which has a program.

By repeating this process, all XML based IEC 61131-3 PLC codes are imported into the ontological knowledge base and ready for transformation.

The next part of the transformation wizard is to perform the ontology mapping between the individuals of those two knowledge bases. The engine processes rules as:

1: for all Step do
2:   for all Action do
3:     for all Script do
4:       Execute SQWRL Query from the original IEC 61131-3 KB and Select variable Results from VAR
5:       Replace Variable in the Template with SQWRL Query Results
6:       Create an Instance in the target IEC 61499 KB with Template Content
7:  end for
8: end for
9: end for

Each step is responsible for processing one ontology node. And actions are handling the mapping between class, object properties and data properties of this node. The engine loops through steps and actions in each step. For each action, the engine runs the SQWRL query in the *Script* node which using *SQWRL* node as left hand side and *VAR* node as right hand side. Variables in the *Template* node are replaced with SQWRL query results. A new OWL individual is created in the IEC 61499 knowledge base which has a type and name defined in the *Destination* node.
with the content in the Template node. This new individual could be a new OWL class individual node in the target knowledge base, an object property instance linked to another node or a data property associated with some values. This new individual naming conversion is defined as:

\[ \text{RootNodeType}_\text{SubNodeType}...\text{IndividualName} \]

For example, if a program instance is created by the engine, it will be named as \( \text{Project\_Task\_Program}_{<\text{ProgramName}>} \).

Unfortunately, the SWRL or SQWRL is not capable of inserting or deleting an individual to/from the knowledge base. This is done via the code written in the C#. Although the data required for the target knowledge base is gathered by the SQWRL query, the new individual has to be written into the IEC 61499 knowledge base by the transformation engine. The details of the mapping rules’ implementations are provided in Section 6.2.3.

The final step of the transformation wizard is to generate the code for the target IEC 61499 FB system. This is the inverse action of the import step. The objects stored in the knowledge base are extracted into the IEC 61499 XML format. The generation process starts from the root node defined in the IEC 61499 XML format - system configuration. All individuals of data properties and object properties associated with this system configuration node in the ontological knowledge base are listed by the code generation engine. An XML root node \( <\text{System}> \) is created and all data properties are mapped to the XML attributes of the root node. The name of the data property becomes the name for the XML attribute and the value of the data property is copied into the value of the XML attribute. The next step is to loop through all object properties and create sub XML nodes named as names of the object properties attached to the root node. This procedure is repeated for every sub node until no further object property is found in the knowledge base. Also if a function block type, resource type or device type is discovered during the code generation process, the same generation procedure will apply to those function blocks, resources and devices. Instead of using \( <\text{System}> \) as the root node, IEC 61499 keywords \( <\text{FBType}> \), \( <\text{ResourceType}> \) and \( <\text{DeviceType}> \) are used respectively.
6.2.3 Migration Rules Implementation

After the entire transformation process is described, the implementation of the mapping rules needs to be specified. All IEC 61131-3 platforms which support exporting PLC files as XML format can be migrated. In this particular example, the Rockwell ControlLogix PLC is selected as the source and the FBDK is used as the destination function block editor. Each mapping rules consists of several steps defined in the ontology. Inside each step, there are numbers of actions required to achieve the mapping rules.

Following the first mapping rule, a system configuration is generated for each PLC project. The PLC name is mapped into the system configuration name. The complete rule is given in Figure 6.12. An action is created with mapping from RSLogix5000Content node in the PLC to the System node in the FB. The script has a content of $b$ which refers to a SQWRL variable in the template for selecting the project name.

The main operator of the SQWRL is sqwrl:select. It will fill a table which uses arguments as column names from the target knowledge base. The data property TargetName of the class RSLogix5000Content is selected as the name of the mapped system configuration. The swrlb:stringConcat in the query is one of the built-in SWRL functions for construct string values. The results of the system name from the SQWRL query are stored in the variable $b$. The actual value replaces the $b$ in the data template of the content and this information is used by creating an ontology individual of the given class and its naming.
Ontology Based Implementation of Transformation Mechanism

Figure 6.12: Step Example of Mapping PLC project to System Configuration.
Next, a step is created for mapping from all IEC 61131-3 resources to IEC 61499 devices. The source of this action is the PLC resource instance (refer to the Controller class in the Rockwell PLC ontology definition) and the destination is the device type instance named RMT_DEV (Remote Device) in the IEC 61499 ontology definition. The SQWRL query for selecting controller name as the new device name is:

\[
\text{RSLogix5000Content(?a)} \wedge \text{Has_RSLogix5000Content_Controller(?a, ?b)} \wedge \text{Has_Controller_Name(?b, ?name)} \\
\rightarrow \text{sqwrl:select(?name)}
\]

Besides direct ontology node mapping, a main PLC scheduling function block is also required for each mapped IEC 61499 device. A new device named scheduling is inserted into the system configuration as a migration action. The first action of this step is to recognize all periodic tasks and schedule those tasks properly into the main PLC scheduling function block. To construct the main PLC scheduling function block, the first SQWRL node of the Query in this Action is to find all periodic tasks and place them into the main scheduling function block:

\[
\text{Task(?Task)} \wedge \text{Has_Task_Name(?Task, ?Name)} \wedge \text{Has_Task_Rate(?Task, ?Rate)} \wedge \text{Has_Task_Priority(?Task, ?Priority)} \wedge \text{swrlb:stringEqualIgnoreCase(?Type, “PERIODIC”) } \\
\text{sqwrl:makeSet(?Task, ?PeriodicTasksSet)} \wedge \text{sqwrl:groupBy(?PeriodicTasksSet, ?Name, ?Priority, ?Rate)} \\
\rightarrow \text{sqwrl:select(?Name, ?Priority, ?Rate)}
\]

The above query retrieves names, types and priorities of all tasks in a PLC configuration. Has_Task_Name, Has_Task_Type, Has_Task_Rate and Has_Task_Priority are the data properties of the Task object. The swrlb:stringEqualIgnoreCase is the SWRL built-in function for comparing string values. The sqwrl:makeSet is also a built-in collection operator to construct and manipulate sets. The main purpose of this operator is to support the closure operations for queries with negation or complex aggregation functionalities. The last operator sqwrl:groupBy is also a SQWRL built-in function which is used to group sets of entities together. By using this query, all names, priorities, scan rate of tasks with type of periodic are listed in the resulting table.

After the context for this action is ready, the query results need to be filled into the step data template. A pair of event input and output as well as an enabling Boolean
variable in the template of the main scheduling function block interface is inserted using the name of the task. Also an EC state named <\texttt{taskname}>_\texttt{DONE} is created for acknowledging the process complete signal from task schedulers. The \textit{REQ} state is always inserted into the main scheduling function block. Emitting tasks trigger events are created in the \textit{REQ} state according to the orders of priorities. The \textit{REQ} algorithm consists of three parts: time stamp calculation (?\texttt{TimeStampPart}), periodic task (?\texttt{PTaskPart}) and continuously task (?\texttt{CTaskPart}).

The content of the data template is given as:

\begin{verbatim}
Timestamp := Timestamp + 1;
IF Timestamp >= SCAN_TIME THEN
    Timestamp := Timestamp - SCAN_TIME;
?TimeStampPart
END_IF;
?PTaskPart
?CTaskPart
\end{verbatim}

The script also consists of three SQWRL queries. The first SQWRL script is given as:

\begin{verbatim}
Task(?Task)
^ Has_Task_Name(?Task, ?TaskName)
^ Has_Task_Type(?Task, ?Type)
^ swrlb:stringEqualIgnoreCase(?Type, "PERIODIC")
^ swrlb:stringConcat(?TimeStampPart, ?TaskName, ":= FALSE;")

All names of periodic tasks are attached to ":= FALSE;” replace the ?TimeStampPart in the content of the data template as a single block. (For example: two periodic tasks P1 and P2 constructed a string “P1\_ENABLE := FALSE; P2\_ENABLE := FALSE;” replace the ?TimeStampPart)

The second script is to collect periodic tasks with their interval rate:

\begin{verbatim}
Task(?Task)
^ Has_Task_Name(?Task, ?TaskName)
^ Has_Task_Type(?Task, ?Type)
^ Has_Task_Rate(?Task, ?Rate)
^ swrlb:stringEqualIgnoreCase(?Type, "PERIODIC")
^ swrlb:stringConcat(?NotTaskName, ": NOT(\", ?TaskName, ": _ENABLE) & NOT\(",
    ?TaskName, " _ENABLE_ONS\")"
^ swrlb:stringConcat(?PTaskPart, "IF Timestamp >= ", ?Rate, ?, NotTaskName, " THEN\"_
    ?TaskName, " _ENABLE := true\"_
\end{verbatim}
The result algorithm in the EC State for a task named P1 with interval 20ms will be:

IF Timestamp $\geq$ 20 & NOT(P1_ENABLE) & NOT(P1_ENABLE_ONS) THEN
  P1_ENABLE := true;
  P1_ENABLE_ONS := true;
END_IF;

The third SQWRL script is similar to the second query. Instead of searching periodic tasks, continuously tasks are replaced.

The second action of this step is to find all continuous tasks from the code knowledge base by executing the following query:

Task(?Task)
  ^ Has_Task_Name(?Task, ?Name)
  ^ Has_Task_Priority(?Task, ?Priority)
  ^ Has_Task_Type(?Task, ?Type)
  ^ swrlb:stringEqualIgnoreCase(?Type, "CONTINUOUS")
  ° sqwrl:makeSet(?Task, ?ContinuousTasksSet)
  ^ sqwrl:groupBy(?ContinuousTasksSet, ?Name, ?Priority)
-> sqwrl:select(?Name, ?Priority)

Similar to the periodic task, all names and priorities of continuous tasks are listed. When no periodic or higher priority continuous task is activated, the continuous task will keep executing.

All IEC 61131-3 tasks are mapped to IEC 61499 resources according to the third rule. The source of this action is the PLC task instance (see Task class in the Rockwell PLC ontology definition) and the destination is the resource type instance named EMB_RES (Embedded Resource) in the IEC 61499 ontology definition. The PLC task name is used as the FB resource name.
After the resource is constructed, a task scheduling FB is necessary for each IEC 61499 resource to schedule programs in this task. Task scheduling function blocks are introduced to the resource created previously and identical for both periodic and continuous tasks as well. In order to generate task schedulers, a list of programs and their execution orders is required in order to filling into the content template. A SQWRL query for selecting all programs in the execution order of a task is given as:

```
Task(?Task)
^ Has_Task_Name(?Task, ?TaskName)
^ swrlb:stringEqualIgnoreCase(?TaskName, <TaskName>)
^ Has_ScheduledProgram (?Task, ?Program)
^ ScheduledProgram(?Program)
^ Has_ScheduledProgram_Name(?Program, ?Name)
-> sqwrl:select(?Name)
```

For each program, a separate EC state, EC state algorithm and a pair of completed Boolean input and enable output are assigned into the task scheduling FB similar to the main scheduler FB. The EC state algorithm is generated following the same procedure as stated for the EC state of the main PLC scheduler FB.

The next step, for the mapping rule 4A, the program must be checked if any function or function block is invoked. The partial SQWRL queries are:

```
Program(?Prog)
^ Has_Function(?Prog, ?Func)
And the second rule:
Program(?Prog)
^ Has_FunctionBlock(?Prog, ? Func)
```

The check for function or function block exists in the mapping rule 4A can used in the opposite way to avoid duplicated generation of programs for the mapping rule 4B. Otherwise, the target code will be represented as multiple IEC 61499 function blocks. The rungs (in Ladder Diagram) or lines (in Structure Text) before the instance call of a function or function block call, after the instance call of a function or function block and between two functions or function blocks will be placed into a new basic function block. Those IEC 61499 function blocks are linked following the original execution order. A SQWRL query is used to list all functions and function blocks in a PLC program. The generated function block individuals are saved back into the ontological knowledge base. Similar approach is applied to the function or function block conversion if the IEC 61131-3 function or function block has nested structure inside. The contents written in either ladder logic diagram (LD), structure text (ST) and
instruction list (IL) are directly copied into the algorithm of the new generated basic function block. The program tags are created as the internal variables in the basic function block. The SQWRL query for getting all tags with data types is:

```sqwrl
Program(?Prog)
  ^ Has_Tags(?Prog, ?Tags1)
  ^ Has_Tag(?Tags1, ?Tag1)
  ^ Has_Tag_Name(?Tag1, ?Name1)
  ^ Has_Tag_DataType(?Tag1, ?DataType1)
- sqwrl:select(?Name1, ?DataType1)
```

The generation procedure for the program scheduling FB is almost equivalent to the task scheduler except the actual scheduling target are functions and function blocks inside the program instead of programs.

For the mapping rule 5, a SQWRL query is required for checking if the function block is programmed in the SFC for this step. In the Rockwell manner, the partial SQWRL query is given as:

```sqwrl
Routine(?FB1)
  ^ Has_SFCContent(?FB1, ?SFCContent1)
```

If the function block is programmed in the SFC, then all SFC steps are created as EC states in the function block version. The SQWRL query for getting all SFC steps, transitions, conditions are given as:

```sqwrl
Routine(?FB1)
  ^ Has_SFCContent(?FB1, ?SFCContent1)
  ^ Has_Step(?SFCContent1, ?Step1)
  ^ Has_Transition(?SFCContent1, ?Transition1)
  ^ Has_Condition(?SFCContent1, ?Cond1)
- sqwrl:select(?Step1, ?Transition1, ?Cond1)
```

The SFC transitions and transition conditions are generated as EC transitions and EC transition conditions in the function block ontology.

For the mapping rule 6, similar to the previous rule, a step of checking whether the function block is programed in the FBD language must be performed prior to the conversion from FBD to composite FB. The partial SQWRL query refers to the Rockwell PLC code is:

```sqwrl
Routine(?FB1)
  ^ Has_FBDContent(?FB1, ?FBDContent1)
```
If the function block is found as a FBD component, all connections in the FBD are converted to the data connections in the IEC 61499. All blocks used in the FBD program are still kept as function blocks in the IEC 61499 format. However, those function blocks must be existed in the target IEC 61499 IDE FB library. The SQWRL query for fetching the blocks and connections in the original PLC FBD is given as:

Routine(?FB1)
^ Has_FBDContent(?FB1, ?FBDContent1)
^ Has_Block(?FBDContent1, ?Block1)
^ Has_Wire(?FBDContent1, ?Wire1)
-> sqwrl:select(?Block1, ?Wire1)

Any extra initialization and request input event with acknowledgement output event are inserted for each block. The event connections of the new function blocks generated are linked in the order of the original block wires.

The last step of the transformation is to store global variables in a basic function block. The SQWRL query for selecting all global variables is:

Controller(?Controller1)
^ Has_Tags(?Controller1, ?Tags1)
^ Has_Tag(?Tags1, ?Tag1)
^ Has_Tag_Name(?Tag1, ?Name1)
^ Has_Tag_DataType(?Tag1, ?DataType1)
-> sqwrl:select(?Name1, ?DataType1)

For each tag in the result list, an internal variable is created in the global variable BFB and a pair of publisher/subscribe SIFB to each consumer function block is inserted in the system configuration.

After the transformation process, all scheduling function blocks are mapped. The execution order of the original PLC configuration is restored in the resulting function block system configuration.
6.3 Migration between IEC 61499-Compliant Tools

The transformation engine proposed in the previous section is applicable not only for converting PLC programs into function block systems, but also for translating projects between IEC 61499-Compliant tools. While FBDK, NxtStudio and other IEC 61499 IDEs are all following the standard, there are still some implementation variations which lead to incompatibility between them. The incompatibility between various tools can be solved by using the same mechanism.

In this migration process, the original and the destination knowledge base all generated from the IEC 61499 DTD files. However, various IEC 61499 dialects may have some different naming of same concept or some definitions which only supported by this platform. For example, nxtStudio introduced some keywords in the XML schema that do not exist in the IEC 61499 standard. As a result, the contents of T-Box for nxtStudio and FBDK are not 100% identical. The XML source code of FB applications created in any IEC 61499 IDE is interpreted by the same OWL import API using for PLC programs and stored into the related knowledge base in the OWL format.

At this point, the knowledge base contains all essential information required for migration. However, there is still another step before the actual code generation can be performed. The mapping rules must be defined prior to code generation. Those mapping rules are also defined in the form as shown in Figure 6.2. The example of migrating from NxtStudio to FBDK is used as the example here. Most of the classes defined in the two knowledge bases are identical. The source and destination nodes of the mapping rules for those nodes are equivalent. The SQWRL query is just ClassName(?a) and the VAR is just ?a. The content of the data template is ?a as well. For object and data properties of each node, they stay the same as before. The SQWRL query used for the object property is ClassName(?a) ^ Has_AnotherClass(?a, ?b) and the VAR is set for ?b. The data property is expressed in a similar way by using the query of ClassName(?a) ^ Has_Class_AttributeName(?a, ?b) and the VAR is also ?b.

Some classes defined in the NxtStudio ontological knowledge base are not required by the FBDK. Those classes are ignored during the migration process. However, there is still couple of exceptions. Firstly, the FBDK uses the date format as “yyyy-MM-
dd” but the NxtStudio treats the date as “MM/dd/yyyy”. The SQWRL query for changing the format of the date is:

```
System(?Sys1)
^ Has_VersionInfo(?Sys1, ?Ver1)
^ Has_VersionInfo_Date(?Ver1, ?Date1)
  swrlb:tokenize(?token1, ?Date1, "/")
  sqwrl:makeSet(?Set1, ?token1)
  sqwrl:nth(?year, ?Set1, 3)
  sqwrl:nth(?month, ?Set1, 1)
  sqwrl:nth(?day, ?Set1, 2)
  sqwrl:select(?year, ?month, ?day)
```

The content of the data template is defined as “?year-?month-?day”. By applying this mapping rule, the date format issue could be solved.

The other issue is that in the EC transition condition, the “[]” is used to split the event condition and the guard condition by the FBDK but this is not required by the nxtStudio. For this step, the SQWRL query is given as:

```
ECTransition(?ECTrans1)
^ Has_ECTransition_Condition(?ECTrans1, ?Cond1)
  swrlb:tokenize(?token1, ?Cond1, "AND")
  sqwrl:makeSet(?Set1, ?token1)
  sqwrl:nth(?eventcond1, ?Set1, 1)
  sqwrl:nth(?guardcond1, ?Set1, 2)
  sqwrl:select(?eventcond1, ?guardcond1)
```

The content of the data template is given as “?eventcond1 AND [?guardcond1]”.

Similar mapping rules could be applied to the reverse process converting from FBDK back to NxtStudio. If the target node is not defined in the original knowledge base, a node with default value will be inserted instead.

The last step in the design chain is the code generation. Based on the target platform, respected code templates and related instances are combined by the code generation engine and output as IEC 61499 XML files. The generated code is capable to be opened again in the target platform IDE and immediately deployed using its mechanisms of compilation and deployment.
6.4 Summary

To summarize, the ontological models defined in the previous section are used in the migration process implementation in this Chapter. The migration process between the PLC knowledge base and the Function Block knowledge base is described and illustrated on an example between Rockwell and FBDK. The generic definition template of the mapping rules is defined in the ontological knowledge base as well. The mapping rules defined in the Migration Models and Rules are implemented as ontology individuals. A transformation engine is built to process those rules in the ontological form and new instances are inserted into the target knowledge base. The main advantage of this approach is reliance on the Semantic Web technologies. The developed migration tool can use the standard S(Q)WRL engine that is configurable by the rules. This approach is more flexible and less resource consuming in development as compared to hard coding the transformation rules. The ontological knowledge base provides a higher abstract level view of the migration process. Followed by the same concept, the seamless migration between IEC 61499 standard compliant tools is also provided.
7 IEC 61499 Semantic Analysis

7.1 Introduction

After PLC projects are automatically transformed into the IEC 61499 function blocks, the last step in the migration process is to ensure the generated function block system configurations are semantically correct. The semantic analysis is an add-on to the migration process which can also be applied independently to IEC 61499 systems designed by the IEC 61499 IDE tools directly. In traditional programming languages, including those defined in the IEC 61131-3 standard, the syntactic and partial semantic check are done by software tools as a part of compilation or editing process.

The IEC 61499 standard uses XML representation for all design artefacts. The use of XML technologies simplifies the implementation of syntax checkers which can be delegated to the standard XML parsers. Such tools are easily configurable which is achieved by changing the corresponding XML schema in case of syntax modifications and extensions. However, the XML technology is of little help when it comes to spotting semantic problems. So an IEC 61499 application with semantic issues could pass the syntactic check.

The function block designs can be quite complicated thus difficult to debug and validate due to nested function block networks as well as massive event and data connections. The semantic analysis can increase performance and efficiency of developers and dependability of automation systems, and eventually help with adoption of the IEC 61499 technology in the automation industry. A small example of semantic check in the IEC 61499 context is implemented in the experimental FBDK tool [101]. The FBDK editor prohibits creating loops in state machines which do not include a transition triggered by an event input. Although they are syntactically correct, such loops can cause deadlocks in the program execution. However, the implementation of this semantic rule is hard-coded in the tool and cannot be extended or modified if the syntax or semantics of ECC changes. For example, it was the case recently with the second edition of the standard introduced. Unfortunately, no
A novel methodology of configurable semantic check for IEC 61499 function blocks is developed in this Chapter. Its configurability is akin to the XML-supported syntax check, but implementation is also based on the ontologies and S(Q)WRL which are used to represent the semantic model of programs and extensible set of semantic rules. The proposed semantic analysis methodology can be conveniently integrated into existing IEC 61499 development environments.

As the IEC 61499 code is represented in XML, the corresponding DTD is accomplished in the IEC 61499 standard. The DTD defines all possible elements and properties which are allowed to be used in the IEC 61499 XML documents. Another form of XML syntax description is XML Schema (XSD). Different from DTD, XSD uses XML to describe syntax of custom XML documents. In addition to DTD, XSD supports all data types including user defined data types, whereas DTD only supports generic character types CDATA and PCDATA. As described in the previous Chapter, both XSD and DTD can be used as syntax description sources. The DTD version is used as the example here.

However, some implementations of general purpose programming languages use semantic rules hard-coded in compilers, but this approach is not flexible. Any change of the input language, or introduction of new semantic rules, will require changing the compiler.

A more flexible approach is proposed here where semantic checker is using an external (and thus extendable) set of semantic rules. The rules refer to the knowledge model representing the related concepts of the function blocks architecture. More complex rules can be built on top of existing ones. The semantic rules defined in the ontology are both human and machine readable and self-explanatory.

The rest of the Chapter is given as following. In the next section, some syntactic corrections to the existing IEC 61499 standard is proposed. In the section 7.3, the complete set of the semantic rules using S(Q)WRL for the IEC 61499 standard are provided. In the section 7.4, the automatic semantic analysis engine is described. Finally, this Chapter is concluded in the section 7.5.
7.2 IEC 61499 Syntactic Corrections

The ontological model of a particular artefact from the function block knowledge base A-box can be exposed to the semantic analysis. However, there are some syntactic correctness of the artefact must be established. The generated DTD-based ontological model can be also used for that purpose. In this methodology, a function block type is syntactically correct if all related object and data properties exist in the A-Box and the quantity constraints are also satisfied. Nevertheless, there are also some contradictions in the syntax definition in the standard which need to be resolved prior to conducting the syntax check. Those uncertainties are clarified as following and included as a part of semantic rules.

There are three types of function block representation: graphical, textual and XML definition. The textual representation is defined in the Extended Backus-Naur Form (EBNF) [107] in the Part 1 Annex B of the IEC 61499 standard. The XML DTD definition is given in the Part 2 of IEC 61499-2. However, the textual format is not always equivalent to the XML format. Some add-on rules defined in SWRL are required to match the XML format with the textual format. In this section, all three representations, namely: DTD, the textual representation and SWRL rules, will be compared using some reference examples.
First, the basic function block, composite function block and service interface function block are mixed as a single node \textit{FBType} in the DTD-based ontological model. To distinguish those function block types, the SWRL rule is defined according to the textual syntax to include three separate types of function blocks as shown in Figure 7.1. A basic FB type has maximum one ECC, a composite FB type has exactly one FB network and a Service Interface FB has exactly one service inside. Those new classes of concepts are also made disjoint.

### DTD-based syntax

\[
\text{<!ELEMENT FBType (Identification?,VersionInfo+,CompilerInfo?,InterfaceList, (BasicFB | FBNetwork)?, Service?) >}
\]

### Textual syntax

\[
\text{fb_type_declaration ::= }
\]

\[
\text{FUNCTION\_BLOCK\ fb\_type\_name}
\]

\[
\text{fb\_interface\_list}
\]

\[
\text{[fb\_internal\_variable\_list] <only for basic FB>}
\]

\[
\text{[fb\_instance\_list] <only for composite FB>}
\]

\[
\text{[plug\_list]}
\]

\[
\text{[socket\_list]}
\]

\[
\text{[fb\_connection\_list] <only for composite FB>}
\]

\[
\text{[fb\_ecc\_declaration] <only for basic FB>}
\]

\[
\text{[fb\_algorithm\_declaration] <only for basic FB>}
\]

\[
\text{[fb\_service\_declaration] <only for service interface FB>}
\]

\[
\text{END\_FUNCTION\_BLOCK}
\]

### SWRL Rules

\[
\text{FBType(?FBType1) ^ (has\_ECC <= 1)(?FBType1) \rightarrow BasicFBType(?FBType1)}
\]

\[
\text{FBType(?FBType1) ^ (has\_FBNetwork = 1)( ?FBType1,) \rightarrow CompositeFBType(?FBType1)}
\]

\[
\text{FBType(?FBType1) ^ (has\_Service = 1)( ?FBType1,) \rightarrow ServiceInterfaceFBType(?FBType1)}
\]

Note:

\[
\text{BasicFBType \cap CompositeFBType \cap ServiceInterfaceFBType} = \emptyset
\]

(disjoint classes)

Figure 7.1: DTD, Textual and SWRL add-on rule Syntax for Basic FB Definition.
Secondly, an EC Action definition is described in the DTD (Figure 7.2) as it can have up to one algorithm and up to one output or none at all. But in the textual definition, an EC action must have at least one algorithm, or an output event, or both. The SWRL rule allows having at least one of algorithm and output event. The two SWRL rules for $ECActionCorrect$ constitute the logical OR of two antecedents.

**DTD-based syntax**

```xml
<!ELEMENT ECState (ECAction*)>
<!ATTLIST ECAction
    Algorithm CDATA #IMPLIED
    Output CDATA #IMPLIED >
```

**Textual syntax**

```
ec_action ::= algorithm_name | '->' event_output_name
             | algorithm_name '->' event_output_name
```

**SWRL Rule**

```
ECAction(?ECAction1) ^ Has_Output(?ECAction1, ?Output1) ->
ECActionCorrect(?ECAction1);

ECAction(?ECAction1) ^ Has_Algorithm(?ECAction1, ?Alg1) ->
ECActionCorrect(?ECAction1);
```

---

Figure 7.2: DTD, Textual and SWRL add-on rule for EC Action Definition.
Finally, different types of connections are not distinguished from the DTD-based syntax. All event, data and adapter connections in FBs and Sub-Applications are declared as a connection with attribute values of a source and a destination. Also in the textual syntax, not all possible connections are listed. For instance, an event input is allowed to connect to an event input of a FB instance or event output of a plug only. Indeed, an event input can also connect to an event input of a socket and an event output directly. The semantically correct event connection of a function block and the refined ontology model are presented in Figure 7.3.

<table>
<thead>
<tr>
<th>DTD-based syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;!ELEMENT EventConnections (Connection+)&gt; &lt;!ATTLIST Connection Source CDATA #REQUIRED Destination CDATA #REQUIRED Comment CDATA #IMPLIED dx1 CDATA #IMPLIED dx2 CDATA #IMPLIED dy CDATA #IMPLIED&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Textual syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>event_conn ::= ((fb_instance_name '.' event_output_name)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ontology-based syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event_connection ⊑ Connection ⊓ Event ⊓ Has_Source.Event ⊓ Has_Destination.Event</td>
</tr>
</tbody>
</table>

Figure 7.3: DTD, Textual and Ontology Syntax for Event Connection Definition.

Similar SWRL rules can be developed to correct other syntactic issues between the DTD format and the textual based format in the future.
7.3 IEC 61499 Semantic Rules

The semantic analysis includes not only checking the system configurations and parameters (for example, that all sources and destinations of data connections have identical data types), but also the execution behaviours (for instance, event deadlock) [91]. However, pure OWL DL is not sufficient to present semantic rules in such complex level. Therefore SWRL is used here as the semantic rule language.

In order to interpret SWRL rules, the SQWRL language is selected. The ontology query language SQWRL [97] as an extension of SWRL allows more comprehensive representation of reasoning results similar to SQL queries. SQWRL takes a standard SWRL rule antecedent and treats it as a pattern specification for a query [108] [109]. SQWRL has some essential extensions as against SWRL, for example, set and string operations.

Semantic rules represented in S(Q)WRL can be regarded as scripts written in a very high level programming language in order to search for a particular kind of semantic problem in applications complying with IEC 61499 standard. The ontological model can be compared to an API that provides access to basic artefacts of the program being analysed. The benefits of the proposed approach are:

1) Compactness of the “scripts” – the search engine is built based on S(Q)WRL interpretation;
2) The “API” is generated automatically based on the syntax of input language;
3) Off-the-shelf tools (like Protégé) can be used as the ontology viewer and editor.

In the remainder of this section, the semantic rules are divided into various catalogues for each function block type and system configuration. SQWRL will be used for the rules definition due to a number of its powerful features as compared to SWRL. Some SQWRL rules are based on the description logic and can be expressed in the description logic format. However, these set of rules are not comprehensive for covering the entire IEC 61499 semantic analysis. Those rules are used mainly for illustrate the idea of applying ontology with SQWRL technology to the IEC 61499 semantic analysis. The proposed semantic analysis method is demonstrated by applying those rules and not really counts towards the thesis contribution.
7.3.1 Basic Function Block Semantic Rules

Firstly, the semantic correctness of a basic FB is defined as follows.

A basic function block type is said to be semantically correct if all of the following rules are satisfied:

**Rule 1**

Inside the ECC, no identical EC transition condition (triggered by same event and guard condition) is allowed from a single EC state;

**Rule 2**

If an EC state connects to more than one EC state, no always true “1” is allowed in any EC transition condition of those connected EC states;

**Rule 3**

Each EC state must have at least one entry and one exit EC transition;

**Rule 4**

Each event input is used in at least one EC transition condition;

**Rule 5**

Each event output is used as at least one EC action output event;

**Rule 6**

Each data input variable is associated with at least one event input of this basic FB;

**Rule 7**

Each data output variable is associated with at least one event output of this basic FB.
The first three semantic rules ensure the execution semantic of ECC is correct. For example, the first rule states that if an EC state is linked to multiple EC states, no identical EC transition condition which means triggered by same event and guard condition is permitted. Identical EC transition conditions can cause some EC states to be never reachable. This rule is presented as a SQWRL query below:

```
FBType(?FBType1)
^ Has_FBType_Name(?FBType1, ?name)
^ Has_BasicFB(?FBType1, ?BFB1)
^ Has_ECC(?BFB1, ?ECC1)
^ Has_ECState(?ECC1, ?ECState1)
^ Has_ECState_Name(?ECState1, ?name1)
^ Has_ECTransition(?ECC1, ?ECT1)
^ Has_ECTransition_Source(?ECT1, ?source1)
^ swrlb:stringEqualIgnoreCase(?source1,?name1)
^ Has_ECTransition_Condition(?ECT1, ?cond1)
^ sqwrl:makeSet(?set1, ?cond1)
^ sqwrl:groupBy(?set1, ?name, ?name1)
^ sqwrl:makeBag(?bag1, ?cond1)
^ sqwrl:groupBy(?bag1, ?name, ?name1)
^ sqwrl:notEqual(?set1, ?bag1)
^ sqwrl:select(?name, ?name1)
```

In the SQWRL expression above, the left hand side is a SWRL rule antecedent. All operators starting with “sqwrl:” are the built-in functions from SQWRL. All EC transitions with identical conditions grouped by EC transition source names will be listed by this SWRL rule. Swrlb:stringEqualIgnoreCase is a SWRL built-in function which compares two string value. Sqwrl:makeSet operator is used to construct sets of results without duplicate elements in the set. Sqwrl:makeBag operator is similar to the Sqwrl:makeSet operator but duplicate elements are allowed in the bag. Sqwrl:notEqual compares two sets/bags to check if every element from both sides are equal. To retrieve elements from a set/bag, Sqwrl:element is used to list all elements. The query results are stored as a list of EC transition condition named as ?e1.
As illustrated in Figure 7.4, the ontology search is starting from the root node \( FBTtype \) (Ontology node presented in circle) in the SQWRL query. In the search, \(?FBType1\) denotes the set of names for the \( FBTtype \) which are found during the search. \( BasicFB \), \( ECC \) and \( ECTransition \) are also the ontology concepts which each object is associated to a set variable respectively. \( Has\_BasicFB \), \( Has\_ECC \) and \( Has\_ECTransition \) are the object properties of the node \( FBTtype \) linked between nodes by solid lines. The instances are linked to their relative concept nodes in T-Box by dashed lines. \( Has\_ECTransition\_Source \) and \( Has\_ECTransition\_Condition \) are the data properties of the node \( ECTransition \).

![Figure 7.4: Graphical View of FBTtype SQWRL Search for Basic FB.](image_url)

The right hand side \( sqwrl:select \) will build a table using arguments as columns of the table. In this rule, the result EC transition condition individuals, which satisfied all the properties’ conditions, are stored in the list represented by the variable name \(?e1\). When an \( FBTtype \) individual satisfies all properties (for instance, \( has\_BasicFB \)) as
well as the built-in functions (for instance, compare two sets/bags), this instance is included into the search results.

The second rule is to prevent a short-circuit EC transition in an ECC. If an always true condition “1” exists in an EC state with multiple EC outgoing transitions, all other transitions will be bypassed regardless. The SQWRL expression of the second rule is as follows:

\[
\text{FBType(?FBType1)} \wedge \text{Has_FBType_Name(?FBType1, ?name)} \wedge \text{Has_BasicFB(?FBType1, ?BasicFB1)} \wedge \text{Has_ECC(?BasicFB1, ?ECC1)} \wedge \text{Has_ECState(?ECC1, ?ECState1)} \wedge \text{Has_ECState_Name(?ECState1, ?name1)} \wedge \text{Has_ECTransition(?ECC, ?ECT1)} \wedge \text{Has_ECTransition_Source (?ECT1, ?Source1)} \swrlb:stringEqualIgnoreCase(?source1, ?name1) \wedge \text{Has_ECTransition_Condition(?ECT1, ?Cond1)} \swrl:makeBag(?bag1, ?cond1) \swrl:groupBy(?bag1, ?name, ?name1) \swrl:makeSet(?set1, 1) \swrl:size(?bag_size, ?bag1) \swrl:greaterThan(?bag_size, 1) \swrl:contains(?bag1, ?set1) \rightarrow \text{swrl:select(?name, ?name1)}
\]

\text{Sqwr:l:size} is a SQWRL function which counts the number of elements in a set or bag and stored in the variable indicated in the first operand. \text{Sqwr:l:contains} is a SQWRL built-in function which checks if a set/bag has all elements from another set/bag. \text{Swrlb:greaterThan} is a SWRL built-in function which compares its operands. In this semantic rule, all EC transitions with condition of always true having more than one EC transition are listed for correction.
The third rule is used to detect dead end of execution in the event chain. If an EC state has no output transition, once the execution of that function block instance is completed, it will stuck in that state forever and never execute anymore. The rule is given as two parts. The first part is for EC states without outgoing EC transitions:

\[
\begin{align*}
&\text{FBType}(?\text{FBType}1) \\
&^\land \text{Has_FBType_Name}(?\text{FBType}1, ?\text{name}) \\
&^\land \text{Has_BasicFB}(?\text{FBType}1, ?\text{BasicFB}1) \\
&^\land \text{Has_ECC}(?\text{BasicFB}1, ?\text{ECC}1) \\
&^\land \text{Has_ECState}(?\text{ECC}1, ?\text{ECState}1) \\
&^\land \text{Has_ECState_Name}(?\text{ECState}1, ?\text{name}1) \\
&^\land \text{Has_ECTransition}(?\text{ECC}1, ?\text{ECT}1) \\
&^\land \text{Has_ECTransition_Source}(?\text{ECT}1, ?\text{source}1) \\
&^\ast \text{sqwrl:makeSet}(?\text{set}1, ?\text{name}1) \\
&^\ast \text{sqwrl:groupBy}(?\text{set}1, ?\text{name}) \\
&^\ast \text{sqwrl:makeSet}(?\text{set}2, ?\text{source}1) \\
&^\ast \text{sqwrl:groupBy}(?\text{set}2, ?\text{name}) \\
&^\ast \text{sqwrl:difference(?set, ?set1, ?set2)} \\
&^\ast \text{sqwrl:element(?e1, ?set)} \\
&\rightarrow \text{sqwrl:select(?name, ?e1)}
\end{align*}
\]

And second part is for EC states without incoming EC transitions:

\[
\begin{align*}
&\text{FBType}(?\text{FBType}1) \\
&^\land \text{Has_FBType_Name}(?\text{FBType}1, ?\text{name}) \\
&^\land \text{Has_BasicFB}(?\text{FBType}1, ?\text{BasicFB}1) \\
&^\land \text{Has_ECC}(?\text{BasicFB}1, ?\text{ECC}1) \\
&^\land \text{Has_ECState}(?\text{ECC}1, ?\text{ECState}1) \\
&^\land \text{Has_ECState_Name}(?\text{ECState}1, ?\text{name}1) \\
&^\land \text{Has_ECTransition}(?\text{ECC}1, ?\text{ECT}1) \\
&^\land \text{Has_ECTransition_Destination}(?\text{ECT}1, ?\text{dest}1) \\
&^\ast \text{sqwrl:makeSet}(?\text{set}1, ?\text{name}1) \\
&^\ast \text{sqwrl:groupBy}(?\text{set}1, ?\text{name}) \\
&^\ast \text{sqwrl:makeSet}(?\text{set}2, ?\text{dest}1) \\
&^\ast \text{sqwrl:groupBy}(?\text{set}2, ?\text{name}) \\
&^\ast \text{sqwrl:difference(?set, ?set1, ?set2)} \\
&^\ast \text{sqwrl:element(?e1, ?set)} \\
&\rightarrow \text{sqwrl:select(?name, ?e1)}
\end{align*}
\]

This search requires an EC transition to have both source and destination. If a FB type is in the result list, either the EC transition source or destination of that FB type is missing.
The SQWRL implementation of the Rule 4 also uses set operations. The rule finds the set of input events which are not included in any EC transition. The rule’s description is as follows:

FBType(?FBType1)
^ Has_InterfaceList(?FBType1, ?List1)
^ Has_EventInputs(?List1, ?EI1)
^ Has_Event(?EI1, ?EV1)
^ Has_Event_Name(?EV1, ?EV1_Name)
^ Has_BasicFB(?FBType1, ?BasicFB1)
^ Has_ECC(?BasicFB1, ?ECC1)
^ Has_ECTransition(?ECC1, ?ECT1),
^ Has_ECTransition_Condition(?ECT1, ?ECCond1)
^ swrlb:normalizeSpace(?ECCond1, ?ECCond1_Nospace)
^ swrlb:substringBefore(?ECCond1_Nospace, “&”, ?EV2_Name)° sqwrl:makeSet(?EV1Set, ?EV1_Name)
^ sqwrl:makeSet(?EV2Set, ?EV2_Name)
^ sqwrl:difference(?EVSet, ?EV1Set, ?EV2Set)
  -> sqwrl:select(?FBType1, ?EVSet)

The first part of the query will return all event input names of a single basic FB. If the event input name, being looked for, is in the result list, this event input is a part of this basic FB type interface. The second part is to gather all events used in the EC transition conditions. The event names are located before “&” symbol in EC transition conditions. Finally the set difference operator is used to find events that are in the first set but not in the second one. It should be noted that the above rule is intended for the situation when EC-transition condition consists of two parts – an event input and a guard condition (In this case they are delimited by the sign “&”) or only event input is presented. For other cases different rules are developed.
Similar to the Rule 4, the Rule 5 is applied to event outputs and ensures event outputs are used in the ECC at least once. The rule is given as:

```
FBType(?FBType1)
^ Has_InterfaceList(?FBType1, ?List1)
^ Has_EventOutputs(?List1, ?EO1)
^ Has_Event(?EO1, ?EV1)
^ Has_Event_Name(?EV1, ?EV1_Name)
^ Has_BasicFB(?FBType1, ?BasicFB1)
^ Has_ECC(?BasicFB1, ?ECC1)
^ Has_ECState(?ECC1, ?ECState1)
^ Has_ECAction(?ECState1, ?ECAct1)
^ Has_ECAction_Output(?ECState1, ?ECOutput1)
  ^ sqwrl:makeSet(?EV1Set, ?EV1_Name)
  ^ sqwrl:makeSet(?EV2Set, ?ECOutput1)
  ^ sqwrl:difference(?EVSet, ?EV1Set, ?EV2Set)
  ^ sqwrl:select(?FBType1, ?EVSet)
```

All event output names in this basic function block are listed as a set (?EV1Set) in this rule. Also the event output used in all EC actions are stored in another set (?EV2Set). The different elements between two sets are indicating event outputs are not emitted by any EC state.

The Rule 6 ensures every data input is updated by at least one event input. No dummy data input exists in the function block interface. The SQWRL query for this rule is:

```
FBType(?FBType1)
^ Has_InterfaceList(?FBType1, ?List1)
^ Has_EventInputs(?List1, ?EI1)
^ Has_Event(?EI1, ?EV1)
^ Has_With(?EV1, ?With1)
^ Has_With_Var(?With1, ?WithVar1)
^ Has_InputVars(?List1, ?DI1)
^ Has_VarDeclaration(?DI1, ?InVar1)
^ Has_VarDeclaration_Name(?InVar1, ?InVarName1)
  ^ sqwrl:makeSet(?EV1Set, ?WithVar1)
  ^ sqwrl:makeSet(?EV2Set, ?InVarName1)
  ^ sqwrl:difference(?EVSet, ?EV1Set, ?EV2Set)
  ^ sqwrl:select(?FBType1, ?EVSet)
```

The data input variables associated with at least one event are compared to the all data input variables defined in the interface. The difference shows the data input variables are declared but never assigned with any event.
The last rule for the basic function block is similar to the previous rule. Except the input data variables associated with event inputs, those are replaced by the output data variables and event outputs. The SQWRL expression for this rule is shown as:

```
FBType(?FBType1)
^ Has_InterfaceList(?FBType1, ?List1)
^ Has_EventOnputs(?List1, ?EO1)
^ Has_Event(?EO1, ?EV1)
^ Has_With(?EV1, ?With1)
^ Has_With_Var(?With1, ?WithVar1)
^ Has_OutputVars(?List1, ?DO1)
^ Has_VarDeclaration(?DO1, ?InVar1)
^ Has_VarDeclaration_Name(?InVar1, ?InVarName1)
  sqwrl:makeSet(?EV1Set, ?WithVar1)
^ sqwrl:makeSet(?EV2Set, ?InVarName1)
^ sqwrl:difference(?EVSet, ?EV1Set, ?EV2Set)
  sqwrl:select(?FBType1, ?EVSet)
```

### 7.3.2 Service Interface Function Block Semantic Rules

After the basic function block semantic rules are defined, the target now is to define the semantic rules for the service interface function block. As service interface function blocks are platform dependent, the rules listed below are not involving any actual implementation of the service interface function block. The semantic rules are majorly checking for events and data variables.

A service interface function block type is said to be semantically correct if all of the following rules are satisfied:

**Rule 1**

Each event input is used in at least one input primitive;

**Rule 2**

Each event output is used in at least one output primitive;

**Rule 3**

Each data input variable is associated with at least one event input of this SIFB;

**Rule 4**

Each data output variable is associated with at least one event output of this SIFB.
The first rule is designed for avoiding dummy event input in the interface list. The SQWRL query for this rule is defined as:

```
FBType(?FBType1)
^ Has_InterfaceList(?FBType1, ?List1)
^ Has_EventInputs(?List1, ?EI1)
^ Has_Event(?EI1, ?EV1)
^ Has_Event_Name(?EV1, ?EV1_Name)
^ Has_Service(?FBType1, ?Service1)
^ Has_ServiceSequence(?Service1, ?Seq1)
^ Has_ServiceTransaction(?Seq1, ?ServiceTran1),
^ Has_InputPrimitive(?ServiceTran1, ?InputPrim1)
^ Has_InputPrimitive_Event(?InputPrim1, ?InEvent1)
^ swrlb:substringBefore(?EventName1, ?InEvent1, “+”)
  sqwrl:makeSet(?EV1Set, ?EV1_Name)
  sqwrl:makeSet(?EV2Set, ?EventName1)
  sqwrl:difference(?EVSet, ?EV1Set, ?EV2Set)
-> sqwrl:select(?FBType1, ?EVSet)
```

In this rule, the first set is still structured by the event inputs declared in the function block interface list. The event inputs actually used in a SIFB input primitive is split by “+” sign in the data property `Event` of the node `InputPrimitive`. The results show the events are not used anywhere in this function block.

The second rule is very close to the first rule. Instead of event inputs and input primitive, event outputs and output primitive are used. The Rule 2 is illustrated as:

```
FBType(?FBType1)
^ Has_InterfaceList(?FBType1, ?List1)
^ Has_EventOutputs(?List1, ?EO1)
^ Has_Event(?EO1, ?EV1)
^ Has_Event_Name(?EV1, ?EV1_Name)
^ Has_Service(?FBType1, ?Service1)
^ Has_ServiceSequence(?Service1, ?Seq1)
^ Has_ServiceTransaction(?Seq1, ?ServiceTran1),
^ Has_OutputPrimitive(?ServiceTran1, ?OutputPrim1)
^ Has_OutputPrimitive_Event(?OutputPrim1, ?OutEvent1)
^ swrlb:substringBefore(?EventName1, ?OutEvent1, “+”)
  sqwrl:makeSet(?EV1Set, ?EV1_Name)
  sqwrl:makeSet(?EV2Set, ?EventName1)
  sqwrl:difference(?EVSet, ?EV1Set, ?EV2Set)
-> sqwrl:select(?FBType1, ?EVSet)
```

The Rule 3 and the Rule 4 are identical to the Rule 6 and Rule 7 from the basic function block semantic rules. The details of SQWRL query please refer to the previous section.
7.3.3 Composite Function Block/Sub-Application Semantic Rules

The last function block type available in the IEC 61499 standard is the composite function block. A composite function block has an internal function block network using combination of basic function blocks, service interface function blocks and itself. A composite FB type is said to be semantically correct if all of the following rules are satisfied:

**Rule 1**

The source and the destination data variable in a data connection must have consistent data types;

**Rule 2**

Each event input is connected to an event input inside the included FB network;

**Rule 3**

Each event output is connected from an event output inside the included FB network;

**Rule 4**

Each data input variable is associated with at least one event input of this composite FB interface;

**Rule 5**

Each data output variable is associated with at least one event output of this composite FB interface.

**Rule 6**

No dangerous event loop is allowed.

**Rule 7**

No short-circuit event is allowed.
The first rule of the composite function block is used to check data type consistency for all data connections in the internal function block network. The SQWRL query for this rule is provided as:

```
FBType(?FBType1)
^ Has_FBNetwork(?FBType1, ?FBNet1)
^ Has_DataConnections(?FBNet1,?DCs1)
^ Has_Connection(?DCs1, ?Conn1)
^ Has_Connection_Source(?Conn1, ?Source1)
^ Has_Connection_Destination(?Conn1, ?Dest1)
^ Has_InterfaceList(?FBType1, ?List1)
^ Has_OutputVars(?List1,?DO1)
^ Has_VarDeclaration(?DO1, ?OVar1)
^ Has_VarDeclaration_Name(?OVar1, ?OVarName1)
^ swrlb:stringEqualIgnoreCase(?Source1, ?OVarName1)
^ Has_VarDeclaration_Type(?OVar1, ?OVarType1)
^ Has_InputVars(?List1,?DI1)
^ Has_VarDeclaration(?DI1, ?IVar1)
^ Has_VarDeclaration_Name(?IVar1, ?IVarName1)
^ swrlb:stringEqualIgnoreCase(?Dest1, ?IVarName1)
^ Has_VarDeclaration_Type(?IVar1, ?IVarType1)
^ sqwrl:makeSet(?EV1Set, ?OVarType1)
^ sqwrl:makeSet(?EV2Set, ?IVarType1)
^ sqwrl:difference(?EVSet, ?EV1Set, ?EV2Set)
^ sqwrl:select(?FBType1, ?EVSet)
```

The first part of this rule is used to list sources (?Source1) and destinations (?Dest1) of all data connections for comparison. Then the middle part is for gathering data types of all input and output data variable whose names exist in the data connections. The input data variable data types and output data variable data types are compared to see if any difference. The results show the inconsistent data types between the source and the destination variables.

The second rule is to ensure no dummy input is declared for this composite function block. The complete SQWRL query is shown below:

```
FBType(?FBType1)
^ Has_InterfaceList(?FBType1, ?List1)
^ Has_EventInputs(?List1,?EI1)
^ Has_Event(?EI1, ?EV1)
^ Has_Event_Name(?EV1, ?EV1_Name)
^ Has_FBNetwork(?FBType1, ?FBNet1)
^ Has_EventConnections(?FBNet1,?EConns1)
^ Has_Connection(?EConns1, ?Conn1)
^ Has_Connection_Source(?Conn1, ?Source1)
^ sqwrl:makeSet(?EV1Set, ?EV1_Name)
^ sqwrl:makeSet(?EV2Set, ?Source1)
^ sqwrl:notContains(?EV2Set, ?EV1Set)
^ sqwrl:element(?e1, ?EV1Set)
^ sqwrl:select(?FBType1, ?e1)
```
Two sets are compared in the query: the first set is all event names declared in the interface list and the second set is the source of events exist in the internal function block. The sqwrl:notContains is a SQWRL built-in library for comparing the target collection is not containing all elements in the original collection. Those elements are printed by the select clause.

The only differences between the third rule and the second rule are replacing the event inputs with the event outputs and changing from sources to destinations in the event connections. The entire SQWRL query is given as:

FBType(?FBType1) ^ Has_InterfaceList(?FBType1, ?List1) ^ Has_EventOutputs(?List1,?EO1) ^ Has_Event(?EO1, ?EV1) ^ Has_Event_Name(?EV1, ?EV1_Name) ^ Has_FBNetwork(?FBType1, ?FBNet1) ^ Has_EventConnections(?FBNet1,?EConns1) ^ Has_Connection(?EConns1,?Conn1) ^ Has_Connection_Destination(?Conn1, ?Dest1) ^ sqwrl:makeSet(?EV1Set, ? EV1_Name) ^ sqwrl:makeSet(?EV2Set, ? Dest1) ^ sqwrl:notContains(?EV2Set, ?EV1Set) ^ sqwrl:element(?e1, ?EV1Set) -> sqwrl:select(?FBType1, ?e1)

The Rule 4 and 5 are identical to the Rule 6 and Rule 7 of the basic function block semantic rules.

Figure 7.5: Dangerous Event Loop and Short-Circuit Event.
An interesting part of a composite FB is the event connections in FB network. As defined in the rules 6 and 7, dangerous event loop or short-circuit event can cause unexpected behaviour in the execution. Before start detecting event loops, an event chain is defined as:

An event chain in a FB network is a sequence of event connections $ec_1, ec_2, \ldots, ec_n$ which satisfies the following conditions:

i) An event connection $ec_i$ is connected to an event input of a FB instance $fbi$ and another event connection $ec_{i+1}$ is connected from an event output of this FB instance $fbi$ (where $1 \leq i < n$);

ii) $fbi$ is “reactive” regarding to $ec_i$ and $ec_{i+1}$, that means an event input from $ec_i$ triggers an event output to $ec_{i+1}$ via some algorithms inside $fbi$;

An event chain is non-stopable if $ec_1=ec_n$ which forms an event cycle. A non-stopable event chain is considered as a dangerous event loop or a deadlock event if there is an external event input that initializes the event loop like a SIFB (Figure 7.5 (A)).

The conditions of dangerous event loops are different for each FB type and can be defined using the concept of causality. In a FB an event input $ei_1$ triggers an event output $eo_1$ if the following conditions are true:

**For Basic FB**

i) The event input $ei_1$ is the only EC transition condition attached for an EC transition;

ii) Associated with this EC state and EC transition there is an EC action firing the event output $eo_1$. A scheme for the event flow through a basic FB is shown in Figure 7.6.

![Figure 7.6: Event flow through a basic FB.](image-url)
An implementation of above concepts using S(Q)WRL rules is presented as following.

Here event chains are defined using OWL object properties \textit{EvFlow} and \textit{EvFlow\_transitive} in SWRL. The event flow via a basic FB can be defined as the following SWRL rule. At that, parameter \textit{conn1} of the rule consequent \textit{EvFlow} represents a preceding event connection and parameter \textit{conn2} represents a successive event connection in an event chain.

\begin{verbatim}
FBType(?FBType1)
^ Has_FBNetwork(?FBType1,?FBNetwork)
^ Has_EventConnections(?FBNetwork,?conns)
^ Has_Connection(?conns, ?conn1)
^ Has_Connection_Destination(?conn1, ?dest1)
^ Has_Connection(?conns, ?conn2)
^ Has_Connection_Source(?conn2, ?source2)
^ swrlb:SubstringBefore(?dest1, ".", ?FBName)
^ swrlb:SubstringBefore(?source2, ".", ?FBName)
^ swrlb:SubstringAfter(?dest1, ".", ?evname1)
^ swrlb:SubstringAfter(?source2, ".", ? evname2)
^ Has_FB(?FBNetwork,?FB)
^ Has_FB_Name(?FB,?FBName)
^ Has_FB_Type(?FB,?FBType2)
^ Has_BasicFB(?FBtype2,?BFBType)
^ Has_ECC(,?BFBType,?ECC)
^ Has_ECTransition(?ECC,?ECTran)
^ Has_ECTransition_Condition(?ECTran,?evname1)
^ Has_ECTransition_Destination(?ECTran,?ECState)
^ Has_ECAction(?ECState,?ECAction)
^ Has_ECAction_Output(?ECAction,?evname2)
->EvFlow(?conn1,?conn2)
\end{verbatim}

\textbf{For SIFB}

There is a service, a service sequence, and a service transaction such that this service transaction has an input primitive triggered by the event input \textit{ei}$_1$ and an output primitive fires the event output \textit{eo}$_1$ (as illustrated in Figure 7.7).

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{Figure7.7.png}
\caption{Event flow through a SIFB.}
\end{figure}
The following SWRL rule expresses the event flow through SIFB.

```
FBType(?FBType1)
^ Has_FBNetwork(?FBType1, ?FBNetwork)
^ Has_EventConnections(?FBNetwork, ?conns)
^ Has_Connection(?conns, ?conn1)
^ Has_Connection Destination(?conn1, ?dest1)
^ Has_Connection(?conns, ?conn2)
^ Has_Connection Source(?conn2, ?source2)
^ swrlb:SubstringBefore(?dest1, ".", ?FBName)
^ swrlb:SubstringBefore(?source2, ",", ?FBName)
^ swrlb:SubstringAfter(?dest1, ",", ?evname1)
^ swrlb:SubstringAfter(?source2, ",", ?evname2)
^ Has_FB(?FBNetwork, ?FB)
^ Has_FB_Name(?FB, ?FBName)
^ Has_FB_Type(?FB, ?FBType2)
^ Has_Service(?FBtype2, ?ser)
^ Has_ServiceSequence(?ser, ?ss)
^ Has_ServiceTransaction(?ss, ?st)
^ Has_InputPrimitive(?st, ?inprim)
^ Has_InputPrimitive_Event(?inprim, ?evname1)
^ Has_OutputPrimitive(?st, ?outprim)
^ Has_OutputPrimitive_Event(?outprim, ?evname2)
-> EvFlow(?conn1, ?conn2)
```

**For Composite FB**

In the FB network inside a composite FB, there is an unstoppable event chain beginning at the event input $e_i_1$ and terminating at the event output $e_o_1$.

However, the relation $EvFlow$ defined in the BFB or SIFB part only gives a pair of event connections via only one function block instance. The following SWRL rules are used for computing transitive closure of relation $EvFlow$ that search through an event chain recursively in the nested function blocks structure.

```
EvFlow (?conn1, ?conn2)
-> EvFlow_Transitive(?conn1, ?conn2)
```

```
EvFlow (?conn1, ?conn2)
^ EvFlow_Transitive(?conn2, ?conn3)
-> EvFlow_Transitive(?conn1, ?conn3)
```

The SQWRL query below prints all event inputs and outputs of non-stoppable event chains of a flat FB network using the above SWRL rule for $EvFlow_Transitive$.

```
EvFlow_Transitive (?conn, ?conn)
^ Has_Connection_Source(?conn, ?source)
^ Has_Connection_Destination(?conn, ?dest)
-> sqwrl:select(?source, ?dest)
```
If the detected function block type is a composite function block, the same SWRL rule is applied to find the event output from the nested function blocks with internal event connections.

The short-circuit event as illustrated in Figure 7.5 (B) is similar to the dangerous event loop rule. If an event is found bypassed a function block and that function block is non-stoppable, this will cause a semantic issue. The downstream function block will be invoked again after bypassing the event connection. Input data will be overwritten by the bypassed function block. All the rules for event flow through basic, composite and service interface function block are identical in this case. The only difference is that to print all short events, those event names must be compared with sources and destinations in event connections. If an event connection source and destination are appeared in the result of the event flow set, it is a short event. The SQWRL query is given as:

\[
\text{EvFlow\_Transitive(\?conn, \?conn)}
\]
\[
\wedge \text{Has\_Connection\_Source(\?conn, \?source)}
\]
\[
\wedge \text{Has\_Connection\_Destination(\?conn, \?dest)}
\]
\[
\wedge \text{FBType(\?FBType1)}
\]
\[
\wedge \text{Has\_FBNetwork(\?FBType1, \?FBNetwork)}
\]
\[
\wedge \text{Has\_EventConnections(\?FBNetwork, \?conns)}
\]
\[
\wedge \text{Has\_Connection(\?conns, \?conn1)}
\]
\[
\wedge \text{Has\_Connection\_Destination(\?conn1, \?dest1)}
\]
\[
\wedge \text{Has\_Connection\_Source(\?conn1, \?source1)}
\]
\[
\text{sqwrl:makeSet(\?EV1Set1, \?source)}
\]
\[
\text{sqwrl:makeSet(\?EV1Set2, \?source1)}
\]
\[
\text{sqwrl:makeSet(\?EV2Set1, \?dest)}
\]
\[
\text{sqwrl:makeSet(\?EV2Set2, \?dest1)}
\]
\[
\text{sqwrl:intersection(\?EV1Set, \?EV1Set1, \?EV1Set2)}
\]
\[
\text{sqwrl:intersection(\?EV2Set, \?EV2Set1, \?EV2Set2)}
\]
\[
\text{sqwrl:element(\?shortsource1, \?EV1Set)}
\]
\[
\text{sqwrl:element(\?shortdest1, \?EV2Set)}
\]
\[
\text{-> sqwrl:select(\?shortsource1, \?shortdest1)}
\]

The first three lines are identical from the previous rule. All sources and destinations of event connections in this function block network are listed and intersected with the sources and destinations in the set of event flows. If duplicated sources and destinations are found, those event connections are the short events.
7.3.4 System Configuration Semantic Rules

All the rules above ensure that components constituting an FB system are semantically correct. The execution semantics of those components is also required to be checked in the system configuration.

A system configuration is semantically correct if all of the following rules are satisfied:

**Rule 1**

The source and the destination data variable in a data connection must have consistent data types;

**Rule 2**

No dangerous event loop is allowed.

**Rule 3**

No short-circuit event is allowed.

**Rule 4**

All function block instances in the function block application must be mapped to a resource.

The first three rules are the same as the composite function block rules. The SQWRL queries are also identical. The Rule 4 requires every function block instance in the function block application to be mapped to a resource. If any function block instance in the function block application is not mapped to any resource, the entire system configuration will not work properly. The complete SQWRL query for this rule is:

```sql
System(?System1)
^ Has_Mapping(?System1, ?Map1)
^ Has_Mapping_From(?Map1, ?Source1)
^ swrlb:SubstringBefore(?source1, ".", ?FBName1)
^ Has_Application(?System1, ?App1)
^ Has_SubAppNetwork(?App1, ?AppNet1)
^ Has_FB(?AppNet1, ?FBs1)
^ Has_FB_Name(?FBs1, ?FBName2)
^ sqwrl:makeSet(?EV1Set, ?FBName1)
^ sqwrl:makeSet(?EV2Set, ?FBName2)
^ sqwrl:difference(?EVSet, ?EV1Set, ?EV2Set)
-> sqwrl:select(?System1, ?EVSet)
```
IEC 61499 Semantic Analysis

The first set is made of all function block instances mapped in the system configuration. The second set consists of all FB instances declared in this system configuration. The difference between two sets indicates the missing mappings of FB instances.
7.4 Automatic Semantic Analysis Engine based on SQWRL

Unlike syntax check, semantic analysis of programs cannot be completely performed by software tools. Many modern compilers implement some semantic analysis functions. However, the methods of such analysis are usually hard coded in the compilers.

The semantic analysis of function block systems presented by their ontological models can be performed using standard ontological tools, such as Protégé [110] with the corresponding SWRL reasoners and SQWRL plugins. In this thesis a more configurable approach is proposed, when the semantic check engine (SQWRL engine + automatic semantic analysis) is able to check properties specified in the ontology model resulted from the syntax correction and semantic analysis. To fulfil semantic analysis of this ontology model, the SQWRL engine is developed. The key of the SQWRL engine is to test consistency of an ontology concept. To build the SQWRL engine for the ontology, the procedures are summarized as [111]:

1. Build a Tree-View model of the ontology concept;
2. Decompose this concept syntactically by applying the Tableau algorithms;
3. For non-deterministic rules in the Tableau algorithms, a search is necessary;
4. Ensure no clash occurs for rules and stop when no more rules applicable.

Following those principles, semantic analysis for an IEC 61499 system design starts with the SQWRL query. A description logic expression shall be built which is containing all properties of this particular ontology concept. For example, to analyse a basic function block design, the description logic expression shall be rearranged as:

\[
\text{BasicFB} \sqcap \text{FBtypeelements} \sqsupseteq 1 \text{Has\_ECC.ECC} \\
\sqsupseteq 1 \text{Has\_InternalVars.InternalVars} \sqcap \exists \text{Has\_Algorithm.Algorithm}
\]

This description logic formula indicates that a class of basic function block descriptors is a sub-class of intersection of the following four classes:

1) The class of function block type elements;
2) A class where every individual has maximum one ECC,
3) A class where every individual has maximum one internal variable list;
4) A class where every individual has some algorithms.
To complete this tree-view model, each concept involved here must be attached with its own description logic formulae. For ECC,

\[
\text{ECC} \equiv \text{FBtypeelements} \sqcap \exists \text{Has ECState.ECState} \sqcap \\
\exists \text{Has ECTransition.EC_Transition}
\]

Description logic expressions must to be passed to the semantic analysis engine (SQWRL engine). The engine will perform the automatic semantic analysis test based on the query ontology concept. SWRL built-in libraries [96] and SQWRL built-in libraries [97] syntax are added on top of the description logic (OWL). The tool also generates automatically T-box ontological models from given DTD descriptions of syntax and import XML descriptions of function block systems into A-box as instances. The tool’s screenshot is shown in Figure 7.8 in process of proving the semantic analysis of a system configuration.

![Figure 7.8: Screenshot of a system configuration semantic analysis using the developed Semantic Analysis Tool.](image)

The SQWRL capability of the tool was especially helpful in this work due to the gap in SQWRL support in Protégé. The version 3.7 of Protégé has a working SQWRL plugin, but supports only OWL 1.0. The ontology has been developed for IEC 61499 and IEC 61131-3 using OWL 2.0, which is supported only by version 4.0, which does not have SQWRL capability yet.
In the tests the queries have been completed in a fraction of second with one system configuration and ten function block definitions. However, in large IEC 61499 systems with thousands of individual nodes, the time can be substantial, so further experiments will be conducted.
7.5 Summary

The need for flexibility of the semantic analysis is motivated by several reasons, the main reasons are stated as follows.

New versions of the standard-compliant software may redefine the semantic correctness, and the corresponding semantic analysis engine would need to be upgraded;

The rules may define correctness in the context of particular design patterns. New design patterns are constantly being introduced which requires adding new semantic correctness definitions;

Last but not least, the standard itself is a living organism which undergoes maintenance, leading to certain syntax and semantics changes.

Advantages of using ontology in the semantic analysis for function block systems are quite obvious. Firstly, ontology is a formal mechanism for describing semantic properties which are originally represented in words. The formal reasoning languages and tools can be applied to check those properties. This increases the reliability of the analysis results. Also it provides a high level view of all properties and relationships of function block system design.

Secondly, the ontologies are extendable that allows describing partial knowledge that can be extended in the future. By using the description logic (DL) and Semantic Web Rule Language (SWRL) rules, the additional ontology definition can be merged into an existing ontology model seamlessly. Finally, the rules defined in DL and SWRL for ontology based semantic analysis can be modified easily without modifying the semantic analysis engine. New semantic rules can be easily defined to the semantic check system in the form of S(Q)WRL rules. This engine can be used for other XML based languages besides IEC 61499. Also it is easier to change the IEC 61499 system configuration in the ontology knowledge base compared with handling the actual function block code directly for people without programming background. It provides better IEC 61499 systems overview and clearly defines what each node can have.

The proposed approach also has a number of advantages as compared to other approaches which develop dedicated validators for each semantic rule.
First, the formal method of ontologies allows expressing precisely, clearly and at a high level many semantic properties of FB systems;

Second, formal representation of semantic properties allows usage of formal reasoning for the proof of these properties, that in turn increases reliability of the received outcomes of the analysis;

Third, by using description logic (DL) and DL-safe SWRL rules guarantees decidability of the task of the analysis that the semantic analysis should be completed in comprehensible time.

Finally, the proposed method is flexible and extensible. Semantic analysis is performed by a universal engine that imports rules from extensible knowledge base.

The proposed method can be extended to semantic analysis of any programming language, especially based on text or XML notation.
8 Case Study and Results

8.1 Case Study Example on Baggage Handling System

To illustrate the ideas of migration process in a practical manner, an airport baggage handling system (BHS) is selected as the case study as shown below.

Figure 8.1: The Case Study Baggage Handling System.

However, it is impractical to present the migration process clearly for the entire baggage handling system due to the complexity. Instead a subsystem of the case study baggage handling system is presented. The selected subsystem provides a proper balance between complexity and generality. The baggage handling systems are designed based on modules of devices which also suits modern concepts (object-oriented programming) of high-level programming languages. The physical layout of the selected subsystem of the BHS is given in Figure 8.2.
Figure 8.2: The Physical Layout of the selected Baggage Handling Subsystem.

There are five transportation conveyors (IB101 to IB105) and one baggage carousel (IB1) in the system. Bags are inducted into this subsystem from the conveyor IB101 through the transfer line and merge onto the baggage carousel IB1. There are also three emergency stops (ESTOP1, ESTOP2 and ESTOP3) installed along with the conveyors for operation safety purposes. This case study example will be applied to the redesign approaches, automatic migration process, migration between IEC 61499-Compliant tools and semantic analysis in the rest of this section.
8.2 Results from Redesign Approaches for the BHS

The baggage handling subsystem was programmed in the structure text (ST) using the RSLogix 5000 software by the Rockwell Software. The original state machine for conveyor control is provided by Glidepath Ltd [112] as a part of the standard PLC code library. The program structure is given as Figure 8.3.

![Figure 8.3: The PLC program structure of the Baggage Handling Subsystem.](image)

There are two PLC tasks scheduled in the project. The ConveyorControl task is for controlling all conveyors in the system. There are six programs associated in this task. Each program controls single conveyor. Inside the program, a conveyor control function block is invoked. The algorithm of the conveyor control function block is also shown in Figure 8.3. When this function block is invoked, the next state is determined by the current state and the transition conditions of this state. After the current state is manipulated, the related state outputs are updated by the algorithm associated with this state. In the second PLC task - FastTask, each emergency stop control station is controlled by a separate program. Also the merge control between the conveyor IB105 and the carousel IB1 is placed in the FastTask.
8.2.1 Object-Oriented Method Results

First, the object-oriented redesign pattern is applied to the inbound BHS example. The first redesign step is to convert each device (object) in the system into a basic function block in the IEC 61499 format.

A. Converted Version

The first choice is to convert the original state machine that precedes the PLC code into the execution control chart (ECC) inside the basic function block. This is a manual process. The conversion starts with creating one EC State for each state from the original finite state machine. In this case, 6 EC states are created in the new conveyor basic function block: start, startup, cascade, run, economy, and fault. Secondly, the conditions of state transitions are mapped directly to the EC transitions conditions. For example, in the PLC code, the premise for jumping from off to startup state is that there is no fault on the conveyor and the “start” button is pressed. It is set as the new EC transition condition between the EC states start and startup. Finally, the state actions are placed into the EC state algorithms. When the system is starting up, the beacon and sounder indicator will be turned on for warning. The beacon and sounder output command is lit in the startup state attached algorithm in ECC. All other EC states are created in the same order: state transitions mapped to EC state transitions and state algorithms put into the EC state algorithms. The bottom part of the Figure 8.4 shows the converted conveyor control function block with the conversion approach.

B. Reuse Version

In this method, no PLC code change is required at all, as it is encapsulated into the algorithm REQ. This is accomplished by the automatic migration engine. The new generated basic function block is a top level entity for detecting the changes of inputs when the REQ event is sent from the I/O capture Service Interface FB. When an input change happens, the basic function block will jump into the state REQ processing request. After the algorithm is executed, the new output data shall be updated. Inside the ECC, there are only two EC states - IDLE and REQ states. When the photo eye on the conveyor is flushed or any other inputs change their value, the FB wakes up from IDLE state, starts executing the algorithm in REQ state. Once the process is
Case Study and Results

completed, the FB remains in IDLE state until any other input changes. The top part of the Figure 8.4 shows the converted conveyor control function block with the reuse approach.

Figure 8.4: Conveyor control function converted in IEC 61499.
Figure 8.5 demonstrates the IEC 61499 deployment for the OO approach. Each function block instance reflects a physical device of conveyor, estop, or control panel. Those function block instances are connected in the order derived from the physical layout.

Figure 8.5: BHS Example OO Version in IEC 61499.

8.2.2 Class-Oriented Method Results

The second choice is to apply class-oriented pattern to the target inbound BHS. Similarly to the object-oriented reuse solution, all PLC code is copied into an algorithm of the new basic FB without any modification. This is also completed manually. For the conveyor control, as described in the section 3.3, in the PROCESS state of the new basic FB, conveyor control CO FB is determining the list of conveyors affected by the I/O change. Those conveyor indexes are loaded and
Case Study and Results

algorithms are repeatedly executed for each conveyor instance data. At last, all outputs are updated.

To fulfill distributed control, an IEC 61499 device is created for each subsystem which includes a class function block for each device class (or a subgroup within the class). Those class function blocks contain all data required and only be invoked when the input data change. Figure 8.6 illustrates the system overview mapping of the class-oriented inbound BHS. For instance, function block instance IBs of the type ConveyorControl contains all conveyors in the inbound system instead of individual conveyor control function blocks.

Figure 8.6: BHS Example CO Version in IEC 61499.
8.3 Automatic Migration Results of the BHS

In the previous section, a migration process is done manually according to the redesign patterns defined. Here, the migration is completed automatically by using the transformation engine developed based on the OWL and the SQWRL language. The original PLC code structure is shown as Figure 8.7. There are two tasks scheduled in the PLC. The FastTask is a periodic task which executes every 25ms. A program FastProgram is associated with this task. There is one merge control and three emergency stop controls scheduled in the program. Also there is a continuously task ConveyorControlTask which has a program ConveyorControl in the PLC configuration. Inside the program, there are six conveyor control function blocks schedule. Each of them is controlling a physical conveyor in the system. Inside all function blocks, there is no nested level of functions or function blocks invoked.

![Figure 8.7: Original PLC Configuration Structure.](image)
The resulting IEC 61499 system configuration achieved by applying the migration process is given in Figure 8.8.

Figure 8.8: Generated Function Block Configuration Structure and Main PLC Scheduler FB Algorithm.

The resulting system configuration has three resources. The resource FastTask is mapped to the periodic task FastTask. Each function block in the PLC program FastProgram is converted to an IEC 61499 function block. Those function blocks are
linked according to the order of their counterparts in the PLC program. The resource *ConveyorControlTask* includes all six conveyor control function blocks in the order of the original PLC program. A new resource Scheduling has a main PLC scheduling FB, two task scheduling FBs and two program scheduling FBs. The enable event is raised by the program scheduling FBs to either *FastTask* resource or *ConveyorControlTask* resource. Once all function blocks in the *FastTask* resource are executed, an acknowledgement event is sent back to the program scheduling FB to indicate the execution of the periodic task is accomplished. The main scheduling function block will switch to the continuously executed task until the pre-defined execution cycle for the periodic task arrives again.

This case study is supported by the automatic migration tool developed. The tool imports PLC codes in the XML format (for example, Rockwell and PLCOpen XML) and converted into the IEC 61131-3 knowledge base. Then those instances in the IEC 61131-3 knowledge base are mapped to the IEC 61499 knowledge base by the migration engine. Finally the FB version of the code is generated out. The result system configuration generated by the migration tools is compared with the original PLC project. All tasks and programs from the original PLC project are converted into the FB version correctly. The complexity of this migration process is linear to the size of the original PLC program. A 8MB PLC XML file takes approximately 5 minutes to complete the entire process.
8.4 Examples on Migration between IEC 61499-Compliant Tools

The overview of the inbound system application transformed automatically by the transformation engine is shown in Figure 8.9. The target function block code for the transformed system is compliance with the standard IEC 61499 XML definition. As seen from Figure 8.9, a FBDK can open the generated system configuration without an issue. Unfortunately, this system configuration cannot be opened by the nxtStudio as several required attributes introduced into this XML definition are missing.

Figure 8.9: The Overview of the Transformed Inbound System.
Case Study and Results

By using the same transformation engine with predefined mapping rules between two XML definitions, the FBDK version is converted to the nxtStudio version automatically as illustrated in Figure 8.10. Those compulsory XML elements and attributes which are set as required in the XML definition are inserted by the transformation engine according to the predefined mapping rules. The transformation is also capable for converting nxtStudio solutions to FBDK system configurations.

Figure 8.10: The Transformed Inbound System for nxtStudio.
8.5 Semantic Analysis Results of the BHS

The last question is how to ensure the generated IEC 61499 systems are semantically correct. The semantic analysis will be applied to all FBs used in the inbound system: basic FBs Conveyor, EStop and EStopZone, and the system configuration as well. The Conveyor FB is responsible for the basic conveyor control functionality. The Conveyor FB has a number of EC states and each state is representing an actual status of a conveyor.

![Original and Corrected ECC of Basic Function Block Conveyor Control](image)

Figure 8.11: Original and Corrected ECC of Basic Function Block Conveyor Control.

As shown in Figure 8.11, the original ECC has two issues found by the semantic analysis in the original PLC code and inherited to the function block version. First, the RUN state has connections to the CASCADE, the ECM and the FAULT state. However, the transition condition to the CASCADE state is “1”. This will cause the conveyor to start and stop all the time, so the motor can be easily damaged in a short period of time. Secondly, the FAULT state has no exit EC transition. As a result, once any conveyor is faulted, it will remain in the fault state and cause a deadlock. Those issues are resolved in the corrected version manually of conveyor control ECC that passed the semantic check.
In order to check semantic analysis rules for the BHS system configuration, a dangerous event loop is created in purpose as shown in Figure 8.12. There is only \textit{REQ} event condition and no EC guard condition for \textit{REQ} state in the ECC of both the \textit{EStop} function block and the \textit{EStopZone} function block. Once the \textit{REQ} input event of the function block \textit{ESTOP1} is triggered, this indefinite event loop will be activated. The emergency stop functionality will not perform properly. If this is not identified, it will be a serious hazard as the system is not capable to ensure the safety requirements. The full path of this dangerous event loop is highlighted in Figure 8.12. This dangerous event loop is ignored by IEC 61499 compilers.

Figure 8.12: Dangerous Event Loop Found in the Emergency Stop Control Part.
8.6 Performance Measurement

An experiment was conducted where the subsystem of selected baggage handling system extended with another subsystem which consists of 30 conveyors was measured on a centralized PLC and compared against that of a one device executing several different FB versions. Based on the results of this measurement one could derive the required characteristics of PLCs and using (3.1), (3.2) and estimate the performance of the distributed configurations. The MRT comparison between different approaches running on a centralized PLC and estimated reaction time over several PLC nodes is presented in the table below.

Table 3: Maximum reaction time comparison between PLCs and FBs.

<table>
<thead>
<tr>
<th></th>
<th>Rockwell ControlLogix L63 PLC</th>
<th>Wago 750-860 with nxtControl FB runtime Ethernet/IP Remote I/Os ²</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Object-Oriented (Manually Converted)</td>
</tr>
<tr>
<td>Centralized Configuration (TScan/TProg/Tin+Tout/TFB) ¹</td>
<td>11ms (7ms/0ms/3ms/1ms)</td>
<td>16ms (16ms/0ms/0ms/0ms)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15ms (15ms/0ms/0ms/0ms)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10ms (10ms/0ms/0ms/0ms)</td>
</tr>
<tr>
<td>BHS on 3 distributed nodes (TScan/TProg/Tin+Tout/TFB) ³</td>
<td>26ms (7ms/7ms/9ms/3ms)</td>
<td>23ms (16ms/7ms/0ms/0ms)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>22ms (15ms/7ms/0ms/0ms)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17ms (10ms/7ms/0ms/0ms)</td>
</tr>
</tbody>
</table>

¹ With example subsystem of the baggage handling system extended with another subsystem with 30 tracking conveyors.

² The scan time is sampled from the beginning of executing input module SIFBs to after all output module SIFBs have been executed with events from the process.

³ The test is using 3 distributed nodes due to limitation of hardware resources. More nodes could be added if more controllers are available.
Figure 8.13: Test results with different configuration.

As one sees from the table, the estimated MRT in the 3 nodes configuration is clearly better for all FB implementations than of the PLC version in the distributed configuration. The gap will grow linearly to the number of nodes in the network: assuming the network propagation time is the same, the scan waiting delay is triplicated for the PLC version, while no such overhead occurs in the FB version. As shown in Figure 8.13, the measured result for 1, 2 and 3 nodes is very similar to the estimated values calculated by using formulas defined in section 3.5. The measured result for 4 and 5 nodes are proving the performance gain by getting rid of scan cycles.

One should also take into account that the PLC used in the test was four times faster than the Wago CPU executing IEC 61499 FBs. We can conclude that on CPUs with similar performance, the reaction of the FB version will be substantially faster.

Therefore, preliminary performance estimations prove the expected gain of the function block implementation that comes “on top” of the improved flexibility and robustness of the system. The most substantial performance improvement is expected from elimination of access to the global database, this issue can be addressed on the runtime level in the future.
8.7 Summary

The case study presented in this chapter proves the entire migration process can be accomplished automatically by applying the ontological knowledge base, the transformation engine and the semantic analysis engine. Firstly, the original PLC codes of this baggage handling subsystem are redesigned into the function block version in both object-oriented and class-oriented approach. In the object-oriented approach, both manually converted and automatic migrated solution are illustrated. Then this example code is again converted into the function blocks automatically by the transformation engine. The object-oriented reuse approach is applied during the automatic transformation. The final migrated function block solution is very similar to the manual converted version with extra scheduling function blocks inserted by the engine. The automatic migration only takes several minutes to complete the entire process. Compared to the manual redesign process takes number of hours, the automatic migration process reduces the development time significantly into the minute level. In the next part, the interoperability between various IEC 61499-compliant tools is achieved by applying another set of mapping rules using the same transformation engine. Although the example only shows the conversion between FBDK and NxtStudio, other IEC 61499 tools are also supported like FBench, 4DIAC IDEs. The semantic analysis is performed for the generated inbound BHS and the design defeats existing in the original PLC program are found. Finally, the performance of the migrated system is measured and the results demonstrate that the IEC 61499 provides better performance with increasing number of distributed nodes compared the PLC version.
9 Conclusions and Future Work

9.1 Efficiency and Impact of the proposed migration method

The major advantages of the proposed migration method are that the entire process could be accomplished automatically and the migration process is configurable using an abstract description language. Compared to the manual migration process, this method reduces the migration time significantly from day’s and hour’s level down to minute’s level or even second’s level. Besides that, the configurable rules also reduce the time required for supporting a brand new PLC platform or changing on the code structure due to maintenance of the PLC or the FB standard.

The time required for performing an automatic and a manual migration process for the case study BHS is shown in Figure 9.1. For a migration process, there are four major tasks: converting POU, mapping instances, configure the target function block system and testing on the result function block system.

![Efficiency Estimation Graph](image)

Figure 9.1: Time Estimation for Auto and Manual Migration Process Based on Case Study in Chapter 8.
Conclusions and Future Work

The first task is to converting all reusable POUs including functions and function blocks into the IEC 61499 version. The manual process will take several hours to complete that but the automatic conversion only requires several seconds to complete. The second task is to map all elements from the PLC into the FB. Another couple of hours are needed to enter all values of PLC elements into the FB editor manually. Again the time required for mapping all elements using the automatic migration could be ignored. The third task is to configure FB systems including connect events and data variables as well as put in constant and parameter values. The manual process also takes a number of hours but the automatic migration only takes several minutes maximum. Finally, the time of testing result FB system configuration could be reduced at least half if the automatic semantic analysis is performed. When all numbers are added together, the time saved by the automatic migration is very substantial.

Next, the comparison between modifying migration rules of introducing a new PLC platform in hard-coded and configurable migration tool is illustrated in Figure 9.2.

![Modify/Create Migration Rules](image)

Figure 9.2: Time Estimation for Changing Migration Rules in Hard-Coded and Configurable Migration Engine.

The test is measured based on the changing the existing migration rules or creation of a new set of migration rules from another PLC platform in SQWRL queries and Microsoft C# code. The results demonstrate by using ontology and SQWRL rules, the time required for modifying migration rules or create new migration rules are significantly less than the hard-coded engine. This proves that by integrating this
technology into the existing IEC 61499 editing tools, large proportion of the redesign time could be saved.
Conclusions and Future Work

9.2 Conclusions
The function block architecture of IEC 61499 standard targets the system-level design of complex distributed automation systems. It proposes a new visual form of programming using block diagrams with embedded state machines and unlimited hierarchical nesting, being distributed across networking devices. The existing PLC standard is not sufficient to provide the performance on large distributed systems. Also the design process of the PLC code development for distributed systems is time consuming and complicated.

As an intermediate step of introducing the IEC 61499 standard into the existing automation industry dominated by the IEC 61131-3 standard, migration from the PLC programs to the IEC 61499 function blocks is a better solution to address the challenges of using PLCs in the complex distributed systems. Although there are several migration approaches published, none of them could perform an automatic transformation which is configurable for various types of PLCs and function block implementations. Also how to address the issue of global and local tags in the PLC is not considered in the previous works. This thesis targets at a configurable automatic migration process as well as addressing those issues which are not covered.

The methods of migration are presented in Chapter 3 and 4. In Chapter 3, several redesign patterns for the IEC 61499 standard are provided including object-oriented convert pattern, object-oriented reuse pattern and class-oriented pattern. The advantages and limitations of those patterns are summarized. The object-oriented reuse option is selected as the most appropriate pattern for the automatic migration. The formal model of the IEC 61131-3 and IEC 61499 are defined in Chapter 4. Also the complete migration mapping rules between two models are listed. The migration rules cover mapping from program structures, algorithms inside programs with different programming languages to proper scope of variables which does not exist in any published work.

Chapter 5 and 6 describe the implementation of automatic migration mechanism using ontology and semantic web technologies. The migration mapping rules defined in the methods are achieved using OWL and S(Q)WRL query language. The migration rules are defined in a template which also using the ontological knowledge base (OWL). The transformation engine built for migration takes different XML codes from various PLCs and converts to the standard IEC 61499 XML format. The
Conclusions and Future Work

generality of the transformation engine is achieved by using the configurable rules defined in the SQWRL language. The automated conversion reduces the time and the cost for the migration. This also ensures future-proof of any newer version of XML representation of PLCs or function blocks. No modification to the engine is required to suit the changes to the XML definitions. This transformation engine is able to convert between IEC 61499 implementations as well. This provides interoperability between various platform dependent IEC 61499 tools. However, the limitation of this approach is that the original PLC platform must support extraction entire project into the XML format. Also the transformation engine assumes that the instructions and built-in functions/function blocks in the original PLC platforms are available in the IEC 61499 development environments which be converted to. This automatic migration process reduces the redesign time significantly.

Chapter 7 illustrates the semantic analysis of IEC 61499 configurations. The complete IEC 61499 semantic rules are defined in the SQWRL/SWRL which are ignored by IEC 61499 compilers. The implementation of the semantic rules is built on top of the SQWRL language as well. The entire semantic analysis process is automated and configurable. This also reduces lots of debugging time and future-proof. The semantic analysis ensures the semantic correctness of the migrated IEC 61499 systems.

A case study of a relatively complicated automation handling system – airport baggage handling system is presented in Chapter 8. The migration methods proposed in the thesis are applied to the case study. The original PLC program of the baggage handling system is converted to function blocks by the transformation engine. The semantic issues in the resulting function block system are captured by the automatic semantic analysis engine. The results of the experiment on distributing the example BHS to multiple PLC and FB controllers demonstrate that with increasing number of distributed nodes in the system, the FB deployment provides a better performance.

Overall, automatic migration from scan cycle based IEC 61131-3 programs to event trigger based IEC 61499 systems provides a number of benefits that not only improving performance and reducing redesign time, but also enabling the intelligence and flexibility into the existing automation industry.
9.3 Extension to the SQWRL

Although the migration rules and the semantic rules proposed in this thesis are defined in the form of the SQWRL query, there are still some limitations of the SQWRL language which must be compensated by high-level programming languages. The first issue of the SQWRL query is that it is not possible to create, update or delete an ontology individual or property from the knowledge base. In the migration process, when the SQWRL queries of the mapping rules are executed, the result data must be written into the target knowledge base using C#. As an extension to the existing SQWRL language, three new built-in functions are proposed as following:

1. **Create a new ontology object individual in the knowledge base**

   \[
   \text{sqwrl:insert}(<\text{OntologyNodeName}>, <\text{IndividualName}>)
   \]

   \(<\text{OntologyNodeName}>\) refers to a name defined in the T-Box of this ontology; \(<\text{IndividualName}>\) represents the name of the new object individual.

2. **Create a new property to an ontology individual in the knowledge base**

   \[
   \text{sqwrl:insert}(<\text{IndividualName}>, <\text{PropertyName}>, <\text{TargetValue}>)
   \]

   \(<\text{PropertyName}>\) is the name of the object property or the data property to be inserted;

   \(<\text{TargetValue}>\) is the name of the referred object in the object property or the actual value in the data property.

3. **Update an existing ontology object individual in the knowledge base**

   \[
   \text{sqwrl:update}(<\text{OntologyNodeName}>, <\text{OldIndividualName}>, <\text{NewIndividualName}>)
   \]

   \(<\text{OldIndividualName}>\) is the original object individual name;

   \(<\text{NewIndividualName}>\) is the new object individual name.

4. **Update an existing property to an ontology individual in the knowledge base**

   \[
   \text{sqwrl:update}(<\text{IndividualName}>, <\text{PropertyName}>, <\text{OldTargetValue}>, <\text{NewTargetValue}>)
   \]

   \(<\text{OldTargetValue}>\) is the original object name for the object property or the original value for the data property;
| Conclusions and Future Work |

<NewTargetValue> is the new object name for the object property or the new value for the data property.

5. Delete an ontology individual in the knowledge base

   sqwrl:delete(<OntologyNodeName>, <IndividualName>)

6. Delete a property of an ontology individual in the knowledge base

   sqwrl:delete(<IndividualName>, <PropertyName>, <TargetValue>)

Also the normal structure of the SQWRL query is given as:

Core Operators ◦ Collection Constructors ◦ Collection Operators
-> SQWRL operands

Another issue of the SQWRL language is that the results listed in the SQWRL operands cannot be used in another SQWRL query. In the semantic analysis process, when a step requires more than one SQWRL query to complete, the results from previous query must be passed to the next query under the help of the C#. So a linkage between SQWRL queries is proposed here as:

SQWRL query 1 * SQWRL query 2 * .... SQWRL query N -> SQWRL operands

Where SQWRL query is in the form of Core Operators ◦ Collection Constructors ◦ Collection Operators.

By introduction this linkage, results from previous queries could be utilized in the next queries without any extra help.
9.4 Recommendations and Future Work

Although the automatic transformation and semantic analysis engine are configurable, the actual SQWRL query could be ended with an excessive length. To simplify the SQWRL query, a more abstract syntax could be developed. This could significantly reduce the time on defining SQWRL queries. On the other hand, the proposed SQWRL extensions mentioned in section 9.3 could be implemented and tested.

On the migration process, the mapping rules between FBD to CFB and SFC to ECC shall be implemented. Also, the existing approach is only implemented on couple of particular tool platforms although the approach is suitable for all PLC and function block platforms. Mapping rules for more IEC 61131-3 PLC platforms could be implemented in the future. Also, the migration between IEC 61499 compliant-tools could add support for more IEC 61499 tools besides NxtStudio and FBDK.

Last but not the least, the current migration process is based on the assumption of the built-in instructions and functions in original PLC platforms are available in the target function block platforms. A method of converting existing PLC built-in libraries into function block versions would be a great support for promoting the IEC 61499 standard. Finally this methodology could be extended to support other transformations rather than just IEC 61131-3 to IEC 61499 XMLs.
Appendix A
IEC 61499 Background Information

The top level entity of a function block application is a system configuration. A system configuration contains all software components including network segment configuration, devices and applications deployed in the system. As shown in Figure A.1, one or more IEC 61499 devices are connected by network segments. Several applications are deployed in those IEC 61499 devices. One feature that the IEC 61131-3 standard cannot achieve is that an application could be distributed across multiple devices.

![Figure A.1: IEC 61499 System Configuration][1]

In each function block application (FBA), there is a function block network which consists of several function blocks with connections as illustrated in Figure A.2. There are two types of connections: event connections and data connections. Function blocks are activated by events. When an event is propagated to the next function block via an event connection, the downstream function block will be executed. Data are flowed through function block network via data connections.
A function block is a module with interface as shown in Figure A.3 that consists of event and data inputs and outputs.

Events also will be further referred to as signals. A function block can be invoked only by an input event. When an input event is raised, data inputs associated with this input are updated at the same time. When the execution of this function block is completed, the related output event is emitted and all data outputs are also updated with the event.

There are three types of function blocks: basic, composite and service interface function block.

The functionality of a basic function block (Figure A.3-a) is defined as a state machine called Execution Control Chart (ECC) as stated in Error! Reference source not found..
Appendix A
IEC 61499 Background Information

The semantics of ECC is similar to Moore finite automata with actions assigned to states. An action consists of an algorithm and output event issuance (either can be omitted). An EC-transition has a condition “clocked” by no more than one event input and having a guard condition that is a predicate over data inputs and internal variables (but no events). The algorithm associated to each state could be written in any IEC 61131-3 programming languages or high level programming languages such as C++ or Java.

A Composite Function Block (Figure A.3) is specified by interface and functionality, defined as a network of function block instances interconnected via event and data connections similar to the function block application. A service interface function block (SIFB) is a platform dependent implementation. SIFB can be understood as a “black box” whose internal structure is not specified.
Appendix B
Ontology Definitions

Ontology Definition

A typical knowledge base using ontology comprises two components: a T-Box and an A-Box [95]. T-Box stands for taxonomy box which describes concepts and their general properties. A-Box or assertion box retains knowledge that is specific to individuals or instances of concepts. The T-Box is the knowledge base of all properties and relationships between concepts. All instances are modelled in A-Box. So the ontology is defined as:

$$ K = (T, A) \text{ (B.1)} $$

Where

- $K$ is the Ontology Knowledge Base;
- $T$ is the Terminological Axioms;
- $A$ is the Assertional Axioms.

Here the T-Box contains any finite set of axioms in the form:

$$ C \sqsubseteq D \text{ (R } \sqsubseteq S \text{) or } C \equiv D \text{ (R } \equiv S \text{) [95]} \text{ (B.2)} $$

Where

- $C$ and $D$ are concepts;
- $R$ and $S$ are roles.

A concept, or in another word – class, can have:

- Equivalent Classes (two concepts are the same thing)
- Super Classes (one is the subset of another)
- Inherited Anonymous Classes (one has all properties that the other has)
- Disjoint Classes (two concepts are totally different)

---

1 The notation definition of the description logic is listed at the end of this Appendix.
A concept is also linked to other elements via some roles. In terms of OWL, these elements are named as classes and these roles are called properties. There are two types of properties in the OWL ontology: Object Property and Data Property. An object property is used to describe a property value that refers to another object. Correspondingly, the data property is used when the property value refers to the actual data literal or a data type. In addition, extra information can be stored in annotation properties.

A concept $Ci$ is defined as:

$$Ci = (Di, PD_i, PO_i, PA_i, DT_i, f_i) \quad (B.3)$$

Where

- $Di$ is a set of Ontology Concept Elements related to $Ci$;
- $PD_i$ is a set of Data Properties belonging to $Ci$;
- $PO_i$ is a set of Object Properties belonging to $Ci$;
- $PA_i$ is an Annotation of $Ci$;
- $DT_i$ is a set of Data Types used by definition $Ci$;
- $f_i$ is a constructor which builds a DL expression from $Di, PD_i, PO_i, DT_i$ for the concept $Ci$.

An object property has one or more characteristics:

- Inverse Property
- Functional Property
- Inverse Functional Property
- Transitive Property
- Symmetric Property
- Asymmetric Property

If a property is functional, for a given individual, there can only be at most one individual to be related via this property. If some property links individual $A$ to individual $B$, then its inverse property will link individual $B$ to individual $A$. If a property is inverse functional, then its inverse property is functional.

If a property is transitive, and the property related individual $A$ to individual $B$, and also individual $B$ to individual $C$, then we can infer that individual $A$ is related to individual $C$ via property $P$. If a property $P$ is symmetric, and the property relates
individual A to individual B, then individual B is also related to individual A via property P. If a property P is asymmetric, and the property relates individual A to individual B, then individual B is not related to individual A via property P.

**Description Logic Notation Definition**

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>□</td>
<td>Union of concepts</td>
<td>C □ D</td>
</tr>
<tr>
<td>▼</td>
<td>Intersection of concepts</td>
<td>C ▼ D</td>
</tr>
<tr>
<td>※</td>
<td>Concept equivalence</td>
<td>C ※ D</td>
</tr>
<tr>
<td>⊆</td>
<td>Concept inclusion</td>
<td>C ⊆ D</td>
</tr>
<tr>
<td>∃</td>
<td>Existential restriction</td>
<td>∃R.C</td>
</tr>
<tr>
<td>¬</td>
<td>Complement</td>
<td>¬C</td>
</tr>
</tbody>
</table>

Where

- C and D are ontology concepts
- R is the restriction condition.
Appendix C
Publications based on this Thesis

Published Papers:

(1) W. Dai, V. Vyatkin, “A Case Study on Migration from IEC 61131 PLC to IEC 61499 Function Block Control”, 7th IEEE International Conference on Industrial Informatics (INDIN 2009), Page 79 - 84.

(2) W. Dai, V. Vyatkin, “On migration from PLCs to IEC 61499: Addressing the data handling issues”, 8th IEEE International Conference on Industrial Informatics (INDIN 2010), Page 1142 – 1147.


(8) W. Dai, V. Vyatkin, “Transformation from PLC to Distributed Control using Ontology Mapping”, 10th IEEE International Conference on Industrial Informatics (INDIN 2012), Accepted.

Unpublished/Submitted Papers:


References

References


References


References


[81] D. Gibson, "Finite State Machines - Making simple work of complex functions." SPLat Controls Pty Ltd.


References


[96] SWRL: A Semantic Web Rule Language Combining OWL and RuleML. Available: http://www.w3g.org/Submission/SWRL/


