Reliable Model-driven Engineering using IEC 61499

Gareth Darcy Shaw
February 2013

Supervisors: Partha S Roop
Zoran Salcic

Department of Electrical and Computer Engineering
The University of Auckland
New Zealand

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Engineering
The University of Auckland

Thesis Consent Form

This thesis may be consulted for the purpose of research or private study provided that due acknowledgement is made where appropriate and that the author's permission is obtained before any material from the thesis is published.

I agree that the University of Auckland Library may make a copy of this thesis for supply to the collection of another prescribed library on request from that Library; and

1. I agree that this thesis may be photocopied for supply to any person in accordance with the provisions of Section 56 of the Copyright Act 1994.

Or

2. This thesis may not be photocopied other than to supply a copy for the collection of another prescribed library.

(Strike out 1 or 2)

Signed: .................................

Date: .................................

Created: 5 July 2001
Last updated: 9 August 2001
Abstract

Complex industrial control systems, such as those used in airport baggage handling systems (BHSs), need to be designed in a way to ensure that they avoid failures. Even small failures in such systems may result in serious economic and social consequences due to delayed flights and lost bags. Consider, for example, the problems caused by failures in the BHS during the opening of a new terminal at Heathrow airport in 2008. The complexity of BHSs has also grown over the years, as exemplified by the BHS of Schiphol airport, which can handle up to 70 million bags in a year.

Model driven engineering (MDE) is a modern design approach that is being touted as the next generation design methodology to deal with design complexity. MDE focuses on the specification of graphical and high-level models, often based on sound mathematical principles, from which low-level implementations are automatically derived. The use of automated code generators makes systems developed using MDE tools and techniques inherently platform and network agnostic. For safety-critical applications, designers also have access to many high-level verification and validation (V & V) techniques to validate the correctness of designs. MDE is already being used to provide robust designs, while meeting time to market pressures. MDE is the main theme of this thesis, and its usage will be illustrated in the design of complex control systems such as airport BHSs.

Unlike the MDE approach, industrial control systems are traditionally designed using low-level languages such as the IEC 61131-3 standard. These languages, while suitable for small to medium scale control systems, are inherently more difficult to design, deploy and maintain compared to MDE-based approaches. IEC 61499 is a recent open standard that is proposed to facilitate a component-based graphical approach for the design of industrial control systems. This thesis adopts IEC 61499 as the main basis for an MDE-based development approach. However, the standard lacks rigorous mathematical semantics that is the cornerstone of MDE. Hence, this thesis makes two key contributions, as outlined below:

(a) A denotational, compositional semantics for IEC 61499 function blocks is proposed. The proposed semantics is based on the well known synchronous languages and is proposed to remove any semantic ambiguity.
(b) Some syntactic sugar has been developed to allow Statechart-like hierarchy and concurrency to define the behaviour of a basic function block. This is done by defining a new language, similar to Statecharts, called hierarchical and concurrent ECCs (HCECCs). HCECCs extend the conventional execution control charts (ECC) as used in the standard. Importantly, HCECCs can be automatically transformed to a standards compliant form using the developed semantics.

Thus, this thesis, for the first time proposes a graphical synchronous language that combines the block-diagram based control/data-flow notation of IEC 61499 with the elegance of Statecharts to create a new MDE framework for complex control system design. In order to illustrate the efficacy of the proposed approach, a compiler has been developed, and this compiler is integrated into two commercial integrated development environments (IDEs). Subsequently, these IDEs have been used to design a small scale BHS in collaboration with a local BHS company. This case study, and several other benchmarks illustrate the effectiveness of the proposed approach.
List Of Publications

The following is a list of published research articles based on the work developed by this thesis:


Acknowledgements

This thesis was made possible thanks to collaboration and assistance from the following people:

- My supervisor Dr. Partha S. Roop has been an excellent guide for my research. He has been continually supportive and an invaluable source of ideas and information.

- My co-supervisor Prof. Zoran Salcic provided essential input guiding the work presented in this thesis.

- Dr. Valeriy Vyatkin also guided aspects of my work, thanks to his expertise in IEC 61499 and industrial automation.

- Dr. Li Hsien Yoong was a valuable resource and friend over the course of my PhD. I was originally the guinea pig for his compiler and after that we worked together on many things. He was able to offer a lot of experience for the implementation of my own semantics and compiler.

- Dr. Roopak Sinha, helped me with many discussions on the semantics presented in this thesis. He was also a valuable resource for all things mathematics, such that the simplicity of the algorithms in chapter 4 owe much to his contributions.

- I would also like to thank Cheng Pang, Zeeshan Bhatti and other friends and colleagues in my office and research group.

- Roz Stevens helped me immensely by proofreading this thesis, for which I am very grateful.

Finally, I wish to thank my family, friends and my fiancée Becky Stevens for their support and understanding.
# Contents

Abstract iii

List Of Publications v

Acknowledgements vii

1 Introduction 1

1.1 Industrial Practice 2

1.1.1 Drawbacks 3

1.2 Model-Driven Engineering 3

1.2.1 Tools and Languages for MDE 4

1.2.2 IEC 61499 for System Design 5

1.3 Research Objectives 7

1.4 Thesis Contributions 8

1.5 Thesis Organization 10

2 Literature Review 11

2.1 Limitations of IEC 61499 11

2.1.1 Ambiguities in IEC 61499 12

2.1.2 Deficiencies in System Specification 14

2.1.3 Uncertain Migration of Legacy Code 15

2.1.4 Availability of Execution Platforms 15

2.1.5 Usability of Design Tools 16

2.1.6 Untested in industry 17

2.1.7 Summary 17

2.2 IEC 61499 Execution 17

2.2.1 IEC 61499 Execution Semantics 17

2.2.2 Formalisms for IEC 61499 Semantics 20

2.2.3 Discussion of IEC 61499 Semantics 21

2.3 Synchronous Function Blocks 21
2.3.1 Synchronous Semantics of IEC 61499 .......................... 22
2.3.2 Comparison with Event-Triggered Semantics ......................... 24
2.3.3 Discussion of Synchronous Semantics .................................... 26
2.4 Enhancing ECCs for reactive systems ................................. 27
  2.4.1 Graphical Languages ........................................ 27
  2.4.2 Existing Approaches ......................................... 28
  2.4.3 Discussion ...................................................... 30
2.5 Migrating From IEC 61131-3 to IEC 61499 .......................... 30
  2.5.1 Existing Approaches ......................................... 31
  2.5.2 Discussion ...................................................... 32
2.6 Remarks .......................................................... 33

3 An IEC 61499 Case Study ................................................. 35
  3.1 Development of a BHS ........................................... 36
    3.1.1 Model-View-Controller Design .................................. 37
    3.1.2 A Conveyor Section ....................................... 37
    3.1.3 Application Configuration .................................. 43
    3.1.4 The Visualisation .......................................... 44
  3.2 Discussion ........................................................ 45
    3.2.1 Advantages of IEC 61499 .................................. 45
    3.2.2 Deficiencies ............................................... 46

4 Hierarchical and Concurrent ECCs for IEC 61499 Function Blocks .... 47
  4.1 Formalisation of IEC 61499 ...................................... 48
    4.1.1 Function Block Interfaces .................................. 49
    4.1.2 Basic Function Block ....................................... 50
    4.1.3 Composite Function Block .................................. 54
  4.2 Hierarchical and Concurrent ECCs .................................. 60
    4.2.1 Parallel HCECCs ........................................... 62
    4.2.2 Refined HCECCs ............................................ 73
    4.2.3 Interleaving of parallel and refinement operators .......... 79
  4.3 Discussion ........................................................ 83
    4.3.1 Comparison of HCECCs with Related work .................... 83
  4.4 Conclusions ........................................................ 84

5 Re-Engineering IEC 61131-3 into IEC 61499 Function Blocks .......... 87
  5.1 IEC 61131-3 specifications ...................................... 88
    5.1.1 Design of a Baggage Handling System ......................... 89
  5.2 Proposed Mapping of IEC 61131-3 to IEC 61499 ..................... 91
5.2.1 Global Variables in IEC 61499 ................................................. 93
5.3 Proposed Approach ............................................................... 95
  5.3.1 Function Block Development ................................................. 97
  5.3.2 PLC Parsing ........................................................................ 101
  5.3.3 Code generation .................................................................... 103
  5.3.4 Summary ............................................................................. 108
5.4 Results ..................................................................................... 108
  5.4.1 Visualisation ....................................................................... 109
5.5 Discussion ............................................................................... 111

6 Compilation of IEC 61499 Function Blocks .................................. 113
  6.1 Compilation to C ...................................................................... 115
    6.1.1 Code generated by the Original FBC ...................................... 116
    6.1.2 Further FBC Development ................................................... 125
    6.1.3 Compiler Performance Results .............................................. 131
  6.2 Compilation for Rockwell PLCs ............................................... 136
    6.2.1 Synchronous IEC 61499 to PLC Semantics ............................ 136
    6.2.2 Task Allocation of Blocks ..................................................... 137
    6.2.3 Generating UDTs for Function Blocks .................................... 138
    6.2.4 Translating Block Networks .................................................. 139
    6.2.5 Translating ECCs ................................................................. 141
  6.3 Discussion ............................................................................... 145

7 Development of IEC 61499 Tools ............................................... 147
  7.1 Integration with nxtStudio ......................................................... 148
    7.1.1 Plugin Development .............................................................. 149
    7.1.2 Compiler Development ........................................................ 150
    7.1.3 FBC and nxtStudio Summary ............................................... 157
  7.2 TimeMe .................................................................................. 157
    7.2.1 Integration of FBC ............................................................... 157
    7.2.2 Implementation of Simulation Mode ...................................... 159
    7.2.3 TimeMe Summary ................................................................. 167

8 Conclusions .............................................................................. 169
  8.1 Hierarchical and Concurrent ECCs for IEC 61499 ...................... 170
    8.1.1 Summary and Contributions ................................................ 170
    8.1.2 Future Work ...................................................................... 170
  8.2 Reverse Engineering IEC 61131-3 into IEC 61499 ...................... 171
    8.2.1 Summary and Contributions ................................................ 171
8.2.2 Future Work .................................................. 172
8.3 IEC 61499 Code Generation ................................. 172
  8.3.1 Summary and Contributions ............................ 172
  8.3.2 Future Work ................................................ 173
8.4 IEC 61499 Development Tools .............................. 174
  8.4.1 Summary and Contributions ............................ 174
  8.4.2 Future Work ................................................ 174
8.5 Concluding Remarks .......................................... 175

References ......................................................... 175
List of Figures

1.1 Common Design flow for Industrial Projects ........................................... 2
2.1 Demonstration of Event Life-cycle ambiguities .......................................... 13
2.2 Illustration of ambiguous block scheduling ................................................ 14
2.3 Synchronous reaction or *tick* ................................................................... 22
2.4 Non-causal connections when using the synchronous approach .................... 23
2.5 ECC for modelling the movement of the conveyor belt .................................... 25
3.1 Layout of the prototype baggage handling system ........................................ 36
3.2 Illustration of Model-View-Controller Design ............................................. 37
3.3 Components of a conveyor ............................................................................ 38
3.4 Interface for the Conveyor_Model Composite Block ..................................... 39
3.5 Network of function blocks which model a conveyor section ......................... 39
3.6 ECC for the ConvBeltModel block ............................................................... 41
3.7 Service interface function blocks to connect the plant model to the controller ... 42
3.8 Part of the Controller’s ECC to control the motor ........................................ 43
3.9 System Configuration for BHS .................................................................... 44
3.10 Screenshot of 2D BHS Visualisation created using nxtStudio [1] .................. 45
4.1 Interface for the Conveyor_Model Composite Block ..................................... 49
4.2 Interface for the Conveyor_Photoeyes_Model Basic Function Block ............. 50
4.3 ECC which simulates the infra-red *PhotoEye* sensor .................................... 51
4.4 Network of blocks within the Conveyor Model composite function block ....... 55
4.5 Illustration showing the execution of Conveyor Model composite function block .................................................................................................................. 58
4.6 HCECC specification of the Conveyor Model .............................................. 61
4.7 Overview of HCECC translation ..................................................................... 62
4.8 A sample parallel HCECC ............................................................................ 63
4.9 Parallel ECCs flattened to a function block network ..................................... 65
4.10 Modification of Parallel ECCs ..................................................................... 69
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.11</td>
<td>HCECC with a state refined by two parallel ECCs</td>
<td>74</td>
</tr>
<tr>
<td>4.12</td>
<td>Parallel ECCs equivalent to refinement in Figure 4.11</td>
<td>75</td>
</tr>
<tr>
<td>4.13</td>
<td>HCECC with multiple levels of refinement</td>
<td>81</td>
</tr>
<tr>
<td>4.14</td>
<td>HCECC with nested refinement removed</td>
<td>81</td>
</tr>
<tr>
<td>4.15</td>
<td>Equivalent HCECC with only parallel ECCs</td>
<td>83</td>
</tr>
<tr>
<td>5.1</td>
<td>IEC 61131-3 Overview</td>
<td>89</td>
</tr>
<tr>
<td>5.2</td>
<td>Timing Diagram Illustrating Execution Schedule for two tasks</td>
<td>90</td>
</tr>
<tr>
<td>5.3</td>
<td>LLD sample including complex Branch behaviour</td>
<td>91</td>
</tr>
<tr>
<td>5.4</td>
<td>Illustration of Option #1</td>
<td>93</td>
</tr>
<tr>
<td>5.5</td>
<td>Illustration of Option #2</td>
<td>94</td>
</tr>
<tr>
<td>5.6</td>
<td>Migration plan from IEC 61131-3 to IEC 61499</td>
<td>96</td>
</tr>
<tr>
<td>5.7</td>
<td>Conveyor Controller Sub-component Network</td>
<td>99</td>
</tr>
<tr>
<td>5.8</td>
<td>PhotoEyeController ECC</td>
<td>100</td>
</tr>
<tr>
<td>5.9</td>
<td>Section of a State-machine written in LLD</td>
<td>100</td>
</tr>
<tr>
<td>5.10</td>
<td>Conveyor Controller HCECC</td>
<td>101</td>
</tr>
<tr>
<td>5.11</td>
<td>Branch in LLD</td>
<td>105</td>
</tr>
<tr>
<td>5.12</td>
<td>Screenshots from MHVIS</td>
<td>110</td>
</tr>
<tr>
<td>6.1</td>
<td>Visualisation of example BHS</td>
<td>114</td>
</tr>
<tr>
<td>6.2</td>
<td>System Architecture for BHS example</td>
<td>115</td>
</tr>
<tr>
<td>6.3</td>
<td>Network of Block Modelling Conveyor Section</td>
<td>121</td>
</tr>
<tr>
<td>6.4</td>
<td>ECC for the conveyormodel_Photoeys Block</td>
<td>124</td>
</tr>
<tr>
<td>6.5</td>
<td>HCECC for Photo-Eye Simulator</td>
<td>127</td>
</tr>
<tr>
<td>6.6</td>
<td>Illustration of Adapters</td>
<td>129</td>
</tr>
<tr>
<td>6.7</td>
<td>Demonstration of Adapters</td>
<td>130</td>
</tr>
<tr>
<td>6.8</td>
<td>Composite block with nested Adapter ports</td>
<td>131</td>
</tr>
<tr>
<td>6.9</td>
<td>Test-bench Execution times</td>
<td>133</td>
</tr>
<tr>
<td>6.10</td>
<td>Normalised benchmark execution times compared to FBC-HCECC</td>
<td>133</td>
</tr>
<tr>
<td>6.11</td>
<td>Test-bench Object Code Size</td>
<td>134</td>
</tr>
<tr>
<td>6.12</td>
<td>Normalised code size compared to FBC-HCECC</td>
<td>134</td>
</tr>
<tr>
<td>6.13</td>
<td>Translation of IEC 61499 into IEC 61131-3</td>
<td>138</td>
</tr>
<tr>
<td>6.14</td>
<td>Encoder State to Encoder Count Block</td>
<td>139</td>
</tr>
<tr>
<td>6.15</td>
<td>Section of Conveyor Controller Network</td>
<td>141</td>
</tr>
<tr>
<td>6.16</td>
<td>ECC for Encoder State to Encoder Count</td>
<td>142</td>
</tr>
<tr>
<td>6.17</td>
<td>LLD for Encoder State to Encoder Count ECC</td>
<td>144</td>
</tr>
<tr>
<td>7.1</td>
<td>nxtStudio Plugin Interface for the FBC Synchronous Compiler</td>
<td>149</td>
</tr>
<tr>
<td>7.2</td>
<td>Graphical Representation of a Dynamic Service Interface Function Block</td>
<td>150</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

7.3 Human-Machine-Interface design elements in nxtStudio .................. 152
7.4 Architecture of HMI Communication ........................................ 153
7.5 HMI Visualisation of a simple Baggage Handling System ................. 154
7.6 Redistributing some Conveyors within the Baggage Handling System example 155
7.7 Communication Blocks Automatically Inserted by nxtStudio ............. 155
7.8 Architecture of Inter-Resource Communication .............................. 156
7.9 TimeMe Project Templates ..................................................... 158
7.10 TimeMe Configuration Selection ............................................. 158
7.11 Code Generation Error Messages shown in TimeMe ....................... 159
7.12 TimeMe Simulation Sequence ............................................... 160
7.13 Conveyor_Model block used for Simulation ................................ 160
7.14 TimeMe Simulation User Interface for Conveyor_Model .................. 164
7.15 TimeMe Simulation Control Interface ...................................... 164
7.16 TimeMe Simulation: HCECC in ConveyorModel_HCECC ................. 165
7.17 TimeMe Simulation: Network in Conveyor_Model .......................... 165
7.18 TimeMe Tick Response XML Tree ........................................... 166
7.19 TimeMe Tick Stack View ..................................................... 166
7.20 Implementation of Function Block Simulation Application ............... 167
## List of Tables

1.1 Tools for Model-Driven Engineering ................................................. 4

2.1 Comparison of Event-triggered and Synchronous Execution ................. 24

2.2 Comparison of Specification Languages ......................................... 29

4.1 CFB Equivalence of Parallel HCECC semantics ............................ 72

4.2 Parallel HCECC Equivalence of Refined HCECC semantics ............... 80

5.1 IEC 61499 implementation of IEC 61131-3 elements ....................... 92

6.1 HCECC and ECC Size Comparison ................................................. 132

6.2 Equivalent PLC Components for IEC 61499 .................................. 137
The design of industrial automation and embedded systems is becoming more difficult due to the demand for larger systems with increasingly complicated mechatronic components. Similarly, the decreasing cost of computational power is encouraging manufacturers to implement more functionality with software [2]. Many domains are also moving to distributed and heterogeneous platforms instead of the traditional approach based on centralised monolithic controllers. However, the majority of industrial control systems are still developed using a bottom-up approach with low-level languages.

Demand for systems with complexities beyond the capability of development tools is a significant challenge faced by modern software developers [3]. This thesis therefore aims to facilitate the development of complex and reliable industrial control systems. Materials handling systems are used as a case study throughout this thesis as they include complex routing functionality with high reliability requirements.

This chapter is structured as follows: Section 1.1 explains the current industrial practice for the development of materials handling systems, identifying the short-comings of the approach. Section 1.2 presents a modern alternative approach, including a brief discussion of the languages and available tools. Section 1.3 outlines the objectives of this thesis, and Section 1.4 presents the contributions of this work. Finally, Section 1.5 describes the organisation of this thesis.
1.1 Industrial Practice

Figure 1.1 shows an example work-flow for the development of a materials handling system (MHS), beginning with the computer aided design of a system for a client. Hardware and software development occur in parallel, linked by knowledge from previous work and the specific goals of the current project. The testing of software within the software development phase is comprised of rudimentary functionality tests for various algorithms. Software development then waits for the hardware to be constructed, before more detailed testing can be accomplished. Software testing with the physical system, commonly referred to as acceptance testing, allows the developers to ensure that the software behaves according to the specification. A suite of tests is iterated over the completed system, and if errors are discovered, the software is modified and then re-tested. The critical path of this approach, coloured in red, is due to the construction of the hardware. Because of the limited testing features of the approach, final handover is delayed for several weeks while the software is more accurately tested on the physical system.

![Diagram of Design Flow](image)

**Figure 1.1: Common Design flow for Industrial Projects**

As with many industrial domains, software for a MHS is developed using the IEC 61131-3 [4] set of languages and deployed to a PLC (programmable logic controller). The IEC 61131-3 standard, first released in 1993, defines an architecture for controller specification and incorporates a set of older languages which can be used to define the control logic. The standard was widely adopted by vendors of industrial control hardware as the languages themselves were already in use prior to the standard. However, inter-operability of IEC 61131-3 between PLC manufacturers has been hampered by proprietary implementations of each of the languages, restricting developers to a single device manufacturer.
1.2 Model-Driven Engineering

1.1.1 Drawbacks

The bottom-up approach used by IEC 61131-3 languages has been found to be inadequate to meet new demands for reconfigurability and reuse [5, 6]. The maintenance and reuse of control logic for a component is also complicated by the lack of object oriented mechanisms. As a result, the code for a single component is spread through multiple routines which may overlap with other components. The development of a model of the physical plant is also difficult because of the lack of separation from control logic and the limited computational resources available on PLCs. As a result, the controller is developed as an open-loop component, which delays full testing and debugging until the hardware for the system is constructed.

The inadequacies of the development tools and approaches have contributed to several notable and costly failures in materials handling systems. The scheduled opening of a new airport in Denver, Colorado in 1995 was delayed by almost 2 years due to significant problems with the baggage handling system [7]. The recent opening of Heathrow’s Terminal 5 in 2008 also serves to motivate a more reliable software development approach. Upon opening, the terminal was plagued with significant problems related to software issues with the new baggage handling system. The result was the cancellation of 300 flights and the mishandling of thousands of bags, costing an estimated $16 million [8, 9].

As systems grow in complexity and size, the distribution of control applications to multiple controllers becomes necessary. However, current usage of the outdated IEC 61131-3 standard is reducing the productivity of industrial manufacturers and producing less reliable software than more modern techniques and languages. These flaws and deficiencies are motivating manufacturers to investigate approaches and technologies that will allow shorter time to market with increased customisation. The next section presents Model-Driven Engineering as an ideal approach for industrial automation and embedded systems.

1.2 Model-Driven Engineering

Model driven engineering (MDE) [10] is widely used in the software engineering domain to tackle software complexity while also ensuring a high quality implementation. In this approach, abstract and component oriented software models are developed to capture and describe the system requirements. From this high-level abstract design, lower level implementation details are iteratively introduced to refine system behaviour. A key feature is the use of generators and transformation engines to synthesize executable code or create alternative model representations for formal analysis. Automated code generation is used to deploy the implementation over a range of target platforms. Designers then have the option of generating either centralised or distributed solutions from the same models.
For industry, model-driven engineering simplifies design by dividing a system into its components, which can then be independently modelled. After development, these abstract components can then be easily reused in different systems. The automated translation of models creates an error-free implementation from a specification and provides additional design benefits such as formal analysis. If the model-view-controller (MVC) [11] approach is also employed, in which the hardware under control is also modelled and visualised, the controller can be visually tested with a simulated system. This enables complete functional testing of the system before construction of the hardware, reducing the time to market.

1.2.1 Tools and Languages for MDE

A number of commercial tools support model-driven engineering, such as Simulink [12] from MathWorks, SCADE Suite [13] from Esterel Technologies and Rational Suite [14] from IBM. In addition, the IEC 61499 open standard [15] is well suited to model-driven engineering, using component-oriented function blocks for the specification of distributed control systems. The standard has already garnered significant interest from researchers, and support and developments from manufacturers of industrial products is increasing, with two commercial integrated development environments already supporting the standard [1, 16].

Table 1.1 compares the identified modelling tools and the IEC 61499 standard. Both Simulink and SCADE are based on proprietary standards for model description, where as IEC 61499 and UML are open standards. The use of a standard is beneficial for customers, as multiple manufacturers can develop their own tools or hardware platforms creating a richer development platform. All of the surveyed tools are capable of generating C code, which can then be deployed to a range of platforms that support C compilation. Simulink also supports the Verilog and VHDL hardware description languages for integrated circuit design. Because they are not proprietary, it is possible for a third party to develop additional code generators for UML and IEC 61499 specifications.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Implementation</th>
<th>Code generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulink [12]</td>
<td>Block Diagrams using MatLAB or C</td>
<td>C, C++, Verilog, VHDL</td>
</tr>
<tr>
<td>SCADE [13]</td>
<td>Lustre and Safe State Machines</td>
<td>C</td>
</tr>
</tbody>
</table>

Table 1.1: Tools for Model-Driven Engineering

Simulink and SCADE can be used for safety-critical design, as both are capable of capturing requirements and verifying models. Both tools generate code that is compliant with functional safety standards such as IEC 61508 [17]. In contrast, the open UML and
IEC 61499 standards offer a generic framework for a range of distributed applications, with similar specification and code generation capabilities.

To develop reliable systems, hardware in the loop testing enables accurate behavioural testing prior to deployment. In addition to the specification of the controller, this requires the development of a simulatable model of the physical system. UML is too abstract to be able to accurately model hardware behaviours [18]. IEC 61499 and SCADE are both capable of such modelling, however, Simulink simplifies such models with libraries of pre-built functions to describe physical behaviour.

Overall, the IEC 61499 standard offers top-down and cross-platform development, avoiding the existing use of platform specific and proprietary tools. While there are currently no formal analysis tools, these are not required for most industrial applications. In addition, because the new standard was created as a replacement for IEC 61131-3, industrial acceptance is likely. What follows is a more detailed description of the standard, a summary of the motivations for using it, and the status of the standard and available tools.

1.2.2 IEC 61499 for System Design

The IEC 61499 or function block standard, presents a new modelling language based on event-driven function blocks for the specification of component-oriented distributed systems. Developed to meet the requirements of intelligent automation using model driven engineering, it uses graphical function blocks as its basic design entities. The standard encourages top-down application design, using abstract function blocks to encapsulate system components. Using this standard it is possible to define a complete system, including the distribution of components on to different execution devices.

To describe system behaviour, the IEC 61499 standard defines three different kinds of function blocks, with a consistent interface, that allow the designer to describe different levels of component behaviour:

- **Basic function blocks** - At the lowest structural level, basic function blocks contain an Execution Control Chart (ECC) graphical state-machine for control flow design and textual algorithms for data handling.

- **Composite function blocks** - These blocks encapsulate a network of other function blocks into a single interface. Blocks of any type can be instantiated in the network. Thus, structural hierarchy can be described by inserting a composite block inside the network of another block.

- **Service interface function blocks** - These provide an interface to device or hardware specific components. The execution of the block is determined at compile time,
where the code generated for such a block binds the function block application to a specific hardware target.

Graphical ECCs remove the need to manually implement a state-machine in sequential code. They also abstract data management into an algorithm call within a state, separating data and control flow. Function block algorithms are used to describe data computations, and can be defined in any language - including the IEC 61131-3 languages. At a higher level of abstraction, function block networks, within composite function blocks, describe the flow of events and data between other function blocks.

The highest level of an IEC 61499 specification describes a complete system of devices, which are an abstraction for an execution platform. A device executes multiple resources, each of which consists of a function block network. This enables tools to simplify redistribution of applications, merely by moving a function block to a resource on another device and introducing blocks for communication.

**Motivations for the use of IEC 61499**

The desire of industrial manufacturers to move to a new approach is inevitable, and to IEC 61499 specifically is most likely, due to its relation to its predecessor. IEC 61499 has also received a lot of interest from research groups because it can be used to define distributed component-oriented systems in a platform independent way. More recently, there have been some developments from industrial automation companies in support of IEC 61499. This includes several published case studies on the use of IEC 61499 for industrial projects, and new commercial development tools and execution platforms.

The graphical notation of ECCs and function block interfaces is appealing as they describe the behaviour of a system in an intuitive and self documenting style. The introduction of a component-oriented architecture allows for a complete model-view-controller design approach. By developing a model of the controlled component, the testable scenarios are greatly improved and thus higher quality software can be developed. In addition, by providing an abstract execution semantics, function block applications are afforded portability between execution platforms. This portability also allows developers to test the functional behaviour of applications on a PC before deployment. Thus, with better testing before deployment and much easier reuse of components, there can be a much shorter time-to-market.

Another advantage of IEC 61499 is its ability to alter the distribution of a function block application without changing any component logic. By enabling a centralised or distributed approach from the same specification, function blocks very easily accommodate designs spread over greater physical distances and designs that require more processing power for each component. Because of the focus on component-oriented distributable
1.3 Research Objectives

designs, function blocks are well suited for the description of intelligent machines [5] - inter-connectible mechatronic components with attached controllers. The development of intelligent machines thereby allows new systems to be created by connecting existing components together in a new configuration.

For distributed systems, or applications with high levels of reuse, the benefits of IEC 61149 over IEC 61131-3 are significant. Model-driven engineering, supported by IEC 61499, serves to simplify design and simultaneously allows a model to be reused in different projects and on different execution platforms.

Status of IEC 61499

There have been several published case studies on the use of IEC 61499 in industry. In [19, 20] Brusaferri et al. describe the development of a modular footwear factory, where different shoes are made by following different routes on the same network of machines. The authors note that their design approach based upon IEC 61499 reduces development effort, and produces a more reliable system suitable for reuse in other projects. Example applications from Chouinard et al. [21] and Yan et al. [22] demonstrate the feasibility of IEC 61499 for large distributed systems.

Two commercial integrated development environments (IDEs) are available for IEC 61499: ISaGRAF [16] and nxtStudio [1]. ISaGRAF, recently purchased by industrial automation company Rockwell [23], is an IDE for IEC 61131-3, with support for IEC 61499. Future versions of ISaGRAF will address compatibility issues with the standard and improve the usability of various aspects of function block design. The nxtStudio IDE from nxtControl [24] includes a unique feature which allows the seamless design and integration of visual components into IEC 61499 applications. Both tools allow code to be deployed to industrial controllers from Beckhoff, WAGO, SIEMENS and Advantech.

Unfortunately, as the following section presents, aspects of the standard are inadequate for the development of reliable software. In addition, other languages offer superior features for the description of some behaviours.

1.3 Research Objectives

The goal of this work is to facilitate the development of complex and reliable industrial control software using IEC 61499. To achieve this, the following aspects of the standard will be investigated:

- **Semantics** - The execution semantics of IEC 61499 are known to be ambiguous [25, 26], leading to many different implementations. Most approaches use a run-time for execution, leading to poor performance and the potential for non-deterministic
behaviour. A deterministic semantics that can be efficiently implemented is required for applications where reliability is important.

- **Language Improvements** - ECCs used in basic function blocks are too simplistic for use in complex applications. Other state-machines, notably Statecharts [27], offer notations which allow developers to succinctly describe complex behaviours. Thus, an alternative to ECCs needs to be investigated, ensuring that it is compatible with the standard to allow a portable specification.

- **Code generation** - Efficient and portable code must be generated from function block specifications in order to be useful for automation and embedded applications. Instead of generating code that relies on a run-time, code that can be directly executed will reduce the performance overhead and memory usage. Code generators will be required to support the new semantics, creating executable code suitable for use on a range of industrial and embedded controllers.

- **Development Tools** - At present, there are only two commercial development environments for the standard. However, both nxtStudio and ISaGRAF are coupled with their own proprietary function block run-times. New tools will therefore be required to simplify the use of language improvements, a new semantics and associated code generators. A single development environment that supports these additions to the standard will aid the design of complex and reliable specifications with the function block standard.

### 1.4 Thesis Contributions

This thesis addresses issues identified in the IEC 61499 standard, which hinder its potential for reliable software design. These issues are discussed in detail in the next chapter. Although materials handling systems are used to motivate and demonstrate this work, the development and approaches presented in this thesis are applicable to a range of industries. By employing ideas in the fields of modelling languages and code translation and generation, this thesis produces the following contributions:

1. A *hierarchical and concurrent extension for IEC 61499 ECCs based on the synchronous approach* [28]. In Chapter 4, this thesis introduces Hierarchical and Concurrent ECCs (HCECCs), which extend the state-machines within IEC 61499. HCECCs provide hierarchy and concurrency at the state-machine level, improving the capability of function blocks to succinctly describe complex behaviour.

The synchronous semantics given to these extensions were also developed, defined as *denotational semantic rules* [29], simplifying their description and ensuring com-
1.4 Thesis Contributions

positionality. The extended semantics also guarantee that any program constructed with the semantics is deterministic.

2. Semi-automated migration of IEC 61131-3 applications into IEC 61499 using a code translator. An approach encouraging the re-engineering of existing IEC 61131-3 applications is presented in Chapter 5. A tool for the automated translation of routines is developed, allowing legacy code to be migrated into standard C code. By automating this step and describing a technique for the creation of the function block architecture of system components, industrial manufacturers can more easily migrate their specifications into IEC 61499.

3. Two compilers for generating code from IEC 61499 applications for two types of platforms. Chapter 6 presents the two separate compilers, which produce directly executable code for function block applications based on the extended synchronous semantics developed in Chapter 4. The first compiler is based on the FBC compiler from Yoong et al. [30], which generates C code from IEC 61499 applications. FBC is developed further to support the synchronous semantics of HCECCs presented in Chapter 4. Support for various aspects of the standard, including algorithms written using IEC 61131-3 Structured Text and function blocks using adapters, which were not originally compiled, are also added to improve the utility of the compiler. To evaluate the compiler, a series of benchmarks were also created to compare it with other IEC 61499 execution approaches.

The second compiler generates IEC 61131-3 Ladder Logic Diagrams (LLDs) from IEC 61499 specifications, including function blocks using HCECCs. As industrial manufacturers prefer the proven reliability of PLCs over embedded processors, compilation to LLD allows IEC 61499 to be used for design before deploying to a PLC.

4. Integration of the FBC compiler into two integrated development environments for an enhanced user design experience. Chapter 7 addresses the shortcomings of IEC 61499 design tools using two different approaches. Firstly, the FBC compiler presented in Chapter 6 is integrated into nxtStudio to allow applications to be executed without a run-time on any target which supports C code. This provides a developer with the usability of nxtStudio, whilst compiling highly portable and efficient code. Secondly, FBC is developed further as the dedicated compiler for a new IDE for IEC 61499 called TimeMe. This provides a dedicated IDE for the developed semantics, which in turn also allows for formal analysis tools such as verification and timing analysis to also be tightly integrated. This is especially important as existing IDEs use run-times for deployment, thus users may design their applications without being cognizant of the synchronous execution semantics used by the compiler.
1.5 Thesis Organization

The remainder of this thesis is divided into seven chapters. Chapter 2 discusses related work, identifying approaches for formalising and improving IEC 61499 as well as migration from IEC 61131-3. The synchronous semantics for IEC 61499, developed by Yoong et al [31], is also introduced to further motivate the extended semantics in Chapter 4. Chapter 3 describes a case study of the development of a baggage handling system using IEC 61499. Chapter 4 introduces HCECCs, simplifying design by allowing state-machine hierarchy and concurrency within an IEC 61499 basic function block.

Chapter 5 presents an approach for the re-engineering of IEC 61131-3 configurations into IEC 61499 applications. The automated code translation tool is described, and the complete approach is demonstrated using an existing baggage handling system controller developed using Ladder Logic Diagrams.

Two compilers are developed in Chapter 6 for the generation of executable code from an IEC 61499 application. The two tools allow a user to generate code from a single application for two different types of platforms. Chapter 7 presents the integration of the improved FBC compiler into two IDEs for improved usability. Finally, the conclusions of this work are discussed in Chapter 8, and possible avenues for future research are identified.
In this chapter, existing work on the IEC 61499 standard will be reviewed. Focusing on reliable software development, attention is brought to work that proposes a formalism for IEC 61499 execution semantics. Further to this, existing approaches which address the ineffectiveness of ECCs for the design of complex and safety critical applications are examined. Finally, the migration of legacy IEC 61131-3 applications into IEC 61499 is examined, in the hopes of bridging the standards and encouraging industry adoption.

Section 2.1 identifies some aspects of the IEC 61499 standard that are hindering its usefulness and delaying industrial adoption. Section 2.2 then discusses some of the proposed formalisms for the standard, which attempt to address issues regarding to the execution of function blocks. Section 2.3 provides more detail on the synchronous semantics for IEC 61499 presented by Yoong et al. [31]. Section 2.4 evaluates and compares ECCs with other graphical state-machine languages. Section 2.5 introduces existing approaches towards migrating work from IEC 61131-3 specifications. Section 2.6 ends this chapter with some concluding remarks.

2.1 Limitations of IEC 61499

As a modern model-driven language, IEC 61499 allows the development of easily redistributable components. However, existing tools and aspects of the standard itself lack features required by industry to develop reliable control systems. The following list
describes features of the standard that require further development:

- **Standards Compliance** - A known issue with the 2005 release of the IEC 61499 standard [15] is the ambiguous nature in which the execution of function blocks is described.

- **Simpler specifications** - A complex component must be easier to define in IEC 61499 than the current industrial practice.

- **Migration process** - The migration to IEC 61499 must cause minimal interference to production.

- **Target platforms** - IEC 61499 must be deployable to platforms used by a manufacturer, including likely future platforms.

- **Design Tools** - Commercial quality development environments are required to ensure the usability of IEC 61499.

- **Industry proven** - To demonstrate its effectiveness, IEC 61499 needs to be demonstrated in large scale projects.

The following sub-sections elaborate on these issues, which later sections in this chapter will address.

### 2.1.1 Ambiguities in IEC 61499

The most vital issue hindering IEC 61499 is the ambiguous description of execution semantics in the standard. This allows for a potential repeat of the IEC 61131-3 languages, where competing manufacturers implemented different and incompatible semantics. The ambiguities in the function block standard are well documented [25, 26], and have already resulted in incompatibilities between implementations.

The most significant discrepancies arise due to an incomplete model of computation:

1. *The lack of a notion of time.* The standard does not explicitly define the *life-span* of an event, i.e., the period of execution that an emitted event should be considered *present*. However it does specify that no two events can be simultaneously present at the interface for a block. As a consequence, ECC transition evaluations can be interpreted ambiguously, producing different behaviour from the same specification.

2. *Ambiguous composition of function block networks.* The standard does not define the combined behaviour of a network of blocks. Depending on the approach chosen, the implementation may result in well known concurrency issues such as *race conditions* or *starvation.*
2.1 Limitations of IEC 61499

Ambiguous Notion of Time

The ECC fragment in Figure 2.1(a) demonstrates the impact of the ambiguous life-cycle event. Assuming that the transition COND1 has a higher priority than condition EI1&COND2, if the event EI1 is present and both boolean conditions are true, the final behaviour is unclear. If the life-cycle of the event EI1 is just for one transition, then the ECC will stop in STATE3. Alternatively, if EI1 is still present, and COND2 is still true, then the ECC will stop in STATE4.

Figure 2.1(b) shows an ECC fragment with two transitions dependent on event EI1. Due to the lack of specificity, it is unclear whether a single emission of EI1 can trigger both transitions in one run.

Activation Order of Function Blocks

In the function block network in Figure 2.2, after the execution of block B1 and the emission of EO1 and EO2, both B2 and B3 must be executed. From the standard it is not clear which block should be executed first, or if they can be executed concurrently. The ambiguity is further compounded by event feed-backs such as from EO3 to EI1, where the use of ill-defined execution policies can lead to:

- race conditions - If B1 is triggered by the output from B2 before it finishes its execution.

- starvation - B3 may never be executed if the loop between B1 and B2 continuously reactivate each other.
Impact of Ambiguities

Current execution approaches vary in their implementation of these features, leading to a fragmentation in application compatibility [32]. As an alternative, Yoong et al. [33, 34] present an approach which attempts to mitigate the ambiguities in the standard by providing an execution semantics which guarantees all function block applications are deterministic and deadlock free.

2.1.2 Deficiencies in System Specification

While IEC 61499 offers many features beyond the capabilities of IEC 61131-3 languages, some deficiencies have been identified by industry and researchers. ECCs are ineffective for describing complex behaviours, as aspects such as concurrency and structural hierarchy are absent - despite being present in IEC 61131-3 Sequential Function Charts (SFCs). Exception handling behaviour is particularly difficult [35], resulting in complicated state-machines unlike other graphical state-machine languages such as UML Statecharts [36].

A function block network is also inadequate for complex components which share data between multiple behaviours. Using multiple function blocks for such a component introduces performance overheads when passing data between blocks and complicates synchronising data values between blocks. Alternatively, the use of a single ECC for the component requires that all of the behaviours of a component are described in a single state-machine, making it difficult for users to distinguish particular behaviours.

Chapter 3 describes the modelling of a baggage handling system using IEC 61499 function blocks, illustrating the benefits and issues in the standard. Section 2.4 also explores alternatives to ECCs, identifying potentially useful features missing from the standard.

Global variables are often used in existing control applications because they can sim-
2.1 Limitations of IEC 61499

plify and speed up certain tasks. However, global variables are not directly implementable in IEC 61499 because they are incompatible with component-based development, complicating their re-use and distribution. Section 5.2.1 includes some discussion of global variable equivalence in the function block framework.

2.1.3 Uncertain Migration of Legacy Code

The adoption of any new language or tool is most commonly hindered by the desire to maintain previous development efforts. In the common case of materials handling systems, this refers to controller specifications using IEC 61131-3 languages. Starting the development of a new controller from scratch in a new language would be independent of previous work and could potentially introduce a series of new bugs to the controllers. Further, while the standard ensures that IEC 61131-3 configurations can also be described using event triggered function blocks, features such as global variables must be implemented less efficiently in function blocks.

Christensen, one of the main proponents of the standard, notes in [37] that "several aspects of this standard are unfamiliar to most practitioners of control systems engineering, especially the ideas of distributed applications, event driven execution control and service interface function blocks...". The difficulties encountered when introducing IEC 61499 to industry are common to any new software discipline. For existing IEC 61131-3 developers, adoption of the function block approach to design represents a significant paradigm shift, as multiple devices can be modelled in a single IEC 61499 system. Further, the use of graphical event-triggered function blocks, while similar to IEC 61131-3 Function Block Diagrams (FBDs), offers a significant change to component modelling. It is clear, however, that the advantages of IEC 61499, especially with respect to a shortened time-to-market, offer significant motivation for migration. In addition, in [35], Vyatkin notes the ease with which new users learn to use this standard.

2.1.4 Availability of Execution Platforms

IEC 61499 enables the design of an abstract specification that can be implemented on any controller with sufficient processing resources. In order to achieve this, however, run-times or code-generators must be developed for the desired hardware targets.

In the long term, it is envisaged that IEC 61499 will encourage distributed intelligent machines [5], where each hardware component has a small dedicated controller of its own. These intelligent machines can then be connected together in arbitrary arrangements to implement easily reconfigurable behaviours. In the more immediate term, it is likely that centralised control will be favoured until more distributed systems are proven to be reliable. Of interest to industry is the continued use of PLCs as a proven execution
platform, whilst using IEC 61499 to improve designs [38]. This will allow a progressive change of software tools followed by hardware. In addition, the existing run-time execution approaches have had minimal testing to persuade industry of their reliability.

2.1.5 Usability of Design Tools

As the IEC 61499 standard was first published in 2004, there has been little time for commercial quality development environments to be produced.

Function Block Development Kit (FBDK) is a mature tool from Holobloc Inc. [39]. It is considered by many to be the reference implementation of IEC 61499 tools for research [38]. FBDK supports most aspects of the standard, and can execute applications using the included Function Block Run-Time (FBRT). However, the interface is not up to the standard of commercial software, and the run-time is slow and inefficient.

Commercial function block development tools such as nxtStudio [1] and ISaGRAF [16] aid the development process, introducing usability functions for development, debugging and deployment.

The purchase of the ISaGRAF by Rockwell Automation Inc., a leading industrial automation company, illustrates the appeal of IEC 61499 for future automation systems. Version 5 of the software was the first to support the IEC 61499 standard, basing the implementation on IEC 61131-3 Function Block Diagrams. ISaGRAF executes function block applications using its own cyclic run-time, which is not compliant with the standard [32]. By cyclically executing function blocks in a function block network, blocks are not event-triggered, and the user is required to define the execution order. Further, as IEC 61499 function blocks in ISaGRAF are based on IEC 61131-3 function blocks, instead of the Execution Control Chart state-machine defined in the IEC 61499 standard, ISaGRAF uses an IEC 61131-3 Sequential Function Chart (SFC). These differences lead to different execution behaviours compared with other run-times [32].

nxtStudio nxtStudio is a dedicated development environment for IEC 61499 from nxtControl [24], which includes a customised run-time (nxtForte) for function block execution based on FORTE [40]. It offers several usability features, such as remote deployment of applications to various platforms and a unique feature which integrates component visualisation with function block design. Unfortunately, the run-time environment is one of the slowest and it requires the highest memory usage of those tested in Chapter 6.

Industrial adoption of IEC 61499 is unlikely until a useful development environment is developed [38]. From a design perspective, nxtStudio offers the most significant usability enhancements to aid and simplify development. However, the use of a run-time for deployment by both commercial tools limits the potential deployment platforms and introduces performance and memory overheads.
2.1.6 Untested in industry

Two years after the 2005 ratification of the IEC 61499 standard, the authors of [41] discussed four factors possibly contributing to the slow adoption of the standard by the major control system vendors. The four factors identified the aspects of function block implementations which were not yet demonstrated: maintainability, scalability, extensibility and predictability of execution. Other work since the publication of [41] have addressed the concerns regarding scalability, such as the large scale examples presented in [21, 22]. The large data-intensive baggage handling system presented in this work also demonstrates function blocks usability and effectiveness. Extensibility refers to reuse and future extension of function block applications, particularly with regard to zero downtime re-deployment. This area is explored in [42, 43], which demonstrate the use of IEC 61499 for such applications. Predictability of execution is investigated in several works including the run-time approach proposed in [44] and the synchronous approach originally proposed in [34].

2.1.7 Summary

Regardless of these deficiencies, IEC 61499 offers significant advantages over bottom-up development with IEC 61131-3 or C code. Platform independence, and a focus on reusable components allows more flexible designs to be developed more quickly. The ability to describe the behaviour of the hardware also allows off-line testing of control logic much earlier in the development process. Optional integration with an application to visualise the system further improves the development process, and provides a useful tool for run-time monitoring. Finally, a tool chain that allows manufacturers to migrate their existing code base into function blocks will play an important role in motivating adoption.

2.2 IEC 61499 Execution

This section explains the execution of IEC 61499 basic and composite blocks and provides an overview of execution approaches. Proposals of a formal execution semantics for the standard are also compared to identify a suitable groundwork for the development of reliable distributed control systems.

2.2.1 IEC 61499 Execution Semantics

The event-triggered semantics of the standard requires the dynamic activation of function blocks on the presence of an event. When a basic function block is activated, the embedded
ECC performs a function block run, during which transitions are evaluated and taken until no further transitions are enabled. Algorithms are executed and events are emitted whenever a state is entered during the run. When a composite function block is enabled by an event, block instances in its network that are connected to the event are also activated.

The most common method of executing function block applications is to use a run-time to implement inter-block semantics, compiling each function block into a new class that can be instantiated and executed. Free run-times such as FBRT [39] and FORTE [40] and commercial run-time environments such as nxtForte [45] and ISaGRAF [16] are available for a range of industrial controllers, as well as standard PCs. A recent alternative approach, presented in [33], proposes a semantics which enables more efficient direct execution of function block applications without a run-time.

Run-time Approaches

IEC 61499 run-times are responsible for managing event propagation and the activation and scheduling of blocks. Individual function blocks and function block applications are compiled into objects which are then managed by the run-time they are deployed to.

The Function Block Run Time (FBRT) [39], included with Function Block Development Kit (FBDK), is considered to be the reference implementation for researchers and developers. FBDK’s function block compiler generates Java code [46] from any function block type, which can then be executed in FBRT. FBRT faithfully implements an event-triggered approach for invoking function blocks, by compiling code which directly calls a function from connected blocks when an event is emitted. This produces a depth-first method of event propagation. While simple, this requires a large stack to implement long chains of event connections. By directly following the standard with regard to event-triggered activation of blocks, FBRT only evaluates data-only transitions once on entry to a state. As a result, if a state only has data-only exiting transitions, if none are true when the state completes its actions, the ECC will not re-activate, even after the data values have changed.

FORTE [40] provides an open-source event-triggered run-time, which has also been adapted and modified into the commercial nxtForte [45] run-time. FORTE features an event dispatcher, which uses a first-in-first-out (FIFO) queue to pass events between blocks. This produces a breadth-first method of event propagation. Unlike depth-first, this de-couples the execution of the emitting block from the receiving block, removing any blocking period during the execution of a single block. The 4DIAC-IDE generates C++ code for FORTE, removing the need for a virtual-machine as used by Java. However, the compiled C++ run-time is still slow and bulky as noted in [30].

In [47] Lastra et al. present an execution semantics for function blocks using a scan-cycle based approach: assigning inputs, executing the logic and then writing outputs in
every cycle. This approach is adopted by the commercial run-time ISaGRAF as explained in [32]. Unlike other approaches, ISaGRAF executes function blocks using a cyclic approach. In this cyclic run-time, function blocks are executed in every scan-cycle regardless of the presence of input events. Emitted output events can be received by other blocks in the same cycle if they are executed after the emitting block, otherwise they are received in the next scan-cycle. In this way, unlike the standard, the behaviour of an application is dependent on the execution order of function blocks in the network. Cyclic execution also gives rise to the possibility for simultaneous events, a scenario not explicitly covered by the standard. In fact, because simultaneous events are not possible in the event-triggered standard, transitions in an ECC can only be dependent on at most one event. As a result, instead of ECCs, ISaGRAF diverges from the standard by using IEC 61131-3 Sequential Function Charts (SFCs) for state-machine design. This also allows specifications in ISaGRAF to react to the absence of events, which is not possible in the standard.

A key disadvantage to run-time approaches is the overhead of event and block management, resulting in sluggish performance and a larger memory footprint. The buffering of events also creates the possibility of event loss due to buffer overflow. The event-triggered run-times may also create non-deterministic behaviours because of the dynamic scheduling of events and block execution. The cyclic model of ISaGRAF, on the other hand, does not accurately follow the the event-triggered standard, forcing the user to decide on an execution order for blocks. Generally however, it is possible for cyclic execution to behave as in the event-triggered semantics. Finally, all run-times need to be developed and compiled for each target hardware platform before a function block application can be executed. This limits users to either: (1) waiting for commercial run-times nxtFORTE and ISaGRAF to be developed for the required platform, or (2) porting the open-source FORTE run-time themselves.

Direct Execution

In [33, 34] Yoong et al. present a method for the direct execution of function blocks without a run-time, by adopting the synchronous approach [28]. The execution of function block applications is similar to the cyclic execution method, but with a formal semantics for the composition of blocks in a network, removing the dependence on block execution order. As a result, this approach has significant advantages over run-time approaches:

• **Predictable temporal properties** – The execution order of function blocks is completely known at compile-time, thus, removing the need for an event-based scheduler during execution.

• **Deterministic behaviour** – The behaviour of function block designs is determined solely by the semantics without needing to consider their interaction with a run-
time environment. Subtle variations in run-time environment implementations have already been shown to result in very different application behaviour [26, 48].

- **Code efficiency** – The clear semantics facilitate the generation of fast and directly executable code, avoiding the overhead of run-time environments.

- **Many potential platforms** – Possible targets are only restricted by the availability of suitable code generators and compilers.

The semantics from Yoong et al. enable the generation of deterministic code which can be directly executed without a run-time environment. Benchmarks have also shown significant performance and memory benefits from this approach [30].

### 2.2.2 Formalisms for IEC 61499 Semantics

There has only recently been some progress towards the adoption of formal execution semantics for the IEC 61499 standard [25, 33]. Formal semantics are attractive because they mitigate ambiguities and allow the usage of other proven tools and techniques such as verification and validation, not directly possible with the standard. Most IEC 61499 formalisms utilise alternative languages to model function blocks, making basic and composite function blocks a sub-set of another language. Proposals of formal execution semantics for IEC 61499 are also reviewed.

Vyatkin et al. discuss a semantics for IEC 61499 based on a sequential hypothesis in [49]. The approach requires a run-time to schedule block activation based on a First-In-First-Out ordering of event emissions. This can be implemented in a single thread to provide predictable execution and potentially more efficient implementations. Using the approach, it is also possible to customise the run-time scheduler to handle real-time constraints.

The µCrons [44] run-time, developed by Zotl et al., adopted the sequential approach to develop a IEC 61499 run-time for real-time requirements. Zotl et al. discuss the behaviour and semantics of the run-time, investigating a deterministic execution of event-triggered function blocks based on real-time scheduling theory [50]. The authors propose a run-time scheduler that schedules the execution of blocks based on their concept of an Event-Chain derived from blocks that are sources of events. An Event-Chain is defined as the sequential series of function blocks that are executed starting with a block that is an Event-Source. Each event-chain can then be given real-time constraints such as a deadline. Any block that is executed by multiple Event-Chains must disable other Event-Chains from executing, or the Event-Chain tasks must be scheduled in such a way that they do not interfere.
2.3 Synchronous Function Blocks

A drawback of the approach for a designer is that for event connections from a single emitter to multiple blocks, it is more intuitive to assume simultaneous invocation of both blocks. The sequential execution is also awkward for the distributed standard. More significantly, the sequential execution means that the compositional behaviour of block networks cannot be studied and hence verification is more difficult.

Earlier work from Vyatkine et al. use Net Condition/Event Systems (NCES) to formalise the execution of function blocks [51, 52]. NCES, based on Petri Nets, use places and transitions to represent conditions and events respectively. To formalise a basic function block, NCES modules are created for each state, action and transition, requiring data to be abstracted. This translation requires many more places and transitions than states and conditions in the function block implementation. Verification is then possible from the NCES model, but the explosion of states suggests the approach is not scalable and data abstraction makes verification less usable.

The synchronous semantics for IEC 61499 developed by Yoong et al. [53] proposes a different execution paradigm based the synchronous tick. For a basic function block, the evaluation of a transition and entering a new state is defined to be a tick. Blocks in a network are considered to be in parallel, where the parallelism is purely logical and each block is executed once per tick. Verification is then possible by generating Esterel [54] from a function block application, as shown in [33]. The semantics allow for direct execution of arbitrary function block applications, and benchmarks demonstrate the superior performance [30].

2.2.3 Discussion of IEC 61499 Semantics

The synchronous semantics from Yoong et al. provide a significant benefit to the user, allowing complete verification of an application via Esterel as well as direct execution. These semantics were chosen as the basis for much of the work in this thesis for this reason. The next section provides more detail on the semantics as well as a comparison with the event-triggered standard.

2.3 Synchronous Function Blocks

The synchronous approach is based on the synchrony hypothesis, which assumes that the reaction time of the system is sufficiently faster than the arrival time of new inputs from the environment. This assumption allows for an elegant simplification of system development. It provides a precise notion of time by dividing execution into a set of discrete logical instants called ticks. A tick, illustrated in Fig. 2.3, is identical to the PLC scan-cycle, in which: inputs are read, computation is performed and outputs are written.
The notion of concurrency among blocks in a network is logical, allowing the execution of a network of blocks to be compiled into a sequential function. Then, like a typical PLC scan-cycle, inputs are sampled at the start of every tick, the function is called, and outputs are emitted. The determinism provided by the synchronous approach leads to efficient formal verification of safety properties using observers [55].

![Figure 2.3: Synchronous reaction or tick](image)

Introducing the synchronous approach to function block execution provides many benefits over the event-triggered execution model [34], such as:

- **determinism** – The precise behaviour of a component can be known prior to run-time.

- **reactivity** – Components will always be capable of reacting to valid inputs, that is they will never deadlock in a state.

This section provides an overview of the synchronous semantics for function blocks developed by Yoong et al. and discusses the implications of using such an approach for the event-driven standard.

### 2.3.1 Synchronous Semantics of IEC 61499

The synchronous semantics for function blocks, presented most recently in [30], defines the execution of each function block type as a reactive function. The work offers a formal semantics for the execution of arbitrary function block applications, instead of the various run-time implementations of the informal semantics described in the IEC 61499 standard. The semantics of function block interfaces, function block networks and ECCs are summarised below.

The function block interface, common across all blocks, retains the behaviour described in the standard. Including the behaviour of event-data associations, where the block
contains a buffered copy of all data ports, which are updated whenever an associated event is present.

The semantics of function block networks includes both composite blocks and IEC 61499 resources. They are summarised as follows:

- Within a function block network, all function blocks are composed using synchronous parallel, and hence executed in lock-step.

- Connections within a network copy events and values from one block interface to another.

As networks of function blocks are in synchronous parallel, arbitrary connections between function blocks can lead to non-causal behaviour. Thus function block networks require analysis to find a valid order for execution. Earlier work from Yoong et al. [33, 34] dealt with this by taking the approach used by the synchronous language SL, simply forcing all inter-block communication to be delayed until the next tick. More recent work in [30], presents a topological sort to detect non-causal connection loops, and either reject them or insert a delay in one of the connections to break the cycle.

![Figure 2.4: Non-causal connections when using the synchronous approach.](image-url)

For example, the network of function blocks in Figure 2.4 has a non-causal loop. Arbitrarily forcing the connection from B2 to B1 to use the value from the previous tick, allows the following reactive function:

1. Read inputs for function block B1.
2. Execute B1.
3. Read inputs for B2.
5. Read inputs for B3.


7. Write any outputs to the interface.

This reactive function is then executed in every \textit{tick}, activating all function blocks regardless of the presence or absence of events.

The semantics of an ECC are summarised below:

- In a \textit{tick}, transitions are evaluated and taken, the new state is entered, and it's actions are executed.

- An event emission is present for a single tick.

- Transitions are evaluated in the order they are declared in the textual syntax of a basic function block.

2.3.2 Comparison with Event-Triggered Semantics

Table 2.1 compares properties of both event-triggered and synchronous execution approaches. The primary difference is the concept of time, where the event-triggered semantics remove all notions of measurable time. In contrast, the synchronous approach divides time into logical time periods or \textit{ticks}, allowing for a more detailed execution description. When an input event from the environment occurs, the event-triggered approach can immediately start reacting by activating the appropriate block. As such, blocks are in-active by default, reducing processor usage until dynamically activated at run-time. However, this creates a lot of execution overhead from event buffering and scheduling blocks for execution. Within the synchronous implementation, all blocks in the system are always active reading inputs and reacting, even when there is no change to the inputs. Therefore, there is a maximum amount of initial latency, followed by a comparatively fast reaction time.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Event-Triggered</th>
<th>Synchronous Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Implicit</td>
<td>Explicit Logical Time</td>
</tr>
<tr>
<td>Block Activation</td>
<td>Inactive by default</td>
<td>Always active</td>
</tr>
<tr>
<td>Function Block Execution</td>
<td>Intelligent Scheduling</td>
<td>Causal Composition</td>
</tr>
<tr>
<td>Event Buffering</td>
<td>Provided by a run-time</td>
<td>Unit-delayed Connections</td>
</tr>
<tr>
<td>Execution</td>
<td>Run-time</td>
<td>Direct</td>
</tr>
</tbody>
</table>

Table 2.1: Comparison of Event-triggered and Synchronous Execution

The scheduling of function block execution is dynamic in the event-triggered approach, as opposed to static with the synchronous approach. Using Figure 2.4 as an example, the
2.3 Synchronous Function Blocks

run-time for the event-triggered approach must dynamically schedule blocks for execution, for example using depth-first or breadth-first traversal, to propagate events in the correct order. In this case, naïve depth-first scheduling would both starve block B3, and create a race-condition where block B1 is reactivated by block B2 before it finishes execution. The synchronous approach on the other hand requires semantics or causal analysis to statically order block execution. Causal analysis of a system identifies non-causal behaviour, where one event will ultimately be the cause of itself. An example of this is the connection loop between blocks B1 and B2, where B1 activates B2 and B2 activates B1. Execution of one block before the other, such as B1 before B2, blurs the distinction between inputs and outputs, as the outputs from B2 will be read before they are written. These can then be handled by forcing one of the connections in the loop to be Unit-Delayed, that is forcing the input to use the output value from the previous tick. As scheduling is done at compile time, this results in a much faster and deterministic implementation with the synchronous approach. Finally, as the synchronous execution does not use a run-time it has a smaller memory footprint and faster reaction time to new inputs.

Figure 2.5 shows the ECC for the function block which simulates the movement of a conveyor belt. The order of outgoing transitions from each state is determined by their order in the textual representation (XML) of the block. Hence, even in cases when two or more transitions can be triggered, the function block deterministically chooses the one which appears first. When a new state is entered, its associated actions are performed. The algorithms are executed and output events are emitted sequentially in the order of the textual representation.

When in the START state, the execution of a single tick proceeds as follows:

1. Evaluate exiting transitions. If none evaluate to true, then stop until the start of the next tick.

2. If the INIT event is true, go to state INIT.

3. Enter state and execute algorithm INIT and emit event INITO.

![Figure 2.5: ECC for modelling the movement of the conveyor belt](image-url)
In the next tick, the ECC will be in the INIT state, and the always true (1) transition will be taken. As emitted events are only present for a single tick, an event will be ignored if it is present in the next tick. Thus, for the synchronous approach, ECCs must test for input events in out-going transitions for every state. This is in contrast to run-time approaches, which execute a function block run whenever an event is received. Execution of a function block run would instead transition into the INIT state and then back to the START state in one activation. Any subsequent events would be buffered by the run-time and the ECC would be executed again. A further complication is that the synchronous approach allows multiple events to be present at the interface for a function block simultaneously. Because an ECC transition can only react to a single event, the event monitored by the lower priority transition will be missed.

2.3.3 Discussion of Synchronous Semantics

The main advantage of synchronous approach comes from guaranteed determinism and the ability to execute a function block application directly without a run-time. The formal semantics of synchronous function blocks overcome the ambiguities plaguing the standard, and introduces efficient and deterministic execution of function blocks. The approach is compliant to the standard, even though it does not use a function block run. This is explained in [56], which notes the IEC 61499 standard does not require exact implementation of ECC execution. Instead only the function equivalent is required, specifically not even necessarily following the concept of a function block run.

The drawbacks of synchronous function blocks arise from the use of ECCs, which were not designed with the synchronous approach in mind. Within the synchronous approach, the behaviour of an ECC is unintuitive and restrictive due to the limit on single events per transition. This limitation means that it is not possible to implement behaviour that reacts to multiple events if two or more are present simultaneously.

Desirable features from other synchronous languages such as state-machine concurrency and refinement are not explored by Yoong et al [34]. State refinement would allow the top down design of behaviour, and introduce a mechanism for prioritising and abortion of the sub-processes in a component. Concurrent state-machines can be used to improve maintainability by the graphical division of related though independent sub-processes. These sub-processes can then be described separately and succinctly in the same state-machine. These features could be introduced by adopting an existing synchronous language such as Statecharts [27] for state-machine design. However, this would diverge from the IEC 61499 standard.
2.4 Enhancing ECCs for reactive systems

The features of the IEC 61499 standard were defined to aid the development, distribution and maintenance of distributed systems. However, the ECC state-machine developed and prescribed by the standard lacks many features that would improve the expressiveness of the IEC 61499 standard. In [35], Vyatkin notes the general acceptance, by both researchers and practitioners, of the use of ECCs to describe communicating distributed systems. However, Vyatkin also notes the difficulties encountered when using ECCs to handle exception conditions. Other languages such as Statecharts [27] have hierarchical states which simplify exception handling behaviour. In contrast, ECCs end up cluttered with multiple extra transitions to and from the exception handling state.

As discussed earlier, ECCs are also restrictive to the point of being unusable for the specification of behaviour within the synchronous approach. This is predominantly due to the potential for simultaneous events, which is not possible in the event-triggered standard. Regardless of ECC design, events monitored by lower priority transitions will still be missed. ECC specifications are also forced to be more complicated as every state needs an exiting transition to monitor every event.

The 2004 draft of the IEC 61499 standard [57] suggested UML Statecharts as a potential replacement for ECCs to improve the design of state-machines. This section will discuss features of other graphical languages to identify the deficiencies of IEC 61499 ECCs. In particular, graphical synchronous languages will be compared with ECCs, as this thesis will focus on the synchronous execution of IEC 61499 due to its guaranteed determinism and absence of deadlocks. Synchronous languages are introduced to illustrate some useful concepts that maybe adapted into the standard’s ECC state-machine.

2.4.1 Graphical Languages

A graphical language provides features for the succinct specification of complex behaviours. The languages, in particular languages based on Statecharts, offer hierarchy and parallelism in a graphical state-machine that allow the user to refine and separate behaviours. For the synchronous execution of IEC 61499, the adoption of such a language would alleviate the issues inherent in using an event-triggered state-machine for synchronous design.

Statecharts are a class graphical state-machines first defined by Harel in [27]. The original Statecharts from Harel introduced parallelism and hierarchy to simplify the description of behaviour. Since then, Statecharts have become more common, including its usage within the UML standard [58], where it is used to model reactive control systems. Parallelism allows for a succinct representation of concurrent control behaviour. While hierarchy is used for the standard software engineering idea of refinement. In addition,
hierarchy simplifies the modelling of pre-emption and priority, which are two key components of reactive programming languages. Recent synchronous languages, such as Argos and SyncCharts [59] (which extends Esterel [54]), adopt these features and create different semantics for different features. For instance, Argos is deterministic and compositional, unlike Harel's original semantics.

Drusinsky et al. discuss the benefits of Statecharts over traditional finite state machines in [60]. They note the unsuitability of traditional state-based approach without concurrency and multi-level nesting for describing complex components, stating the deficiencies as almost universally accepted and inherent limitations.

However, the semantics of these languages are not compatible with the IEC 61499 standard. Instead, a new compatible semantics is required to allow their use. The following sub-section presents some approaches which are compliant to the standard. These languages are then compared to motivate a new synchronous semantics for IEC 61499 that is compatible with the standard.

2.4.2 Existing Approaches

In [61], Riedl et al. propose the use and translation of SFCs into function block networks with library blocks to aid with the semantic differences. The existing IEC 61131-3 SFC graphical language is much more expressive than ECCs for state-machine design. IEC 61131-3 SFCs support hierarchy using a macro-state in which more detailed SFCs can be drawn and parallelism is created using simultaneous branches, in which all branches are executed. For compatibility, the SFC is translated into a function block network, requiring additional library blocks to implement the different semantics.

In [62] by Barji et al. provide a comparison of Petri-net based C-Net, IEC 61499 function blocks and UML Statecharts using a variety of specification criteria. For this work, the comparison will be restricted just ECCs and alternative state-machines, instead of the entire set of specification elements. ECCs, in contrast to C-Net and UML Statecharts, do not support modular design with hierarchy or concurrency. While C-Net diagrams and UML Statecharts are able to describe synchronisation, as ECCs are flat state machines, synchronisation instead occurs between basic function blocks. C-Net and UML Statechart diagrams also provide mechanisms to realize mutual exclusion. In the context of an ECC, mutual exclusion is not useful. Instead, function blocks in an function block network can synchronise based on event and data communication. Further, the syntax of a function block network enforces that no two blocks can ever write to the same variable.

In [63] Hagge et al. directly compare the modelling features of C-Net diagrams and IEC 61499 function blocks. C-Net combines the modelling of control and data flow into a single consistent language. This has the advantage of visualising data flow, unlike the abstracted algorithms used by ECCs. C-Net also avoids errors related to inconsistent
data, as events and data are inseparably combined unlike event-data associations in IEC 61499.

Barji et al. also observe that there are a large number of design elements in IEC 61499 compared to C-Net and UML Statecharts, requiring much more understanding from a designer to use effectively. However, IEC 61499 is much more than just a framework for the description of a system, as it includes elements to describe the distribution and communication of hardware devices. The authors also describe IEC 61499 as an incomplete graphical language due to the textual design of algorithms. This classical textual description of data flow is however more common in other languages, reducing the learning curve for IEC 61499. It is also possible to describe algorithms in any language, including graphical languages such as SFCs.

Hagge, in [64], proposes merging of C-Net diagrams and IEC 61499, where the more versatile C-Net can be used in the place of ECCs. Simple adapters can be used within the C-Net model to communicate to and from the function block interface. The resulting function block and C-Net hybrid model is then no longer compatible with the IEC 61499 standard. Execution of these hybrid models is then provided by translating both function blocks and C-Net models into an X-Net model. Code is then generated, following the approach presented in [65] for compilation of IEC 61499.

Table 2.2 compares various specifications approaches that could replace ECCs. While all of the alternatives shown are capable of modelling parallelism and hierarchy, only SFCs are indirectly compatible with IEC 61499 by using extra function blocks to accommodate the different semantics. It should be noted that sub-sets of the given languages could be made to be compatible with the standard, as demonstrated by the formalisms which translate ECCs for formal analysis. Approaches such as NCES and the Argos language which do not support the modelling of arbitrary variables are too limited to be considered as a general replacement for ECCs.

<table>
<thead>
<tr>
<th>Language</th>
<th>Feature</th>
<th>IEC 61499 Compatible</th>
<th>Data Modelling</th>
<th>Parallelism</th>
<th>Hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECCs [15]</td>
<td>Yes</td>
<td>External</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SFCs [61]</td>
<td>Yes</td>
<td>External</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>C-Net [64]</td>
<td>No</td>
<td>Yes or External</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>UML Statecharts [36]</td>
<td>No</td>
<td>Boolean Only</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>NCES [66]</td>
<td>No</td>
<td>External</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SyncCharts [59]</td>
<td>No</td>
<td>Boolean Only</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Argos [67]</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.4.3 Discussion

The graphical synchronous languages offer formal execution semantics with hierarchy and concurrency for the design of complex control behaviour. Unfortunately, the languages are not compatible with the IEC 61499 standard.

Existing attempts to replace IEC 61499 ECCs have been predominantly focused on formalising function blocks, instead of improving the design capabilities of the standard. With the exception of SFCs, the proposals for an alternative specification language are similarly incompatible with the standard. However, the use of SFCs to describe more complicated behaviour is not ideal because of the additional function blocks required to make the behaviour compatible with the standard. As a result, none of the languages surveyed can be used as a direct replacement for ECCs, instead IEC 61499 compatible sub-sets or semantic changes would be required.

In conclusion, IEC 61499 ECCs can be substantially improved by adopting features from existing languages. The features of hierarchy and parallelism within graphical synchronous languages are of particular interest for a replacement of ECCs. In order to utilise a wider range of tools and platforms compliance with the function block standard is important. Therefore, Chapter 4 presents a new synchronous semantics for the standard, that extends ECCs by adopting features from graphical synchronous languages but remains compatible with the standard.

2.5 Migrating From IEC 61131-3 to IEC 61499

For industries that require high reconfigurability and a fast time-to-market, the restrictive and outdated programming techniques and languages of IEC 61131-3 already motivate manufacturers to look for newer technologies. For industry, the IEC 61499 standard presents numerous benefits with respect to specification, as well as an open standard for platform independent design. In addition, the IEC 61499 standard ensures systems described using one of the IEC 61131-3 languages can also be described using event triggered function blocks. A tool chain to aid manufacturers to migrate their existing code base into this new language will be an important milestone in the adoption of the new standard.

In [68], Wright et al. discuss reasons why people might chose to re-engineer existing code: a reduction in maintenance costs, additional functionalities, and new hardware. IEC 61499 simplifies the development of control systems with a component-oriented structure improving reuse between projects. In addition, IEC 61499 provides specification methods for the distribution of a system with abstract communication channels. A function block specification is also platform independent allowing reuse on a wide range of hardware.

While it is possible to directly integrate existing IEC 61131-3 code into algorithms in
2.5 Migrating From IEC 61131-3 to IEC 61499

IEC 61499 basic function blocks, modification of code for the PLCOpen [69] semantics of IEC 61131-3 languages would be required. However, this approach would retain many of the disadvantages of existing coding practices, and provide minimal advantages from IEC 61499.

A significant difference in the two standards is the use of global variables in IEC 61131-3 which are not possible in IEC 61499. While global variables can be re-implemented as interface variables passed between function blocks, implementing a data intensive controller across multiple blocks complicates the design. For example, before any component can alter the data, the data produced by all other blocks must first be synchronised and merged together. Further, any IEC 61499 re-implementation of global variables will introduce a performance overhead not present in the original specification. This highlights a comparative limitation of the IEC 61499 standard, where absence of global data, as used in IEC 61131-3, leads to unnecessarily complicated or infeasible implementations. A complete migration approach must therefore address the usage of global variables in IEC 61499.

What follows is a review of translation approaches from IEC 61131-3 languages into IEC 61499. While many approaches do not handle the translation of global variables, the core techniques presented may be adopted and extended.

2.5.1 Existing Approaches

A single generic tool or approach for all IEC 61131-3 languages is not feasible because of the differences between the textual and graphical IEC 61131-3 languages and their vendor-dependant semantics. Most works focus only on translation of routines described in one of the IEC 61131-3 languages, rather than the complete specification for a PLC controller. Such approaches are still useful as a comparison of methodologies, but they would need to be extended to support data intensive control applications.

In [70] Wenger et al. extend their previous work [71] to translate IEC 61131-3 Function Block Diagrams (FBDs) into a IEC 61499 function block network in a semantically correct way. The work uses a look-up table to identify equivalent IEC 61499 function blocks for each of the library FBDs that are used in the original implementation. This required some initial work as all library FBDs had to be created as a function block, and additional meta data was collected in order to reconcile the differences between the interfaces. The differences include changes in port names and the use of events instead of some boolean data ports. Event connections for the function block network were inferred from the XML or ST representation of the FBD network.

After the initial creation of library blocks, translation of arbitrary FBD programs is possible. Though unlikely for a single FBD, the generated function block network can be distributed in a variety of ways. However, the execution approach for IEC 61499 function
blocks adds a significant overhead compared to the original sequentially executed FBD.

Vyatkin et al. [72] present an approach titled *Straight-forward ladder to FB transformation* that similarly translates a ladder logic routine into a network of function blocks. The only advantage of this is that it allows the distribution of sections of ladder logic routines. However, the approach relies on a large number of simple blocks to represent otherwise simple code, which reduces the maintainability.

Riedl et al. [61] translates IEC 61131-3 SFCs into an IEC 61499 network of blocks. Two IEC 61499 function blocks were also developed to re-create SFC execution semantics in an event-triggered network. The translation abstracts SFC transitions and actions to event inputs and outputs in the function block network. Transitions that are complicated boolean expressions would require an additional basic function block to generate the required event. Similarly, complicated actions would also require a separate function block to perform data computation. While behavioural equivalence is achieved, the performance cost of executing the function block network is much higher than the SFC implementation. In addition, the maintenance of a network of function blocks is higher than a single SFC diagram.

Dai et al. in [73] present a unique class oriented solution, where IEC 61131-3 is reproduced in IEC 61499 by grouping together functionalities instead of components. This technique reduces or removes the amount of data being shared between blocks, as a single class encapsulates all data required for its functionality. It is also useful for removing global variables which are not easily implemented in IEC 61499. Unfortunately, the migration does not fully utilise the function block standard, as the ECC for a class just executes an algorithm defined in the original IEC 61131-3 language. The model-view-controller design pattern is also not implementable when component functionalities are separated in this way. Although, distribution is still possible as multiple instances of the class function block can be instantiated on different devices to control different components.

### 2.5.2 Discussion

Most of the existing approaches create function block networks from routines, but do not address the other aspects of a IEC 61131-3 specification. By creating a function block network from a single routine, the approaches allow sub-sections of a routine to be distributed. However, for realistic industrial applications, it is unlikely that a single routine within an application would be distributed on different devices. The approach also creates a more complicated specification, as multiple function blocks with interfaces and ECCs or block networks need to be maintained instead of a single textual or graphical routine. The semantics of a function block network also introduces a significant performance overhead compared to the much simpler semantics of an IEC 61131-3 routine, leading to a slower
implementation.

As global variables are incompatible with the concept of easily distributable components proposed by IEC 61499, they are not present in the standard. Unfortunately for industrial developers with data-intensive controllers, there is no efficient way to share data between multiple blocks. Data connections are only suitable for a flow of data through a chain of processing units. If multiple blocks need to both read and write to the same variable, there is a lot of wasted memory and execution overhead for the intended behaviour.

The class-oriented solution presented by Dai et al. [73] is able to re-implement a specification without sharing data between blocks. However, the approach relies on the mutually exclusive grouping of routines into basic function block algorithms so that internal variables can be used. In addition, the component-based design of other function blocks cannot be easily used in conjunction with a class-based block. Section 5.2.1 includes some further discussion of global variables in the IEC 61499 standard.

2.6 Remarks

The surveyed research gives an indication of the current state-of-the-art in IEC 61499 for industrial control design. None of the existing literature adequately address the needs identified in Section 2.1. Adoption of the synchronous semantics for IEC 61499 developed by Yoong et al. is a partial solution, as execution issues in the standard may be avoided. Further to this, direct execution afforded by the semantics provides the highest possible performance. However, ECCs lack the ability to describe synchronous behaviour, and are less capable than other graphical state-machines for the description of complicated behaviour.

As the existing proposals for ECC replacements either lack the required features or are incompatible with the standard, this thesis will extend the state of the art by introducing a new syntax and semantics for control-flow design in basic function blocks.

Using the new synchronous semantics, code generators will be developed to allow users to execute applications that use the new syntax on different platforms. A code generator will also be integrated with the nxtStudio commercial IDE, to offer some of the benefits of the synchronous semantics to its users. In addition, the same code generator is strongly integrated with a new IDE for IEC 61499 that is based on the synchronous semantics for IEC 61499. This IDE provides a user friendly interface for the extended ECC syntax, and provides a central tool for additional formal analysis tools made possible by the formal semantics.

To encourage the adoption of IEC 61499, this thesis will also develop a migration approach from IEC 61131-3 to IEC 61499. The new synchronous semantics removes some
of the differences between the two standards, simplifying migration and encouraging the use of the new extended syntax for control flow design.
This chapter describes the IEC 61499 standard by presenting the design of a simplified baggage handling system, which will be used throughout this thesis. This industrial application serves to demonstrate the advantages of IEC 61499 over current industrial practices. The tools and techniques offered in this thesis will also be further motivated, with the goal of simplified development of complex and reliable control systems.

Baggage handling systems (BHSs) are ideal applications for IEC 61499 because of the frequent reuse of components, such as conveyors, spread over a large area. The task of a BHS is to accurately route bags from multiple sources to multiple destinations. All bags must also be routed through a common security sub-section, generally consisting of X-ray machines. As bag positions are tracked and determined from a limited set of inputs, the control of a BHS is very data-driven.

The trend for larger and more versatile airports is driving system designers to adopt technologies that enable greater customisation without increasing the development time unreasonably. IEC 61499 is well-placed to meet these challenges, as its component-based approach facilitates easy reuse of components and the design of distributed control systems across large physical distances.

Though applied to baggage handling, this work is a realistic case study of using function blocks for developing material handling systems. The ability to maintain the same system description throughout the entire development (from design conception, to architectural modelling, simulation, visualisation, and synthesis) ensures coherency between
user requirements and the final implementation. The system description can be incrementally refined and tested, while necessary modifications can be easily fed back to the model throughout the entire design flow. The importance of thorough testing and simulation for complex material handling systems, like baggage handling in airports, has already been well-documented [7, 8].

In the following section, the design methodology for IEC 61499 applications is demonstrated. Section 3.2 then concludes with a discussion on the use of IEC 61499, especially with regard to the older IEC 61131-3 standard [4].

3.1 Development of a BHS

This section presents an approach for the development of the prototype BHS shown in Figure 3.1 using IEC 61499. The BHS consists of 23 conveyors, labelled T101–T304, each of which has one infra-red bag detector. Conveyor T105, at the top of the figure, has two additional entry points, into which the paths from conveyors T206 and T304 merge. This system was deployed onto three IEC 61499 devices (an abstraction for a controller), as illustrated in the figure, with one device for the main loop (conveyors starting with T1), and one for each of the other paths (T2 and T3).

![Diagram of conveyor system]

Figure 3.1: Layout of the prototype baggage handling system.
3.1 Development of a BHS

3.1.1 Model-View-Controller Design

Instead of just developing the controller for a component, IEC 61499 is also suitable for the Model-View-Controller design pattern [74]. In this approach, three elements are defined for each component, illustrated in Figure 3.2. The model, sometimes referred to as the plant, simulates the physical aspects of the component, including sensor data and actuator motion. The controller controls the model to implement the required behaviour. The visualisation presents a visual representation of the component, optionally also including a graphical interface to allow a human-machine interface to the controller. The set of elements for a component are connected together, where each element provides and receives data with the other elements. This approach allows the closed-loop testing of the controller together with a model of the actual component. The visualisation is used to provide the developer or user with a detailed view of the system at run-time.

![Model-View-Controller Diagram](image)

Figure 3.2: Illustration of Model-View-Controller Design.

3.1.2 A Conveyor Section

Figure 3.3 shows a sub-section of a baggage handling system, where a single conveyor section is comprised of a motor for the belt, sensors and optionally merge or divert points. As the motor moves the belt forwards or backwards, the movement of the belt is measured by a rotary encoder (not pictured), allowing a controller to measure the movement and speed of the belt. To detect bag positions, there is at least one infra-red bag detector or photo-eye on each conveyor section. Photo-eyes provide a binary input to the controller indicating whether or not a bag is present at a particular position.

The arrows at the bottom of the figure indicate the sources of bags, which follow the green or blue conveyor paths. At the merge point, which is part of the large conveyor on the right, the two paths combine into a single path. A conveyor may also have a diverter, which is able to push bags from the middle of one conveyor section onto the start of
Figure 3.3: Components of a conveyor.

another. Using diverters, the controller must route bags in such a way that each bag gets to its required destination.

In the following subsections the development of the model and controller for a generic conveyor section will be described. Issues in the specification of certain behaviours are also identified to motivate the extensions to IEC 61499 presented in Chapter 4.

Plant Model

The plant model mimics the actual conveyor by modelling the movement of bags on the belt and the action of any diverters. It also simulates two sets of data: a rotary sensor monitoring belt movement and infra-red bag detectors at various positions along the conveyor.

The interface for the Conveyor_Model function block is shown in Figure 3.4. The interface separates event and data ports to the upper and lower sections of the block respectively. Data ports must be associated with at least one event, as the internal (buffered) copy of the variable is only updated when an associated event is present. These associations are illustrated with vertical lines between the ports.

Conveyor_Model is a composite function block, which contains the function block network shown in Figure 3.5. This network contains the embedded basic function blocks: Conveyor_Belt_Model and Conveyor_Photoeyes_Model, which simulate the two sets of data required to form the conveyor model. Conveyor_Belt_Model simulates the rotary encoder by modelling the movement of the belt including acceleration
and deceleration. The measurement of belt movement, relative to its start-up position, is passed to the Conveyor_Photoeyes_Model function block through the connection between the EncCount data port on each block. Using this position data, Conveyor_Photoeyes_Model maintains a representation of the bags on the conveyor and simulates the photo-eye detectors.

Figure 3.5: Network of function blocks which model a conveyor section.
In IEC 61499, function blocks are only activated once an input event is present. During execution, events and data that are received at the interface of the composite block are passed to the connected ports in the network. Blocks which receive an input event are activated, then they are executed, possibly emitting their own events. The connections between function blocks in a network dictate the propagation of events and thus the activation order of blocks. For example, when the Tick event is received at the interface to the Conveyor_Model block, the connection to Belt_Model’s TICK input event activates the block. Then, when Belt_Model emits the CNF output event the Photoeye_Model function block is activated. After the blocks finish their execution, they are deactivated until another connected input event is present at the interface of Conveyor_Model.

As the plant models physical behaviour, the function blocks need an input to identify the passage of time. This can be achieved in two ways: (1) an event can be used to indicate the passage of a pre-configured amount of time, or (2) an input data port can define the quantity of time passed since the last execution.

In the function block network, the input event TICK is shown on the interface of both blocks. The input msSinceMidnight is used by Conveyor_Belt_Model to define how much time has passed between TICK events. For every TICK event, the distance travelled by the belt is re-calculated using the simulated speed. The rotary encoder is simulated by toggling the EncoderState output between 0 and 1 every time the belt moves 5cm.

The Conveyor_Photoeyes_Model does not include the msSinceMidnight input, as its behaviour is only dependent on the position of the belt and not on the passage of time. The infra-red detectors require an array of all the bag positions on the conveyor to determine if they intersect with the position of any attached detectors. Bag data is therefore passed between conveyor sections as they exit the downstream end or when they intersect an activated diverter.

Both the Conveyor_Belt_Model and the Conveyor_Photoeyes_Model blocks are basic function blocks. While the behaviour of composite function blocks is defined by a network of embedded blocks and their inter-connections, the behaviour of a basic function block is defined by its execution control chart (ECC).

Figure 3.6 shows the ECC for the Conveyor_Belt_Model function block, which consists of EC (execution control) states, EC transitions, and EC actions. By convention, the initial state of an ECC is represented with a bold or double outline. A state can optionally be associated with a set of actions, which may consist of an algorithm and/or output event to be emitted at the completion of the algorithm. Each algorithm can be described using any of a range of languages, including IEC 61131-3 languages [4], Java, C or C++. As an ECC is a Moore-machine, the actions associated with a given state are executed once on entry to the state. The priorities of transitions are labelled above the
conditions between angled brackets. In the ECC, \(< 0 >\) is the highest priority. Priorities are defined by the user and stored in the XML file for the function block, where the textual order of transition declarations indicate the transition priority.

![Figure 3.6: ECC for the ConvBeltModel block.](image)

In the IEC 61499 standard, an ECC executes a function block \textit{run} when activated by an event. During a \textit{run}, transition conditions are evaluated and new states are entered until no further transitions evaluate to true. These conditions are expressed using an input event and/or a boolean guard condition, where the condition \( 't' \) is used to represent an \textit{always true} transition.

In contrast, using the synchronous semantics presented by Yoong et al. [30], ECCs execute once per \textit{tick}, and take at most one transition. As a result, ECCs must be designed specifically for the synchronous semantics to ensure that no input events are unintentionally missed. To achieve this, events must be tested in transitions exiting from every state. From the initial state \texttt{START}, when the event \texttt{INIT} is present, the ECC will transition to the \texttt{RESET} state. Upon entry, the algorithm \texttt{Initialise} will be executed, then the event \texttt{INITO} will be emitted. In the next tick, if no input events are present, the ECC will remain in the \texttt{RESET} state. Alternatively, if the event \texttt{INIT} is present again, the ECC re-enters the \texttt{RESET} state and executes the algorithm \texttt{Initialise} and emits the output event \texttt{INITO}.

The sampling of block inputs at the start of a \textit{tick} also allows for multiple events to be simultaneously present. As the IEC 61499 standard is event-triggered it explicitly declares that events cannot occur simultaneously. This is enforced within an ECC by forcing transition conditions to test the presence of at most one event. Hence, if both \texttt{INIT} and \texttt{TICK} events are \textit{true} simultaneously, the transition to the \texttt{RESET} state will be taken and the \texttt{TICK} event is missed.

The plant model communicates with the controller using service interface function
blocks shown in Figure 3.7. The PUBLISH_2 and SUBSCRIBE_2 blocks encapsulate socket communication, allowing the controller to be executed remotely or on the same device. The abstraction of hardware specific functionalities into service interface function blocks, allows function block specifications to be agnostic to the execution platform. Different hardware can then be targeted by generating hardware specific code for a particular service interface function block. It is also possible to change the type of service interface function blocks in order to use different protocols or directly access hardware inputs and outputs.

The Controller

The controller for each conveyor section uses the data from the infra-red detectors and the rotary encoder to estimate the positions of the bags on the conveyor. The controller is responsible for a number of functions, as outlined below:

- **Monitor bag positions** – Each bag is uniquely identified, and an accurate model of bags on the conveyor section is maintained in order to correctly route them.

- **Manage merging** – If a bag is about to merge but there is no free space, the merging conveyor will pause until there is a sufficient gap for the waiting bag to merge.

- **Transport and route bags** – The conveyor belts and diverters are controlled in order to transport each bag to the correct location.

- **Report conveyor status** – The status of the routing controller, the belt motor, and the infra-red detectors are recorded and reported periodically. This data is then used to investigate various fault conditions.
- **Motor Control** – Control the movement of the conveyor belt to avoid bag collisions. In addition, power usage should be minimised by stopping the conveyor when there are no bags on it.

In an ideal implementation, the unrelated controller tasks of bag management, status reporting and motor control would be implemented in separate blocks with connections between them for shared data. However, there is no mechanism in IEC 61499 that can be used to maintain a consistent value of data shared between multiple blocks. As such, the controller function block is instead implemented as a basic function block with internal variables to store all bag tracking data. As a result of this inadequacy, the ECC for the controller has many transitions to react to several different events which read and modify the bag data.

![Diagram of ECC](image)

Figure 3.8: Part of the Controller’s ECC to control the motor.

Figure 3.8 shows a sub-set of the Controller’s ECC, only responsible for the control of the conveyor’s motor. This small ECC demonstrates the difficulty of handling exceptions such as the **Error** input event. Each state in the ECC requires a high-priority transition that reacts to this event, which in a larger ECC introduces many more transitions complicating the design.

### 3.1.3 Application Configuration

The IEC 61499 standard provides additional components to describe the architecture and deployment properties of an application. An IEC 61499 **System** is used to define a collection of **Devices**, or processors, on which the application can be distributed. Figure 3.9 shows the system configuration for the example BHS application. Within a **Device**, such as **DEVICE0**, multiple independent **Resources**, or processes, can be defined. **DEVICE0** contains a resource for the management of the device itself, and two resources which contain the plant and controller function block networks respectively.
Within a single plant resource, the conveyor models exchange bag information, as the bags move from one conveyor to another. Similarly, in a controller resource, the conveyor controllers constantly share a variety of status information with upstream and downstream conveyors. For this example application, input and output data is exchanged between the two resources using pairs of **PUBLISH** and **SUBSCRIBE** service interface function blocks. **PUBLISH** and **SUBSCRIBE** blocks are also used to exchange data between the plant and controller resources on two different devices.

### 3.1.4 The Visualisation

The state of the simulated plant and controller can be visualised together by using a separate application for domain specific visualisation. Using nxtStudio, a visualisation can be developed for each component using a graphical editor with support for animations and advanced features using C# to control the visual appearance. A canvas is then used to graphically arrange the visual representation of each component to provide a complete view of the system as shown in Figure 3.10. The visualisation can also aid human-machine interaction by displaying graphical control panels, as shown at the top of the figure, allowing the user to interact with the simulated system.
3.2 Discussion

In the development of this case study, the advantages of IEC 61499 were utilised to create a reusable library of distributable components. However, some aspects of the resulting design were not ideal, such that other languages are able to produce more succinct descriptions of the same behaviour.

3.2.1 Advantages of IEC 61499

IEC 61499 offers significant advantages over the traditional IEC 61131-3 approach. Function blocks can be used for model-driven engineering, improving the reuse of components and enabling code synthesis for multiple deployment platforms. IEC 61499 is also able to employ the model-view-controller design pattern and describe both the model and the controller, allowing closed-loop testing prior to deployment. Visualisation of the system further aids the designer by displaying complete system information and providing a graphical interface for human-machine interaction. Visualisation also benefits debugging of the controller prior to the construction of the physical system, and once the hardware has been constructed, the same visualisation can be used to monitor the deployed system.

The graphical nature of the standard also simplifies many aspects of a design. In particular, ECC state-machines and the configuration and connections of components in a function block network are much simpler to design and understand than textual code. IEC 61499 systems, devices and resources also allow developers to easily define and modify the distribution of an application. Tools are then used to simplify the deployment of an application to multiple devices.
3.2.2 Deficiencies

As noted in the design of the conveyor controller, it is not possible for multiple blocks in a network to all read and write to the same variable. Instead, internal variables in a single basic function were used, requiring the ECC to describe multiple behaviours. This usage of a single ECC also demonstrated that ECCs are too simplistic to describe complex behaviours, to the extent that the ECC for the controller block requires more states and transitions than would be necessary using other state-machine languages, including IEC 61131-3 SFCs. Further, as discussed in the previous chapter, ECCs have also been shown to be ineffective for describing exception handling behaviour, lacking hierarchy and parallelism found in other state-machine languages.

The most significant deficiency of the standard is the ambiguous semantics which were discussed in the previous chapter. As a result, function blocks developed using one tool are likely to be ineffective when executed with another tool. The synchronous semantics for IEC 61499, developed by Yoong et al. [31], offers many benefits including superior performance and determinism. Despite not being event-triggered, the semantics are compatible with the standard, allowing function blocks developed assuming event-triggered semantics to also be executed. However, as with all variations in execution approaches, the behaviours will diverge under certain scenarios. Importantly, ECCs cannot be used for reliable software development with a synchronous semantics, as simultaneous events will always be missed. The following chapter seeks to address the identified deficiencies by developing an extended synchronous semantics for IEC 61499.
Hierarchical and Concurrent ECCs for IEC 61499 Function Blocks

This chapter enhances basic function blocks by allowing explicit hierarchy and concurrency within an IEC 61499 basic function block, similar to Statecharts [27]. These extensions, called Hierarchical and Concurrent ECCs (HCECCs), can be resolved into a fully IEC 61499 standard compliant form.

As discussed in section 3.2.2 of the previous chapter, the ECC Moore-machines contained within basic function blocks are inadequate for the design of complex behaviours. For example, the single control flow described by an ECC complicates the modelling of concurrent behaviours. In addition, ECCs do not allow users to separate high-level behaviour such as initialisation and error handling. As a result, more states and transitions are required to describe the same behaviour compared to other graphical state-machines.

Existing formalisms, such as Statecharts provide the ability to embed hierarchy and concurrency within a single state-machine. However, they cannot be used as a replacement for ECCs as their semantics are incompatible with the IEC 61499 standard. For example, Statecharts is essentially a control flow language while IEC 61499 requires an integration of control and data flow. Also, IEC 61499 has a restriction that does not allow transitions to test more than one input event at any time.

HCECCs introduce some syntactic sugar in the form of two operators, (parallel and refinement) which enable a Statecharts-like state-machine to be embedded within a
single basic function block. Each of these operators can be resolved to an IEC 61499 standard composite function block executing using a synchronous semantics. As shown later in Chapter 5, HCECCs can be used to model real-life applications. Also, Chapter 6 shows how efficient and standard-compliant code can be generated from HCECCs.

The main contributions of this chapter are:

1. A formalism for IEC 61499 function blocks and their execution using synchronous semantics.

2. The development of the HCECC framework using the parallel and refinement operators.

3. A denotational synchronous semantics of IEC 61499 function blocks that maps HCECCs to a standards-compliant representation.

This chapter is organized as follows. Section 4.1 presents a formalism for IEC 61499 and a description of the semantics for basic and composite function blocks. The proposed hierarchical and concurrent extension to ECCs (HCECCs) is introduced in section 4.2. The two operators used by an HCECC are presented in sections 4.2.1 and 4.2.2. Section 4.3 discusses the semantics of HCECCs by comparing them with other graphical state-machines. Finally, section 4.4 concludes with a summary of the features, limitations and advantages of HCECCs.

## 4.1 Formalisation of IEC 61499

This section presents a formal model for IEC 61499 function blocks (FBs). The following concepts and notations will be used frequently throughout the rest of this chapter:

- ** Totally ordered set:** A set \( A \) is totally ordered iff for any two elements \( a, b \in A \), where \( a \neq b \), \( a <_A b \) or \( b <_A a \) where \( <_A \) is the antisymmetric and transitive ordering relation for set \( A \). The sub-script is omitted when the context is obvious and unambiguous. The notation \( A = \{a_1, a_2, \ldots, a_n\} \) is used to describe a totally ordered set containing \( n \) elements where for all \( i, j \in [1, n] \), if \( i < j \), then \( a_i <_A a_j \).

- **Totally ordered union:** If \( A \) and \( B \) are totally ordered and non-intersecting sets, then the totally ordered union \( C = A \oplus B \) is defined as follows. Firstly, \( C \) contains all elements of \( A \) and \( B \). Also, for each element pair \( a_1, a_2 \in A \), if \( a_1 <_A a_2 \), then \( a_1 <_C a_2 \). For each element pair \( b_1, b_2 \in B \), if \( b_1 <_B b_2 \), then \( b_1 <_C b_2 \). Furthermore, for any two elements \( a \in A \), and \( b \in B \), \( a <_C b \). The totally ordered union therefore represents a *concatenation* of set \( A \) followed by set \( B \).
4.1.1 Function Block Interfaces

Function blocks have well-defined interfaces containing input and output events and variables. Figure 4.1 shows the interface for the Conveyor_Model block presented previously in Figure 3.4 of Chapter 3, which encapsulates the complete behaviour of a conveyor in a baggage handling system. As shown in Fig. 4.1, a function block interface is divided into four parts: input events on the top left, output events on the top right, input variables on the bottom left, and output variables on the bottom right. The IEC 61499 standard requires that each input (output) variable to be associated with at least one input (output) event, represented by vertical lines between events and data variables. For example, the input variable MotorForward is associated with the input event Tick. These associations enforce an input variable to be sampled only when an associated input event is present. (Events are either present of absent, where as variables have a concrete value at any instance). When no associated event is present, the block must use a previously sampled value for the variable. Similarly, the value of an output variable is updated only when an associated event is emitted.

![Figure 4.1: Interface for the Conveyor_Model Composite Block.](image)

Interfaces are common to both basic and composite function blocks in IEC 61499 systems. They are defined as follows.

**Definition 1** (Function Block Interface). A function block interface is a tuple \( I = (E^I, V^I, \alpha^I, E^O, V^O, \alpha^O) \) where \( E^I \), \( V^I \), \( E^O \), and \( V^O \) are finite sets of input events, input variables, output events and output variables respectively. \( \alpha^I \subseteq E^I \times V^I \) and \( \alpha^O \subseteq E^O \times V^O \) are the sets of input and output associations. For every \( v_I \in V^I \), there must exist some \((e_I, v_I)\) \(\in\alpha^I\) for some \(e_I \in E^I\). Similarly, for every \(v_O \in V^O\), there must
exist some \((e_O, v_O) \in \alpha_O^T\) for some \(e_O \in E_O^T\).

The interface \(\mathcal{I}\) of the \texttt{Conveyor\_Model} block in Figure 4.1 contains the following:

- \(E_I^T = \{\text{Init}, \text{Tick}, \text{BagIn}, \text{BagMerge}\}\)
- \(V_I^T = \{\text{msSinceMidnight}, \text{MotorForward}, \text{MotorReverse}, \ldots\}\)
- \(\alpha_I^T = \{(\text{Tick}, \text{msSinceMidnight}), (\text{Tick}, \text{MotorForward}), (\text{Tick}, \text{MotorReverse}), \ldots\}\)
- \(E_O^T = \{\text{InitO}, \text{TickO}, \text{BagOut}, \text{BagDivert}\}\)
- \(V_O^T = \{\text{EncoderState}, \text{PEDetects}, \text{BagOutID}, \ldots \text{BagDivertID}, \ldots\}\)
- \(\alpha_O^T = \{(\text{TickO}, \text{EncoderState}), (\text{TickO}, \text{PEDetects}), \ldots\}\)

### 4.1.2 Basic Function Block

Basic function blocks contain a function block interface, as well as algorithms, internal variables and an execution control chart (ECC). Figure 4.2 shows the interface for the basic function block \texttt{Conveyor\_Photoeyes\_Model}. Algorithms are user defined functions, described in any language suitable for data processing. They operate over the input and output variables of the interface as well as the internal variables of the block. Internal variables and algorithms are collectively referred to as \textit{local FB declarations}. Local declarations do not have a visual representation in IEC 61499. They are defined below.

![Figure 4.2: Interface for the Conveyor\_Photoeyes\_Model Basic Function Block.](image-url)
Definition 2 (Local FB Declarations). Given a FB interface $\mathcal{I}$, local FB declarations $L^I$ is a pair $\langle V_L^I, A_L^I \rangle$, where $V_L^I$ and $A_L^I$ are sets of internal variables and algorithms respectively. Algorithms operate over internal variables $V_L^I$, as well as input and output variables $V_I^O$ (of the interface $\mathcal{I}$).

For the basic function block Conveyor_Photoeyes_Model in Figure 4.1, the local declarations are $L^I = \langle V_L^I, A_L^I \rangle$ where:

- $V_L^I = \{\text{BagModel, BagExited, BagDiverted}\}$
- $A_L^I =$ \{\text{INIT, TICK, CheckDiverters, BagExited, BagDiverted, BagIn, BagMerge1}\}

Recall that in addition to an interface and local declarations, a basic function block also contains an execution control chart (ECC). Figure 4.3 shows the execution control chart for the Conveyor_Photoeyes_Model block. The ECC consists of a finite set of states, with an initial state, START, shown with a bold outline. Each state is associated with a finite sequence of actions, which are comprised of algorithm executions (from $A_L^I$) and the emission of output events. For instance, state INIT is associated with a sequence of actions that will execute the algorithm INIT and emit the INITO output event. This sequence of actions is executed whenever state INIT is entered.

![ECC Diagram](image)

Figure 4.3: ECC which simulates the infra-red PhotoEye sensor.

A transition in an ECC consists of a source state, a condition consisting of an input event and a boolean guard condition over variables and a destination state. For example, in Figure 4.3, the source state START has a transition that requires the presence of the
INIT input event and a *true* boolean condition to the destination state INIT. A transition triggers when the corresponding input event is present and the associated boolean condition evaluates to true. The triggering input event and boolean condition are optional, and may not be provided for a transition. *Always-true* transitions, shown with a condition of 1 have no associated events or boolean guard conditions.

Transitions exiting a state are totally ordered based on transition priorities. This order is shown in Figure 4.3, where the transition from IDLE to Tick is ordered first (labelled with \(<0\>\), before other transitions out of state IDLE (labelled with \(<1\>, <2\> \ldots\). This label is omitted if the state only has one exiting transition. As described later, transition ordering ensures deterministic execution since only the highest priority enabled transition is taken during execution.

Having discussed the various constituents of a basic function block, it is now defined below.

**Definition 3** (Basic Function Block). A basic function block, BFB, is a tuple 
\(\langle I, L^I, ECC^L, L^T \rangle\) where \(I\) is a FB interface, \(L^I = (V^I, A^I_L)\) are the local declarations for the block and \(ECC^L, L^T\) is a tuple \(\langle S, s_0, \lambda, T \rangle\). Here, \(S\) is a finite set of states with \(s_0 \in S\) being the initial state. The action function \(\lambda : S \rightarrow (A^T_L \cup E^T_0)^*\), returns a finite sequence of algorithms and/or outputs events (actions), that are executed and emitted for any state \(s \in S\). \(T : S \rightarrow 2^{(E^T_L \cup \{1\}) \times B(V) \times S}\) is the transition function, where \(2^{(E^T_L \cup \{1\}) \times B(V) \times S}\) is restricted to contain only ordered sets, \(T(s)\) therefore, is a totally ordered set for any state \(s \in S\). \(\hat{V}\) refers to the set of all possible valuations of internal, input and output variables, and \(B(\hat{V})\) refers to all boolean expressions over such valuations.

A transition \(t \in T(s)\) is described as \((e, b, s')\), where \(e\) is either an input event, or 1 (no triggering event), \(b\) is a boolean expression that can be evaluated to be *true* or *false* based on the values of internal, input and output variables of the FB. The transition \(t\) is said to be enabled when \(e\) is present (or is equal to 1), and \(b\) evaluates to true. Any transitions that use the *always-true* event (1) with a *true* boolean condition is always enabled. If \(t\) is enabled, the ECC moves from being in state \(s\) to \(s'\) and the actions \(\lambda(s')\) are executed. The short-hand \(s \xrightarrow{\text{true} \ b} s'\) is interchangeably used instead of \((e, b, s') \in T(s)\).

Note that the set of transitions for every state in the ECC of the basic function block is totally ordered. As we discuss later, our execution semantics restrict ECC execution to always take the highest-priority enabled transition, resulting in deterministic execution.

For the ECC \(ECC^L, L^T = \langle S, s_0, \lambda, T \rangle\) shown in Figure 4.3:

- \(S = \{\text{START, INIT, IDLE, TICK, BagIn, BagExited, BagMerge, BagDiverted}\}\)
- \(s_0 = \text{START}\)
• For the state BagExited of the ECC:

- \( \lambda(\text{BagExited}) = \{\text{BagExit}, \text{BagOut}\} \).
- \( T(\text{BagExited}) = \{(1, \text{BagDiverted}, \text{BagDiverted}), (1, \text{true}, \text{IDLE})\} \), which is an ordered set.

Using the short-hand stated earlier, these transitions can be rewritten as:

\( \downarrow \text{BagExited} \rightarrow \text{BagDiverted} \)

\( \text{BagExited} \rightarrow \text{IDLE} \).

Since \( T(\text{BagExited}) \) is an ordered set, the first transition to \( \text{BagDiverted} \) has a higher priority than the second transition to state to \( \text{IDLE} \).

**Execution Semantics of Basic Function Blocks**

The synchronous execution of IEC 61499, separates execution into individual steps known as ticks.

The synchronous execution of a basic function block involves the following steps in every tick:

1. All event inputs \((E^T_f)\) are sampled at the beginning of the tick. An input variable (in \(V^T_f\)) is sampled only if an associated input event is present (as per the associations contained in \(a^T_f\)).

2. From the current state \(s\) in the ECC of the function block, the trigger conditions of transitions in \(T(s)\) are evaluated in order. The first enabled transition (for which the related input event is present and the boolean condition evaluates to \text{true}), is taken and a new state is entered. However, if no transitions are enabled, the ECC remains in the current state, and execution for the current tick completes.

3. When a destination state \(s'\) is entered after a transition, algorithms and events in its set of actions \((\lambda(s'))\) are executed and emitted sequentially in order.

4. At the end of the tick, output events emitted during step 3 are used to sample the values of their associated output variables (according to associations contained in \(a^T_f\)).

The ECC in Figure 4.3 is used to demonstrate the execution sequence of a basic function block. Assuming that the ECC is in its initial state \text{START}, the execution takes the following steps:

1. At the start of the tick, input events and any associated input variables are sampled.

2. The transitions of \text{START} are evaluated in order. Since there is only one transition to state \text{Init}, it is checked whether it can be enabled.
3. If the Init event (the trigger input for the above transition) is present, the ECC takes the transition to the INIT state. The actions of the state Init, the algorithm INIT and the output event INIT0, are executed and emitted respectively.

4. Any interface output variables associated with the emitted output INIT0 are updated.

5. The tick ends, and the same sequence is repeated in the next tick (with state INIT being the current state in the next tick).

**Observation 1** (Determinism of Basic Function Blocks). A BFB \( BFB = \langle I, I^T, ECC^L, L^T \rangle \) is deterministic.

The interface for any function block, including basic function blocks, is deterministic because given the same input events and variables, the sampling of inputs (step 1 of the semantics) is always the same. Similarly, given the same output events and variables, the sampling of outputs (step 4 of the semantics) is always the same. Intuitively, the execution of an ECC is deterministic because of the explicit ordering of transitions. Even if two transitions can be enabled in a tick, only the highest priority transition is taken, (steps 2 and 3).

### 4.1.3 Composite Function Block

A composite function block (CFB) contains an interface and a finite network of connected function block instances.

A function block instance, \( fb = \langle name, FBT \rangle \), has a unique identifier or name, and instantiates some basic or composite function block (FBT). The set of function block instances in the function block network in a CFB are contained in a totally ordered set, and as discussed later, this order is vital for the deterministic execution of the composite function block.

Figure 4.4 shows the function block network embedded within the Conveyor_Model block. This example contains two instances, in order:

- **BeltModel**, an instance of the Conveyor_Belt_Model basic function block.
- **PhotoeyeModel**, an instance of the Conveyor_Photoeyes_Model basic function block.

Note that when referring to the input or output events and variables of a FB instance, the notation \(<\text{instance name}>,<\text{event/variable name}>\) is used. For example, the output variable PEDetects of instance BeltModel is written: BeltModel.PEDetects. Input or output events and variables of the interface of the
CFB are referred to directly by the name of the event or variable. For example, \texttt{InitO}\ refers to the output event \texttt{InitO}\ of the CFB.

Each FB instance in a function block network may be connected through event and data connections to the interface of the CFB and/or to other FB instances in the network. The connections allow the exchange of information between FB instances and the interface of the CFB. The notation \texttt{src} $\rightarrow$ \texttt{dest} is used to describe a connection from a source (event or variable) \texttt{src} to a destination \texttt{dest}. A connection \texttt{src} $\rightarrow$ \texttt{dest} can be written as a pair (\texttt{src, dest}) for brevity.

There are three types of event connections in a function block network:

- \texttt{C_{EI2I}}: From an input event of the block’s interface to an input event of a FB instance contained in the block, such as \texttt{Tick} $\rightarrow$ \texttt{BeltModel.TICK}

- \texttt{C_{EO2I}}: From an output event of a FB instance to an input event of another (or the same) FB instance in the network. For example \texttt{BeltModel.CNF} $\rightarrow$ \texttt{PhotoEyeModel.Tick}. Connections from one instance to an instance that is ordered earlier (such as the connection: \texttt{PhotoEyeModel.Cnf} $\rightarrow$ \texttt{BeltModel.Tick}) are called feed-back connections by convention.

- \texttt{C_{EO2O}}: From an output event of a FB instance to an output event of the block’s interface. For example \texttt{PhotoEyeModel.Cnf} $\rightarrow$ \texttt{TickO}

Similarly, there are three types of data connections:
• \( C_{V12I} \): From an input variable of the interface to an input variable of a FB instance in the network. For example \( \text{Accel} \mapsto \text{BeltModel.Accel} \).

• \( C_{VO2I} \): From an output variable of a FB instance to an input variable of another (or the same) FB instance. For example \( \text{BeltModel.EncCount} \mapsto \text{PhotoEyeModel.EncCount} \).

• \( C_{VO2O} \): From an output variable of a FB instance to an output variable of the composite block, such as \( \text{PhotoEyeModel.PEDetects} \mapsto \text{PEDetects} \).

Only a single source variable can be connected to a destination variable in a CFB. Further, variable connections are restricted such that only variables of the same type can be connected together. Events have no such restriction, such that the input event \( \text{TICK} \) of \( \text{BeltModel} \) can be connected to the input event \( \text{TICK} \) of the CFB and the output event \( \text{PhotoEyeModel.Cnf} \).

Having discussed the various constituents of a CFB, a formal definition is provided below.

**Definition 4** (Composite Function Block). A composite function block CFB is a tuple \((\mathcal{I}, \text{FBNetwork}^I)\) where \( \mathcal{I} = (E^I_1, V^I_1, \alpha^I_1, E^O_2, V^O_2, \alpha^O_2) \) is a FB interface, and \( \text{FBNetwork}^I \) \((\text{FBs}, C_{\text{events}}, C_{\text{var}})\) is the FB network of the CFB, where

- \( \text{FBs} = \{ f_{b1}, f_{b2}, \ldots, f_{bn} \} \) is a finite and totally ordered set of function block instances of size \( n \). Here each \( f_{bi} \) is a FB instance \((\text{name}_i, \text{FBT}_i)\), where \( \text{FBT}_i \) is either a basic or composite function block with interface \( \mathcal{I}_i = (E^I_{i1}, V^I_{i1}, \alpha^I_{i1}, E^O_{i2}, V^O_{i2}, \alpha^O_{i2}) \).

- The event connections set, \( C_{\text{events}} \), contains the connections.

\[
C_{\text{events}} = C_{E12I} \cup C_{EO2I} \cup C_{EO2O} \quad \text{where:}
\]

\[
C_{E12I} \subseteq (E^I_1) \times \left( \bigcup_{i=1}^{n} f_{bi}.E^I_{i1} \right)
\]

\[
C_{EO2I} \subseteq \left( \bigcup_{i=1}^{n} f_{bi}.E^O_{i2} \right) \times \left( \bigcup_{i=1}^{n} f_{bi}.E^I_{i1} \right)
\]

\[
C_{EO2O} \subseteq \left( \bigcup_{i=1}^{n} f_{bi}.E^O_{O} \right) \times (E^O_2)
\]

- The set of data connections, \( C_{\text{var}} \), contains the connections between variables.

\[
C_{\text{var}} = C_{V12I} \cup C_{VO2I} \cup C_{VO2O} \quad \text{where:}
\]

\[
C_{V12I} \subseteq (V^I_1) \times \left( \bigcup_{i=1}^{n} f_{bi}.V^I_{i1} \right)
\]

\[
C_{VO2I} \subseteq \left( \bigcup_{i=1}^{n} f_{bi}.V^O_{O} \right) \times \left( \bigcup_{i=1}^{n} f_{bi}.V^I_{i1} \right)
\]

\[
C_{VO2O} \subseteq \left( \bigcup_{i=1}^{n} f_{bi}.V^O_{O} \right) \times (V^O_2)
\]
Further, for any two distinct connections \((\text{src}_1, \text{dest}_1), (\text{src}_2, \text{dest}_2) \in C_{\text{var}}, \text{dest}_1 \neq \text{dest}_2\).

For the function block network within the \texttt{Conveyor\_Model} CFB (Fig.4.4), FBs contains the two instances in the following order:

- \texttt{BeltModel} of type \texttt{Conveyor\_Belt\_Model}
- \texttt{PhotoeyeModel} of type \texttt{Conveyor\_Photoeyes\_Model}

The set of event connections contains all connections to and from event ports in the network, including:

- \texttt{BeltModel.CNF} \leftrightarrow \texttt{PhotoEyeModel.Tick}
- \texttt{PhotoEyeModel.Cnf} \leftrightarrow \texttt{TickO}

Similarly, the set of data connections contains all connections between data ports, for example:

- \texttt{Accel} \leftrightarrow \texttt{BeltModel.Accel}
- \texttt{BeltModel.EncCount} \leftrightarrow \texttt{PhotoEyeModel.EncCount}.

**Execution Semantics**

The execution of a composite function block \(\langle I, \text{FBNetwork}^I \rangle\) (where \(\text{FBNetwork}^I = \langle \text{FBs}, C_{\text{events}}, C_{\text{var}} \rangle\)) is formalised in Algorithm 4.1. The execution of Algorithm 4.1 involves the following steps in every tick, which are illustrated using the function block network within \texttt{Conveyor\_Model}, shown in Figure 4.5:

1. As per section (A) of the algorithm (lines 3–5) and Figure 4.5, the inputs of the composite function block interface are sampled at the start of the tick. Values of input variables are updated if associated input events are present.

2. Section (B) of the algorithm (lines 6–25) executes each function block instance in the network, in the order they appear in the set FBs. In Figure 4.5 the instances have been drawn in order from left to right, showing that steps B1 and B2 of the instance \texttt{BeltModel} execute before steps B1 and B2 of \texttt{PhotoeyeModel}. The execution of each instance \(\text{fb}_i\) involves the following steps:

   a) Step (B1) (lines 8–24) propagates connected inputs to the instance. In this step, an input event of the interface of the executing instance is set to present if any source event connected to it (through event connections) is present
Figure 4.5: Illustration showing the execution of Conveyor Model composite function block.

(line 12). Note that if a source event is an output event of an FB instance $fb'$ in FBs such that $fb_i <_{FBs} fb'$ or $fb_i = fb'$, then the source event is tested for its presence in the previous tick (line 14). This is done because $fb'$ has not been processed in the current tick (it appears later in the order of FB instances in the network). For example, the connection $\text{PhotoEyeModel}.\text{Cnf} \rightarrow \text{BeltModel}.\text{TICK}$ reads the status of the output event Cnf of $\text{PhotoeyeModel}$ in the previous tick. $\text{BeltModel}.\text{TICK}$ is therefore present if $\text{PhotoEyeModel}.\text{Cnf}$ was present in the last tick, or if the input event $\text{Tick}$ is present in the current tick. The above approach helps avoid causality issues [28]. An event is marked as absent if no source event is found to be present. Connected input variables are handled similarly, such that the input variable takes either the value of the connected port in the current (line 22) or previous tick (line 20).

(b) Step (B2) (line 25) executes the FB instance as per the semantics appropriate for the type of function block. For example, instance $\text{BeltModel}$ is executed using the semantics of a basic function block as discussed in section 4.1.2.

3. Once all FB instances in the network have been executed, section (C) (lines 27–33) processes connections to outputs of the interface, and handles the updating of output variable values. An output variable is updated if associated output events are present.
1. Initialize all variables in $V_f^T$ and $fb.V_f^T$ and $fb.V_o^T$ for all $fb$ in FBs;
2. **Repeat Each** tick
3.  
4.  
5.  
6.  
7.  
8.  
9.  
10.  
11.  
12.  
13.  
14.  
15.  
16.  
17.  
18.  
19.  
20.  
21.  
22.  
23.  
24.  
25.  
26.  
27.  
28.  
29.  
30.  
31.  
32.  
33.  
34. **End**

**Algorithm 4.1:** Execution semantics for a Composite function block.
As shown above, the execution of a CFB involves three parts in every tick: (A) the sampling of interface inputs, (B) the execution of the contained FB instances, and (C) the emission of interface outputs.

For any function block, including composite function blocks, interface events, variables and their associations remain the same during every execution (every tick) of the CFB. Hence, for step (A), given an environment providing the same input events and input variable values during two different ticks of execution, the input sampling will be carried out in the same manner. Similarly, for step (C), if the function block instances inside the CFB network produce the same output events and variables during two different ticks of execution, the output sampling (emission of outputs and modification of output variable values) will be identical.

Given the above, the only possible source of non-determinism in the processing of a CFB could be step (B): the execution of the FB instances contained in the CFB network. Recall that the FB instances in a CFB network are always executed in the same fixed order every tick. Hence, the determinism of CFB execution depends on establishing that the execution of every instance contained in the network is also deterministic. A CFB can contain two types of FB instances: BFB instances or CFB instances. By observation, BFB instances always execute deterministically (Obs. 1). On the other hand, a CFB instance can be resolved into a network of FB instances, resulting finally into a network of BFBs, executing in a fixed order every tick. Since the execution of every CFB can be resolved into the fixed order execution of basic function block instances, and knowing that BFBs execute deterministically, CFBs therefore also execute deterministically (Obs. 2).

**Observation 2** (Determinism of Composite Function Blocks). The execution of composite function block $CFB = \langle I, FBNetwork^I \rangle$ is deterministic.

### 4.2 Hierarchical and Concurrent ECCs

HCECCs introduce two operators to enhance the modelling of IEC 61499 function blocks. The operators are parallel and refinement, which enable Statecharts-like [27] specifications within basic function blocks. The parallel operator allows the description of independent or concurrent reactions by allowing multiple ECCs to execute concurrently within a single basic function block. The refinement operator allows a top-down behavioural specification, where a state in an ECC can be refined by one or more parallel ECCs.

As an example, Figure 4.6 shows a reimplementation of the `Conveyor_Model` composite block (shown earlier in Fig. 4.4), as a basic function block that uses the two HCECC operators. There are two parallel ECCs, named `BeltModel` and `InfraredModel`.
These parallel ECCs refine the state Running of the main ECC for the basic function block. Intuitively, whenever the ECC of the basic function block enters the state Running, the two parallel ECCs start executing. These ECCs execute using the inputs and outputs of the interface of the basic function block, and may also communicate with each other using the internal variables of the basic function block. The parallel ECCs are executed sequentially and in a fixed order once per tick, while the HCECC is in state Running. The parallel execution is terminated when the ECC of the basic function block takes the top-level transition out of state Running.

HCECCs allow arbitrary levels of nested refinement and concurrency within a standard function block interface. The semantics are compositional, following the approach presented by Argos [67], and despite the syntactic extensions, HCECCs remain compatible with the IEC 61499 standard. This section shows how a multi-level HCECC is translated into a standard composite function block. The process, illustrated in Figure 4.7, uses three different algorithms, built around the two operators.

The two operators are now presented, starting with the parallel operator in section 4.2.1 and refinement in section 4.2.2. Finally, the algorithm to flatten arbitrary levels of nested refinement and concurrency is presented in section 4.2.3.
4.2.1 Parallel HCECCs

A parallel HCECC consists of an ordered set of ECCs that share a common interface and local declarations (internal variables and algorithms). Parallel HCECCs can be translated directly to an equivalent composite function block. Parallel ECCs can be used to describe concurrent behaviours, separating independent reactions, but allowing communication between them. In the presented synchronous semantics, they are especially useful as they allow for the design of a state-machine that can react to multiple simultaneous events using multiple ECCs.

It is defined as follows.

**Definition 5** (Parallel HCECC). A parallel HCECC $\text{HCECC}_{||}$ is a tuple $\langle \mathcal{I}, L^I, \text{ECCs} \rangle$, where $\text{ECCs} = \{\text{ECC}_1^L, L^I, \ldots, \text{ECC}_n^L, L^I\}$ is an ordered set of ECCs within an interface $\mathcal{I}$, and with local declarations $L^I$.

Figure 4.8 shows a parallel HCECC $\text{HCECC}_{||} = \langle \mathcal{I}, L^I, \text{ECCs} \rangle$, where $\text{ECCs} = \{\text{ECC1}, \text{ECC2}\}$ contains two ECCs ECC1 and ECC2 that are logically parallel with a synchronous execution semantics similar to Pret-C [75]. Each of these ECCs can access the events and variables of the common interface. In addition, each ECC can also read and modify internal variables and execute any algorithms contained in the local declarations of the function block.
4.2 Hierarchical and Concurrent ECCs

![Image of a sample parallel HCECC](image)

**Figure 4.8: A sample parallel HCECC**

**Execution semantics of Parallel HCECCs**

The synchronous execution of a parallel HCECC involves the following steps in every tick:

1. Interface inputs are sampled, i.e. input events and associated input variables.

2. ECCs contained in ECCs are executed sequentially *in order* (in which they appear in ECCs).

3. Finally, interface outputs (events and variables) are emitted and the tick completes.

Note that the fixed order execution of parallel ECCs mirrors the fixed order execution of function block instances within a CFB (section 4.1.3). This similarity is used to map parallel HCECCs to CFBs in a straightforward manner. This mapping also allows simultaneous events to be reacted to, with each ECC or FB instance reacting to a single event.

Parallel ECCs communicate with each other directly, using variables, which can cause ambiguities or non-determinism in the execution behaviour of a parallel HCECC. For example, if an output variable of the common interface is written to by two ECCs in a single tick, only one of these values can be actually emitted to the interface. In the setting of HCECCs, any ambiguities in execution behaviour are avoided by defining a semantics that ensures every parallel HCECC can be resolved into a composite function block. Thus, each ECC is considered to be analogous to a FB instance in a CFB, executed sequentially in the same fixed order. Communication between these FB instances must therefore recreate the style of direct communication between each ECC. As such, variables that can be written to by an ECC must be passed to the next FB instance that will be executed, even if they are not modified. From this communication, it is elementary to see
that the last FB instance to execute (corresponding to the last ECC) must emit output variables to the CFB interface. Output events are more easily handled with connections, as the FB network can have a event connection from each FB instance to the output event in the interface of the CFB.

Since composite function blocks execute deterministically (section 4.1.3)), parallel HCECCs can therefore also be shown to always execute deterministically and unambiguously. The transformation of parallel HCECC into an equivalent CFB is now formally presented.

Parallel HCECC to CFB Conversion

Algorithm 4.2 is used to convert a parallel HCECC into a composite function block. The connections and basic function block instances within the CFB are created to implement the same behaviour as the sequentially executed ECCs with shared variables. Intuitively, the algorithm performs the following actions:

1. The interface of the HCECC becomes the interface of the newly created composite function block (line 13, Alg. 4.2).

2. A new basic function block instance is created for each parallel ECC in the HCECC (lines 3–6, Alg. 4.2).

3. All newly created basic function block instances are connected in a function block network so that each FB instance can exchange information (using events and variables) with the main interface as well as the other function block instances (lines 7–10, Alg. 4.2).

The details of the above steps are illustrated by showing how the composite function block network in Figure 4.9 is obtained from the parallel HCECC shown in Figure 4.8.

Algorithm 4.2 calls algorithm CreateBFB (Alg. 4.3) for each parallel ECC in the HCECC (line 4). CreateBFB (explained in detail later) converts each ECC \( ECC_{t,t}^s, s_0, \lambda, T \) contained in the parallel HCECC into an instance \( fb \) of a newly created basic function block. Importantly, the interface of each instance is similar to the interface of the parallel HCECC, but additional inputs and outputs have been added to propagate internal and output variables between instances.

After the creation of a FB instance for every parallel ECC, Alg. 4.2 creates the connections between these instances using algorithms CreateEConns (Alg. 4.4) and CreateVConns (Alg. 4.5) (both explained later), on lines 8 and 10. The connections allow each FB to access the same information (events and variables) that were available to the corresponding ECC in the parallel HCECC. The function block network is created
Input: Parallel HCECC $\langle I, L^I, ECCs \rangle$
Output: CFB CFB

1 //Create BFBs;
2 Ordered set FBs = \emptyset;
3 for each ECC $\in$ ECCs do
4    fb = CreateBFB(ECC, $I, L^I$);
5    FBs = FBs $\oplus \{fb\}$;
6 end
7 //Create event connections;
8 Set of event connections $C_{events}$ = CreateEConns(FBs, $I$);
9 //Create variable connections;
10 Set of variable connections $C_{var}$ = CreateVConns(FBs, $I, L^I$);
11 //Create CFB;
12 FBNetwork$^I = \langle FBs, C_{events}, C_{var} \rangle$;
13 CFB = ($I$, FBNetwork$^I$);
14 return CFB;

Algorithm 4.2: Algorithm Par2CFB. Resolves a parallel HCECC into a composite function block

using these instances and sets of connections (line 12). Finally, the CFB is created using this new FB network and the same interface as the parallel HCECC (line 13).

Fig. 4.9 shows how the instances in the CFB are connected. Intuitively, the events (inputs and outputs) of the main interface $I$ connect to corresponding events of the interface of each of the FB instances. For example: $A \mapsto \text{ParallelECC1}.A$ and $\text{ParallelECC2}.Z \mapsto Z$. All FB instances read the input variables of the CFB interface. As only one output variable can be connected to each output variable of the CFB interface, only the last FB instance in FBs (according to the order of the set of ECCs in the original HCECC) is connected. Moreover, internal and output variables of the
original HCECC are propagated between instances by connecting the output version of these variables from each FB instance to the corresponding input version of the FB instance that will be executed next. For example connections: `ParallelECC1.Output` $\rightarrow$ `ParallelECC2.Output` and `ParallelECC2.Output` $\rightarrow$ `ParallelECC1.Output` as shown in Fig. 4.9. An event connection, such as `ParallelECC1.PARO` $\rightarrow$ `ParallelECC2.PARI` and `ParallelECC2.PARO` $\rightarrow$ `ParallelECC1.PARI` is used to force each FB instance to re-sample the connected internal and output variables in each tick. This enables subsequent FBs to read the latest variable values during execution, and the first FB instance will read the values emitted by the last FB instance at the end of the previous tick.

Algorithm CreateBFN (Alg. 4.3), which creates a new basic function block and FB instance from a parallel ECC, is now discussed in detail. First, an interface $I'$ is created for the new BFB $BFB'$ (lines 1-6). $I'$ has the same input and output events as the interface $I$ of the parallel ECC (allowing it to read/write all input and output events of $I$) as well as a new input event `PARI` and output event `PARO` (line 2) (more details appear later). $I'$ also has the same input and output variables, but it also contains: an input copy of each internal or output variable and an output copy of each output variable (line 3). Note that the input and output associations of $I'$ preserve the associations between events and variables of the original HCECC interface $I$ (line 5), as shown in Fig. 4.9 (the associations are shown explicitly within the CFB to aid readability).

The two additional events (`PARI` and `PARO`) are used to force sampling of interface variables at the start and end of a execution respectively. `PARI` is associated with the input copies of internal and output variables (line 5). Similarly, `PARO` is associated with the output copies of internal and output variables (line 5). The communication between the original parallel ECCs is recreated between FB instances by forcing `PARO` to be emitted in each tick (by modifying the ECC as described later), and connecting `PARO` (and associated variables) to `PARI` (and associated variables) of the instance that will execute next. Thus, in each tick, each instance samples the internal and output variables at the start of its execution and transfers the variables to the interface of the next instance.

The interfaces for each FB instance can be seen in Figure 4.9, where each new basic function block has the same interface. Instances and BFBs are named based on the name of the ECC in the parallel HCECC in Fig. 4.8. For example, instance `ParallelECC1` of a new BFB `ParallelEg3_ParallelECC1` in Fig. 4.9 corresponds to the ECC `ECC1` in Fig. 4.8. The input and output events of the original interface (Fig. 4.8) ($A, B$ and $Y, Z$ respectively) remain, as well as the additional input and output events `PARI` and `PARO`. The interfaces have input variables corresponding to all input, internal (`Internal1` and `Internal2`) and output variables (`Output`) of the original HCECC. In addition, the interfaces have output variables correspond to all local variables (`Internal1` and `Internal2`)
4.2 Hierarchical and Concurrent ECCs

Input: FB interface $I = \langle E_I, V_I, \alpha_I, E'_O, V'_O, \alpha'_O \rangle$, Locals $L = \langle V_L, A'_L \rangle$, ECC $ECC^I, L^I = \langle S, s_0, \lambda, T \rangle$

Output: BFB instance $fb$

1. //Create interface;
2. Set $E_I = E'_I \cup \{\text{PARI}\}$; Set $E_O = E'_O \cup \{\text{PARI}\}$;
3. Set $V_I = V'_I \cup V_L \cup V'_O$; Set $V_O = V_L \cup V'_O$;
4. // Maintain original associations, adding: PARI associated with input copies of internal and output variables, and PARI associated with all output variables;
5. $\alpha_I = \alpha'_I \cup \{(\text{PARI}, v_I)|v_I \in V_L \cup V'_O\}$; $\alpha_O = \alpha'_O \cup \{(\text{PARI}, v_O)|v_O \in V'_O\}$;
6. $I' = \langle E_I, V_I, \alpha_I, E_O, V_O, \alpha_O \rangle$;
7. //Create locals;
8. $L^I = \langle \emptyset, A'_L \cup \{\text{UpdateOutputs}\} \rangle$;
9. //Create ECC;
10. Set $S' = S$;
11. Set $s'_0 = s_0$;
12. Create empty action function $\lambda'$ and transition function $T'$;
13. for each $s \in S$ do
14. // Make each state execute UpdateOutputs and emit PARI;
15. $\lambda'(s) = \lambda(s) \oplus \{\text{UpdateOutputs}, \text{PARI}\}$;
16. for each transition $t = s \overset{e,b}{\rightarrow} s_1$ in $T$ do
17. $T'(s) = T(s) \oplus \{t\}$;
18. end
19. //Loop states creation;
20. if there is no transition $s \rightarrow s_1$ then
21. Mark $s_I = s$; // Use $s_I$ to refer to $s$;
22. if $\lambda(s)$ is not empty then
23. Create new loop state $s_I$; $S' = S' \cup \{s_I\}$; // Make $s_I$ refer to a new state;
24. $\lambda'(s_I) = \{\text{UpdateOutputs}, \text{PARI}\}$;
25. for each transition $t = s \overset{e,b}{\rightarrow} s_1$ in $T(s)$ do
26. $T'(s_I) = T'(s_I) \cup \{s_I \overset{e,b}{\rightarrow} s_1\}$;
27. end
28. end
29. $T'(s) = T'(s) \oplus \{s \overset{1}{\rightarrow} s_1\}$; // Create a transition from $s$ to $s_I$ (where $s_I$ refers to $s$ or a new state);
30. end
31. end
32. Create ECC $ECC'^I, L'^I = \langle S', s'_0, \lambda', T' \rangle$;
33. Create Basic FB $\text{BFB}' = \langle I', L'^I, \text{ECC}' \rangle$;
34. Create FB instance $fb = (\text{name}, \text{BFB}')$ where name is a unique string;
35. return $fb$;

Algorithm 4.3: Algorithm CreateBFB. Creation of BFB instance for a parallel ECC.
and output variables (Output) of the original HCECC.

Next in Alg. 4.3, the local declarations \( L^T = \langle V'_L, A^T_L \rangle \) of BFB' are created (line 8). There are no internal variables in the new basic function blocks as the internal variables of the parallel HCECC have been converted into input and output variables previously (on line 3). \( A^T_L \) contains all of the algorithms in the parallel HCECC (\( A^T_L \)), as well as a new algorithm \( \text{UpdateOutputs}^{1} \). \( \text{UpdateOutputs} \) is a simple algorithm that is used to update the value of every output variable in \( T' \) after every tick. It is used to copy the input internal and output variables to the corresponding output variables. Regardless of whether or not \( \text{ECC}^{T,L} \) modifies these internal or output variables, by executing \( \text{UpdateOutputs} \) at the end of each tick the output copies of each variable are updated in order to be propagated to other instances.

Finally, Alg. 4.3 creates the ECC \( \text{ECC}^{T',L'} = \langle S', s'_0, \lambda', T' \rangle \) for BFB' (lines 9–32). E.g., for the parallel HCECC shown in Fig. 4.8, Fig. 4.10 shows the original ECC \( \text{ECC} \) (Fig. 4.10(a)) and the corresponding ECC created by Alg. 4.3 (Fig. 4.10(b)). The algorithm proceeds as follows. Initially, the set of states \( S' \) contains all states of the original ECC \( \text{ECC}^{T',L'} \) (line 10) and the initial state is set to be the initial state of \( \text{ECC}^{T',L'} \) (line 11).

Next, the action functions and transition functions \( \lambda' \) and \( T' \) are initialized (line 12) and populated using the for loop on lines 13–31. Here, each state \( s \) in the original ECC \( \text{ECC}^{T',L'} \) is processed as follows. Firstly, the action map \( \lambda' \) for \( s \) is modified to contain the same sequence of algorithms and outputs as in \( \lambda \) (line 15). In addition, the execution of the algorithm \( \text{UpdateOutputs} \) and the emission of the output \( \text{PARO} \) is appended to this sequence (line 15). This is done to ensure that all output variables are updated (by executing \( \text{UpdateOutputs} \)) by the new ECC whenever it executes. For example, \( \lambda'(S1) = \{Y, \text{UpdateOutputs}, \text{PARO}\} \). Also, \( s \) contains the same transitions in \( T' \) as in the original ECC (lines 16–18).

In lines 19–30, the algorithm checks if a complementary loop state needs to be created for \( s \). A loop state is added to ensure that a transition is always enabled in the new ECC, each added loop state is a copy of some original state but without the original action map. This is because the new ECC must execute \( \text{UpdateOutputs} \) and emit \( \text{PARO} \) to ensure that the values of the input variables are copied to corresponding output variables at the end of each tick. If \( s \) has no always-true (or default) transition (line 20), then a loop state is needed to handle the case where no existing transition may be taken in a tick. If the action map of \( s \) (\( \lambda(s) \)) in the original ECC is not empty, then a new loop state \( s_l \) is created (line 24) and added to the set of states \( S' \). This loop state executes \( \text{UpdateOutputs} \) and emit \( \text{PARO} \) whenever it is entered (line 24). The loop state \( s_l \) has the same transitions as \( s \) (lines 25–27). In case a loop state is not needed (\( s \)'s action map

\(^1\)\( \text{UpdateOutputs} \) contains assignments of the form \( v_O = v_I \) for every \( v_O \) is an output variable of the interface \( T' \) created from an interface output variable or an internal variable. \( v_I \) is the matching input variable of the interface \( T' \) created from an interface output variable or an internal variable.
was originally empty), then \( s \) itself becomes the loop state. Finally, \( s \) is modified to have an extra always-true transition to the loop state (line 29). For example, \( S1 \xrightarrow{1} S1_P \) as in Fig. 4.10(b).

After all of the above computations are done, the new ECC is constructed on line 32 and the corresponding basic function block is created on line 33. Finally, a new instance \( fb \) is created and returned by Alg. 4.3 in lines 34 and 35 respectively.

![Original ECC1](a) Original ECC1  
![ECC within ParallelECC1](b) ECC within ParallelECC1

**Figure 4.10:** Modification of Parallel ECCs

Algorithm 4.4 describes how event connections are created for the set of FB instances FBs. Each FB instance \( fb_i \) in this set is processed as follows. Firstly, input events of the main interface \( I \) are connected to the corresponding input events of \( fb_i \) (lines 6–8). Similarly, output events of the main interface \( I \) are connected to the corresponding output events of \( fb_i \) (lines 10–12). Finally, for the output \( PARO \) of \( fb_i \) is connected to the input \( PARO \) of \( fb_j \), where \( j = i + 1 \) if \( fb_i \) is not the last fb instance in FBs. If \( fb_i \) is the last fb instance in FBs, then \( fb_j \) is the first instance in FBs. The connections between \( PARO \) and \( PARI \) create a cycle of event connections from the first instance to the last instance (and back) which allows the propagation of updated variable data (as explained in the description of Alg. 4.5 later). For example, in Figure 4.9 \( PARO \) of instance ParallelECC1 is connected to \( PARI \) of instance ParallelECC2 and \( PARO \) of instance ParallelECC2 is connected to \( PARI \) of instance ParallelECC1.

Algorithm 4.5 describes how variable connections are created for the set of FB instances (FBs). Each FB instance \( fb_i \) in this set is processed as follows. Firstly, input variables of the main interface \( I \) are connected to the corresponding input variables of \( fb_i \) (lines 6–8). Only the output variables of the last instance \( fb_n \) are connected to the corresponding output variables of the main interface \( I \) (lines 10–14). For example, Output of instance ECC2 is connected to Output of the interface.
Input: Ordered set of FB instances $FBs$, FB interface $I$
Output: Set of event connections $C_{efb}$
1. $C_{eI2I} = \emptyset$; $C_{eO2O} = \emptyset$; $C_{ePAR} = \emptyset$;
2. $n = |FBs|$;
3. for each $i \in [1, n]$ do
   4. Select the $i$-th element $fb_i \in FBs$;
   5. // input event connections;
   6. for each $e_I \in E_I^I$ do
      7. $C_{eI2I} = C_{eI2I} \cup \{(e_I \mapsto fb_i, e_I)\}$;
   8. end
   9. // output event connections;
   10. for each $e_O \in E_O^O$ do
      11. $C_{eO2O} = C_{eO2O} \cup \{(fb_i, e_O \mapsto e_O)\}$;
   12. end
   13. // PARI to PARO connection;
   14. if $i < n$ then
      15. $fb_j = fb_{i+1}$;
   16. else
      17. $fb_j = fb_i$;
   18. end
   19. $C_{ePAR} = C_{ePAR} \cup \{(fb_i, PARO \mapsto fb_j, PARI)\}$;
20. end
21. return $C_{eI2I} \cup C_{eO2O} \cup C_{ePAR}$;

Algorithm 4.4: Algorithm CreateEConns

Finally, the output variables of $fb_i$, corresponding to the internal and output variables of the original HCECC, are connected to the corresponding input variables of the next instance $fb_j$ (lines 15-27). Here, where $j = i + 1$ if $fb_i$ is not the last $fb$ instance in $FBs$. If $fb_i$ is the last $fb$ instance in $FBs$, then $fb_j = fb_1$ (the first instance in $FBs$). For example, the $Internal1$ output of instance $ParallelECC1$ is connected to in $Internal1$ input of instance $ParallelECC2$, and similarly the $Internal1$ output of instance $ParallelECC2$ is connected to in $Internal1$ input of instance $ParallelECC1$.

Summary

Table 4.1 shows an overview of how two parallel ECCs in a parallel HCECC are executed as an equivalent CFB. The input and output sampling of the interface (steps 1 and 6) are the same for both implementations. Steps 2-5 show how each ECC or FB instance is executed. The first ECC ($ECC_1$) or FB instance ($fb_1$) reads the current input values from the interface. Internal and output variables are accessed directly by $ECC_1$, but through connections by $fb_1$. In both cases, the values of the variables are as they were at the end of the previous tick. $ECC_1$ or $fb_1$ is executed (step 3), where $fb_1$ has the same behaviour as $ECC_1$ but will end by also emitting $PARO$ and internal and output variables. The second ECC ($ECC_2$) or FB instance ($fb_2$) executes similarly, except the values of internal
4.2 Hierarchical and Concurrent ECCs

**Input:** Ordered set of FB instances $\text{FBs}$, FB interface $I$, Locals $L^I$

**Output:** Set of variable connections $C_{vfb}$

1. $C_{vI} = \emptyset$; $C_{vO2O} = \emptyset$; $C_{vPAR} = \emptyset$;
2. $n = |\text{FBs}|$;
3. for each $i \in [1, n]$ do
   4. Select the $i$-th element $fb_i \in \text{FBs}$;
   5. // interface input connections;
   6. for each $v_I \in V^I_{I}$ do
      7. $C_{vI} = C_{vI} \cup \{(v_I \rightarrow fb_i.v_I)\}$;
   8. end
   9. // interface output connections (only from last instance);
   10. if $i = n$ then
       11. for each $v_O \in V^O_{O}$ do
           12. $C_{vO2O} = C_{vO2O} \cup \{(fb_i.v_O \rightarrow v_O)\}$;
       13. end
   14. end
   15. if $i < n$ then
       16. $fb_j = fb_{i+1}$;
   17. else
       18. $fb_j = fb_1$;
   19. end
   20. // internal variable connections;
   21. for each $v_L \in V^L_{I}$ do
       22. $C_{vPAR} = C_{vPAR} \cup \{(fb_i.v_L \rightarrow fb_j.v_L)\}$;
   23. end
   24. // output variable connections;
   25. for each $v_O \in V^O_{O}$ do
       26. $C_{vPAR} = C_{vPAR} \cup \{(fb_i.v_O \rightarrow fb_j.v_O)\}$;
   27. end
   28. end
29. return $C_{vI} \cup C_{vO2O} \cup C_{vPAR}$;

**Algorithm 4.5:** Algorithm CreateVConns
<table>
<thead>
<tr>
<th>Execution Steps</th>
<th>Parallel HCECC</th>
<th>Equivalent CFB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Sample inputs of interface.</td>
<td>1. Interface samples input events and associated variables.</td>
<td>1. CFB interface samples input events and associated variables.</td>
</tr>
<tr>
<td>2. Identify inputs for first component.</td>
<td>2. The first ECC (ECC(_1)) reads new input events and variables and directly reads internal and output variables as they were at the end of the previous tick.</td>
<td>2. The first BFB instance (fb(_1)) reads new input events and variables and reads internal and output variables connected from the second BFB instance (fb(_2)) as they were at the end of the previous tick.</td>
</tr>
<tr>
<td>3. Execute first component, producing outputs.</td>
<td>3. ECC(_1) is executed, possibly modifying internal and output variables.</td>
<td>3. fb(_1) is executed, possibly modifying internal and output variables, and emits PARO and output copies of all internal and output variables.</td>
</tr>
<tr>
<td>4. Identify inputs for second component.</td>
<td>4. The second ECC (ECC(_2)) reads new input events and variables and directly reads internal and output variables as they were modified by (ECC(_1)).</td>
<td>4. The second BFB instance (fb(_2)) reads new input events and variables and the latest internal and output variables connected from the first BFB instance (fb(_1)).</td>
</tr>
<tr>
<td>5. Execute second component, producing outputs.</td>
<td>5. ECC(_2) is executed, possibly modifying internal and output variables.</td>
<td>5. fb(_2) is executed, possibly modifying internal and output variables, and emits PARO and output copies of all internal and output variables.</td>
</tr>
<tr>
<td>6. Sample outputs at interface.</td>
<td>6. Interface samples output events and associated variables.</td>
<td>6. CFB interface samples output events and associated variables.</td>
</tr>
</tbody>
</table>

Table 4.1: CFB Equivalence of Parallel HCECC semantics

and output variables have just been processed by ECC\(_1\) or fb\(_1\) respectively.
4.2 Hierarchical and Concurrent ECCs

The conversion of a parallel HCECC to a composite function block ensures that parallel HCECCs are compliant to the IEC 61499 standard. The CFB transformation introduces some execution overhead due to the additional interfaces that must be processed. In addition, each ECC has additional states and transitions, noting that there is a linear increase in the number of states and transitions. However, the manual design of a CFB is more complicated than the equivalent parallel HCECC, and therefore more likely to introduce bugs requiring more time to debug. The execution overhead is therefore minor compared to the design-time simplification afforded by a parallel HCECC. In addition, the execution overhead can be mitigated by directly compiling HCECCs for execution, as shown in the compilers presented in chapter 6.

4.2.2 Refined HCECCs

Refinement allows a set of one or more parallel refining ECCs to describe the behaviour of a single state of an ECC. This introduces a notion of behavioural hierarchy within a basic function block, that is present in other aspects of the IEC 61499 standard (such as CFBs). Such hierarchy allows a high-level behavioural description to be refined, such that within any of its states there can be further (refining) ECCs. A refined state in a refined ECC in a function block is shown to be equivalent to a set of parallel ECCs, where the first refined ECC communicates with the later refining ECCs to enable and disable their behaviour. It is also possible for multiple states in an ECC to be refined in this way. A definition for a refined HCECC follows.

**Definition 6 (Refined HCECC).** $\text{HCECC}_s = (\mathcal{I}, L^I, \text{ECC}^{\mathcal{I}, L^I}, \text{ECCs}, \triangleright)$ is a **refined** HCECC, where $\text{ECC}^{\mathcal{I}, L^I}$ is the **refined** ECC and $\text{ECCs} = \{\text{ECC}_1^{\mathcal{I}, L^I}, \ldots, \text{ECC}_m^{\mathcal{I}, L^I}\}$ is a set of $n$ refining ECCs. $\text{ECCs}$ is partitioned into $m$ ordered and non-empty sets $\text{PARTS} = \{\text{ECC}_{s_1}, \ldots, \text{ECC}_{s_m}\}$ where $1 \leq m \leq \max(|S|)$ ($S$ is the set of states of $\text{ECC}^{\mathcal{I}, L^I}$). $\triangleright$ is the state-refinement function that maps $m$ states of $\text{ECC}^{\mathcal{I}, L^I}$ to unique partitions $\text{ECCs}_i$ of $\text{ECCs}$. If $\triangleright(s) = \text{ECCs}_i$ for some $s \in S$, then $\text{ECCs}_i \in \text{ECCs}$ is said to refine state $s$.

Fig. 4.11 shows a refined HCECC $\text{HCECC}_s = (\mathcal{I}, L^I, \text{ECC}^{\mathcal{I}, L^I}, \text{ECCs}, \triangleright)$ where: $\text{ECCs} = \{\text{Para1, Para2}\}$, $\text{PARTS} = \{\{\text{Para1, Para2}\}\}$, and $\triangleright(\text{RefinedState}) = \{\text{Para1, Para2}\}$. This means that parallel ECCs Para1 and Para2 refine the behaviour of the state $\text{RefinedState}$ in the main ECC. Note that each of the refining ECCs operate using the events and variables of the HCECC (from the interface $\mathcal{I}$ and local declarations $L^I$).

$^2\text{ECCs} = \text{ECCs}_1 \oplus \ldots \oplus \text{ECCs}_m$, and for all $i, j \in [1, m]$, if $i \neq j$ then $\text{ECCs}_i \cap \text{ECCs}_j = \emptyset$. 

Execution of Refined HCECCs

A refined HCECC $\text{HCECC}_r = \langle \mathcal{I}, L^T, \text{ECC}^T, L^T, \text{ECCs}, \triangleright \rangle$ executes as follows:

1. Interface inputs are sampled, i.e. input events and associated input variables.

2. The refined ECC $(\text{ECC}^T, L^T)$ is executed just as a standard ECC. If a transition is taken out of a refined state $s$ (for which $\triangleright(s)$ is defined), the refining ECCs are not executed in this tick, and stop being executed until $s$ is re-entered. This results in the pre-emption of the lower-level refining ECCs, due to a higher-level transition being taken.

3. If the refined ECC is in a refined state $s$, the parallel ECCs in $\triangleright(s)$ are executed in a fixed order.

4. Finally, interface outputs (events and variables) are emitted and the tick completes.

A refined HCECC can be resolved into an equivalent parallel HCECC. As shown in the previous section, parallel HCECCs are IEC 61499 standard-compliant since they can always be resolved into composite function blocks. Hence, the conversion from a refined to a parallel HCECC (and eventually to a composite function block) ensures that all refined HCECCs are also standard-compliant. The transformation from a refined HCECC into an equivalent parallel HCECC is now formally presented.

Refinement to Parallel Conversion

Initially, to simplify discussion, an algorithm is presented that removes the refinement from a refined ECC assuming only one state is refined. Algorithm RefineToPar (Alg. 4.6) creates a set of parallel ECCs with local declarations in which the refined ECC is executed first, followed by the refining ECCs. The refined ECC is modified to set a variable that will enable and disable the behaviour of refining ECCs, making each refining ECC transition to a Disabled state until re-enabled.
4.2 Hierarchical and Concurrent ECCs

Given a FB interface $I$, local declarations $L^I$, a refined ECC $ECC^{|I|}$, a refined state $s$ in that ECC, and the refining ECCs $ECC_s$, Alg. 4.6 returns the equivalent set of parallel ECCs $ECC_{sb}$ and modified local declarations $L_{sb}^I$. For the example in Figure 4.11 the algorithm produces the parallel ECCs shown in Figure 4.12.

![Figure 4.12: Parallel ECCs equivalent to refinement in Figure 4.11](image)

Lines 1–5 of Alg. 4.6 modify the local declarations, adding a new internal variable ($s_b\_Disabled$) on line 2 and two new algorithms on line 3. The new algorithms ($s_b\_Enable$, $s_b\_Disable$) clear and set the $s_b\_Disabled$ variable respectively.

The first step in constructing the set $ECC_{sb}$ is to modify the main ECC (lines 8–22). This involves adding the algorithm $s_b\_Enable$ to the actions of $s_b$ (line 13) so that whenever state $s_b$ is reached, the variable $s_b\_Disabled$ is cleared (allowing the refining ECCs to execute). Similarly, for every state $s$ to which $s_b$ has an outgoing transition, the respective action map is modified to add the algorithm $s_b\_Disable$ (line 16). This ensures that the variable $s_b\_Disabled$ is set when state $s_b$ is exited in the main ECC (forcing the refining ECCs to stop executing). For example, for the parallel ECCs in Figure 4.12, the action map of the first ECC has been changed to the following:

- $\lambda_0(\text{Start}) = \{\text{RefinedState\_Disable}\}$
- $\lambda_0(\text{RefinedState}) = \{\text{RefinedState\_Enable}, X\}$, where the emission of output event $X$ was part of the original refined HCECC.

The main ECC, modified as above, is inserted into the set $ECC_{sb}$ as its first element (line 23). This allows the execution of this ECC to enable and disable execution of the refining ECCs.

Lines 24–43 modify each refining ECC (in $ECC_s$), adding a new state (Disabled) (line 26) that is used disable the standard behaviour of the ECC when the refined ECC is not in state $s_b$. This new state can be seen in the ECCs in Figure 4.12 that were originally
Input: $I, L^I = (V_L, A_L^I), \text{ECC}_{I,L^I} = (S, s_0, \lambda, T), s \in S, \text{ECC}_{s}$

Output: Pair $(\text{ECC}_{s^0}, L^I_0)$

1. //Modify locals;
2. Variable set $V_L' = V_L \cup \{s_{\text{\_Disabled}}\}$ ($s_{\text{\_Disabled}}$ is of type Boolean);
3. Algorithm set $A_L'^I = A_L^I \cup \{s_{\text{\_Enable}}, s_{\text{\_Disabled}}\}$;
4. //Create new local declarations;
5. $L^I_0 = (V_L', A_L'^I)$;
6. //Create ECCs;
7. Initialize ordered set of ECCs $\text{ECC}_{s} = \emptyset$;
8. //Modify $\text{ECC}_{I,L^I}$;
9. Initialize empty action map $\lambda_0$;
10. //Modify states;
11. for each state $s \in S$ do
12. if $s = s_0$ then
13. $\lambda_0(s) = \lambda(s) \oplus \{s_{\text{\_Enable}}\}$;
14. end
15. else if $s_{\text{\_has a transition (in T) to s}}$ then
16. $\lambda_0(s) = \lambda(s) \oplus \{s_{\text{\_Disabled}}\}$;
17. end
18. else
19. $\lambda_0(s) = \lambda(s)$;
20. end
21. end
22. $\text{ECC}^I_{0,L^I} = (S, s_0, \lambda_0, T)$;
23. $\text{ECC}_{s} = \text{ECC}_{s} \oplus \{\text{ECC}_0\}$;
24. //Modify ECCs in $\text{ECC}_{s}$;
25. for each $\text{ECC}^I_{s}, L^I_s \in \text{ECC}_{s}$ do
26. Create set of states $S_{s_0} = S_s \cup \{\text{Disabled}\}$;
27. Set state $s_{s_0} = \text{Disabled}$;
28. if $s_{s_0} = s_0$ then
29. $s_{s_0} = s_0$;
30. end
31. Initialize action map $\lambda_{s_0} = \lambda_0$;
32. Add $\lambda_{s_0}(\text{Disabled}) = \emptyset$;
33. Initialize transition function $T_{s_0} = T_0$;
34. for each $s \in S_{s_0}$ do
35. if $s = \text{Disabled}$ then
36. $T_{s_0}(s) = \{s \xrightarrow{1,s_{s_0}} \text{Disabled}, T_0 \}$;
37. else
38. $T_{s_0}(s) = \{s \xrightarrow{1,s_{s_0}} \text{Disabled} \} \oplus T_0(s)$;
39. end
40. end
41. Create ECC $\text{ECC}_{s_0} = (S_{s_0}, s_{s_0}, \lambda_{s_0}, T_{s_0})$;
42. $\text{ECC}_{s} = \text{ECC}_{s} \oplus \{\text{ECC}_{s_0}\}$;
43. end
44. return Pair $(\text{ECC}_{s}, L^I_0)$;

Algorithm 4.6: Algorithm RefineToPar. Remove refinement from a single refined state creating a parallel HCECC.
refining ECCs (Para1 and Para2). The initial state of a refining ECC is set to be the new Disabled state (line 27). Although, if the refined state \( s_0 \) is the initial state of the refined ECC, then each refining ECC needs to be initialised to be enabled (line 29 sets the initial state to be the original initial state of the refining ECC in this case).

The action map for the modified refining ECC is initialised to be the same as the original action map for the refining ECC (line 31). The Disabled state is given an empty set of actions (line 32).

The transition function for the modified refining ECC copies the original transition function (line 33). New transitions are added to react to changes in the new local variable \( (s_0\_Disabled) \), entering state Disabled from any other state if true (line 38). Line 36 adds a transition out of the Disabled state, to enter the original initial state of the refining ECC when \( s_0\_Disabled \) is false. For example, all of the original states in ECC Para1 in Figure 4.12 have a new transition to the new Disabled state when the internal variable \( (RefinedState\_Disabled) \) is true. The Disabled state has a single transition to the original initial state S1 when the internal variable is false.

Finally, line 41 creates a new ECC based on the modified set of states, initial state, action map and transition function. This new ECC is appended to the set of equivalent parallel ECCs (ECCs\(_0\)) on line 42. After all refining ECCs have been processed in this way, line 44 returns a pair of the equivalent parallel ECCs (ECCs\(_0\)) and modified local declarations \( (L^2) \).

The general case of a refined HCECC, where \( m \) states are refined, is now presented. The algorithm uses RefineToPar to remove refinement from each refined state one at a time.

Algorithm RemoveRefinement in Alg. 4.7 shows the steps followed to convert a refined HCECC \( (\text{HCECC}_{\_0} = (I, L^2, \text{ECC}^{I}, L^2)) \) into a parallel HCECC. The refined ECC (containing the states: \{Start, RefinedState\} in Figure 4.11) is added to a set ECCs' that will eventually contain the ordered set of ECCs in the resulting parallel HCECC on line 1. Line 2 initialises a copy of the local declarations of the block \( (L^2) \) that will be modified by multiple calls to the RefineToPar algorithm.

Each refined state \( s \) in the refined HCECC, for instance RefinedState in Figure 4.11, is processed on lines 4–17 as follows.

- The refined ECC \( (\text{ECC}_{\_0}) \), is retrieved from the first element of set ECCs' (the refined ECC is modified at every iteration and reinserted to be the first ECC in ECCs'). The set of parallel ECCs that refine the behaviour of state \( s \) are obtained \( (\text{ECCs}_s) \) using the state-refinement function \( (>) \) (line 7).

- RefineToPar (Alg. 4.6) is called to remove the refinement from state \( s \), returning new local declarations and a set of parallel ECCs that includes the refined ECC
Input: Refined HCECC HCECC₀ = \langle I, L^I, ECC^I,L^I \rangle = \langle S, s_0, \lambda, T \rangle, ECCs, \triangleright
Output: Parallel HCECC HCECC∥ = \langle I, L^∥, ECCs' \rangle

1 Initialize ordered set of ECCs ECCs' = \{ECC^I,L^I \};
2 \quad L^∥ = L^I;
3 // Remove refinement from every state;
4 for each refined state s in S do
5 \quad // Use the (possibly modified) refined ECC stored in ECCs';
6 \quad Initialize ECC ECCₐ = first element of ECCs';
7 \quad Eccsₐ = \triangleright(s);
8 \quad Pair (ECCsₐnew, Lₐnew) = RefineToPar(I, L^∥, ECCₐ, s, ECCsₐ);
9 \quad L^∥ = L^I_new;
10 \quad // Replace the refined ECC stored in ECCs' with the recently modified version;
11 \quad Remove ECCₐ from ECCs';
12 \quad ECCₐ = first element of ECCs new;
13 \quad Remove ECCₐ from ECCs new;
14 \quad Insert ECCₐ as the first element of ECCs';
15 \quad // Append other ECCs to the set of parallel ECCs;
16 \quad ECCs' = ECCs' ⊕ ECCs new;
17 end
18 Create parallel HCECC \langle I, L^∥, ECCs' \rangle;
19 return HCECC;

Algorithm 4.7: Algorithm RemoveRefinement. Resolves a refined HCECC into a parallel HCECC.

(with ECCs refining state s now in parallel) (line 8). Line 9 updates the local declarations that will be used in the parallel HCECC.

- Lines 11-11 replace the refined ECC in the set ECCs' with the recently modified refined ECC returned by RefineToPar at the start of ECCs new.
- Finally, the modified refining ECCs returned by RefineToPar are appended to the set of parallel ECCs (line 16).

Once all refined states have been processed, lines 18 and 19 construct and return a parallel HCECC with the newly computed locals (L^∥), set of parallel ECCs (ECCs') and the same FB interface as the given refined HCECC.

Summary

Table 4.2 shows an overview of how a refined HCECC is executed as a parallel HCECC, given one state is refined by two ECCs in parallel. Notice the primary difference: in a refined HCECC the refining ECCs are conditionally executed (step 3), whereas in the equivalent parallel HCECC they are always executed (steps 4-7). This is made possible
as each refining ECCs is modified to conditionally execute the original refining behaviour. The input and output sampling of the interface (steps 1 and 4) are the same for both implementations (steps 1 and 8 for the parallel HCECC). The refined ECC is executed in steps 2-3, reading internal variables last written in the previous tick. If the refined ECC is in the refined state: The refining ECCs are executed in steps 3(a) - 3(d) of the refined HCECC, or executed as normal (with the internal $s_c\_Disabled$ variable set to false) in steps 4-7 of the parallel HCECC. If the refined ECC is not in the refined state: The refined HCECC does nothing, and continues at step 4. The parallel HCECC will still execute each refining ECC (steps 4-7), but they will enter or remain in the Disabled state.

The refinement operator shows that a refined HCECC can be reduced to parallel HCECC in order to guarantee deterministic execution and to be compatible with the IEC 61499 standard. As with the transformation of parallel HCECCs to CFBs, this transformation introduces some execution overhead due to the continuous execution of refining ECCs. Each refining ECC also has a single extra state and an extra transition per state. Importantly, the development of a refined HCECC is much simpler than manually implementing the same behaviour in a parallel HCECC or, more significantly, in a CFB. The execution overhead is therefore minor compared to the design-time simplification afforded by a refined HCECC. The following sub-section describes how HCECCs that contain multiple levels of refinement and concurrency are handled.

4.2.3 Interleaving of parallel and refinement operators

While HCECCs can have arbitrary levels of nested refinement and concurrency, the definition of a refined HCECC (Def. 6) only accounts for a single level of refinement. In this section, an algorithm is presented that allows nested refinement and parallel operators to be resolved.

Figure 4.13 shows an HCECC with nested refinement and concurrency. The algorithm FlattenHCECC in Alg. 4.8 is used to remove nested refinement. It reads a refined HCECC and a map $MAP_c$, which maps some ECC in ECCs$_b$ to a pair $(ECCs_i, \varphi_i)$ such that $(\mathcal{I}, L^T, ECC_i, ECCs_i, \varphi_i)$ is a refined HCECC. After execution, FlattenHCECC returns a refined HCECC with no nested refinement (such as in Figure 4.14).

FlattenHCECC starts by creating a copy of the local declarations on line 1. ECC$^{\flat}$ on line 2 is a set that will eventually contain an updated set of ECCs refining the top ECC (ECC$^{T,L^T}$). Line 3 initialises a new state-refinement function for the new refined HCECC.

Each state in ECC$^{T,L^T}$ is looped through (lines 4 - 19), and checked for refinement (line 7). If $s$ is refined, and one or more refining ECCs are refined further, the nested refinement is removed by a recursive call to FlattenHCECC followed by a call to RemoveRefinement. For example, when ECC R2ECC1 of Fig. 4.13 is found to be re-
<table>
<thead>
<tr>
<th>Execution Steps</th>
<th>Refined HCECC</th>
<th>Equivalent HCECC</th>
<th>Parallel HCECC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Sample inputs of interface.</td>
<td>1. Interface samples input events and associated variables.</td>
<td>1. Same as Refined HCECC.</td>
<td></td>
</tr>
<tr>
<td>2. Identify inputs for refined ECC.</td>
<td>2. Refined ECC ($\text{ECC}_o$) reads new input events and variables and directly reads internal and output variables as they were at the end of the previous tick.</td>
<td>2. Same as Refined HCECC.</td>
<td></td>
</tr>
<tr>
<td>3. Execute refined ECC, producing outputs. If the refined ECC is in the refined state:</td>
<td>3. $\text{ECC}_o$ is executed, possibly modifying internal and output variables. If $\text{ECC}_o$ is in the refined state:</td>
<td>3. Same as Refined HCECC. But the refining ECCs in the following steps (4-7) are unconditionally executed.</td>
<td></td>
</tr>
<tr>
<td>3. (a) Identify inputs for first refining ECC.</td>
<td>3. (a) The first refining ECC ($\text{ECC}_1$) reads new input events and variables and directly reads internal and output variables as they were modified by ($\text{ECC}_o$).</td>
<td>4. Same as Refined HCECC.</td>
<td></td>
</tr>
<tr>
<td>3. (b) Execute first refining ECC.</td>
<td>3. (b) $\text{ECC}_1$ is executed, possibly modifying internal and output variables.</td>
<td></td>
<td>5. $\text{ECC}_1$ is executed, but if the refined ECC is not in the refined state it will either transition to or stay in state Disabled.</td>
</tr>
<tr>
<td>3. (c) Identify inputs for second refining ECC.</td>
<td>3. (c) The second refining ECC ($\text{ECC}_2$) reads new input events and variables and directly reads internal and output variables as they were modified by ($\text{ECC}_1$).</td>
<td></td>
<td>6. Same as Refined HCECC.</td>
</tr>
<tr>
<td>3. (d) Execute second refining ECC.</td>
<td>3. (d) $\text{ECC}_2$ is executed, possibly modifying internal and output variables.</td>
<td></td>
<td>7. $\text{ECC}_2$ is executed, but if the refined ECC is not in the refined state it will either transition to or stay in state Disabled.</td>
</tr>
<tr>
<td>4. Sample outputs at interface.</td>
<td>4. Interface samples output events and associated variables.</td>
<td></td>
<td>8. Interface samples output events and associated variables.</td>
</tr>
</tbody>
</table>

Table 4.2: Parallel HCECC Equivalence of Refined HCECC semantics
fined within state R2, line 5 creates a new ordered set of ECCs for the state and line 9 gets the corresponding refined HCECC. Line 10 removes any further nested refinement by the recursive call to FlattenHCECC, and RemoveRefinement returns the equivalent parallel ECCs and modified local declarations. The modified local declarations replace the existing copy of local declarations ($L^T_{flat}$), so that multiple refined states each extend the same local declarations (according to RefineToPar. The parallel ECCs, which do not contain any further refinement, are appended to the new set of refining ECCs for the state ($ECCs_r$) on line 11. If the ECC is not refined, then it is simply appended to $ECCs_r$ (line 14).

Once all refining ECCs have been processed, line 17 updates the state-refinement function for the current state ($\triangleright_{flat}(s)$). The new ECCs are also added to $ECCs_{flat}$ on line 18. Once all states have been processed, a new refined HCECC is created on line 20 and returned on line 21. Because of the recursive calls to FlattenHCECC, given the HCECC in Fig.4.13, the algorithm will ultimately produce a refined HCECC with:
Input: Refined HCECC \( (I, L^I, \text{ECC}^{I,L^I} = \langle S, s_0, \lambda, T \rangle, \text{ECCs}, \triangleright) \), \( \text{MAP}_r \) (a map from an ECC \( \text{ECC}_i \in \text{ECCs} \) to a set of parallel ECCs (\( \text{ECCs}_i \)) and a refinement operator \( (\triangleright_i) \) such that \( (I, L^I, \text{ECC}_i, \text{ECCs}_i, \triangleright_i) \) is a refined HCECC.

Output: Flattened refined HCECC \( (I, L_{\text{flat}}^I, \text{ECC}^{I,L_{\text{flat}}^I}, \text{ECCs}_{\text{flat}}, \triangleright_{\text{flat}}) \)

1. \( L^I_{\text{flat}} = L^I \);
2. \( \text{ECCs}_{\text{flat}} = \emptyset \);
3. Initialize \( \triangleright_{\text{flat}} \);
4. foreach refined state \( s \in S \) do
   5. Ordered set of ECCs \( \text{ECCs}_s := \emptyset \);
   6. foreach \( \text{ECC}_i \in \triangleright(s) \) do
      7. if \( \text{MAP}_r(\text{ECC}_i) \) is not null then
         8. \( (\text{ECCs}_i, \triangleright_i) = \text{MAP}_r(\text{ECC}_i) \);
         9. Construct refined HCECC \( \text{HCECC}_{\text{ECCs}_i} = \langle I, L_{\text{flat}}^I, \text{ECC}_i, \text{ECCs}_i, \triangleright_i \rangle \);
         10. Parallel HCECC \( \langle I, L_{\text{flat}}^I, \text{ECCs} \rangle = \text{RemoveRefinement}(\text{FlattenHCECC}(\text{HCECC}_{\text{ECCs}_i}, \text{RefinedHCECCMap})) \);
         11. \( \text{ECCs}_s = \text{ECCs}_s \oplus (\text{ECCs}_i) \);
      end
   else
      12. \( \text{ECCs}_s = \text{ECCs}_s \oplus \text{ECC}_i \);
   end
5. Set \( \triangleright_{\text{flat}}(s) = \text{ECCs}_s \);
6. \( \text{ECCs}_{\text{flat}} = \text{ECCs}_{\text{flat}} \cup \text{ECCs}_s \);
end
19. Construct flattened HCECC \( \text{HCECC}_{\text{ECCs}_{\text{flat}}} = \langle I, L_{\text{flat}}^I, \text{ECC}^{I,L_{\text{flat}}^I}, \text{ECCs}_{\text{flat}}, \triangleright_{\text{flat}} \rangle \);
20. return \( \text{HCECC}_{\text{ECCs}_{\text{flat}}} \);

Algorithm 4.8: Algorithm FlattenHCECC.

\( \triangleright_{\text{flat}}(R^2) = \{ R^2 \text{ECC}1, S^2 \text{ECC}1, R^2 \text{ECC}2, T^2 \text{ECC}1, T^2 \text{ECC}2 \} \).

Each ECC will have been modified from the original ECCs of the same names according to one or more iterations of the refinement operator.

Figure 4.15 shows the parallel HCECC that is the result after the refined HCECC returned by FlattenHCECC (Fig. 4.14) is processed by RemoveRefinement to remove the final level of hierarchy. All ECCs that had refined states have actions added to enable and disable the refining ECCs. Similarly, all refining ECCs have a disabled state that is used whenever the refined state is exited. Because of the fixed order of execution defined by the parallel operator, a refining ECC only needs to react to the deactivation of the state it is refining. For example, if the HCECC is in state \( \mathcal{W}1 \) when event \( A \) happens:

- The refinement of \( R^2 \) will be disabled, disabling ECCs: \( R^2 \text{ECC}1 \) and \( R^2 \text{ECC}2 \).

- This in turn disables ECC \( S^1 \text{ECC}1 \).
4.3 Discussion

In Annex E.1 of the IEC 61499 draft standard [57], it was proposed that the introduction of Statecharts-like [27] hierarchy and parallelism may be desirable. In this chapter, the presented HCECC framework has been shown offer such features, while remaining compatible with the standard. In this section, HCECCs are compared with some related graphical languages.

4.3.1 Comparison of HCECCs with Related work

Argos [67] and SyncCharts [59] are other synchronous languages with support for parallelism and refinement. However, unlike HCECCs and IEC 61499, they do not offer a block diagram for the design of control and data flow. Both languages also use signals instead of separated events and variables as defined by the IEC 61499 standard. In addition, both languages allow for internal signals which are incompatible with the IEC 61499 standard. The individual operators used by Argos and SyncCharts are also incompatible with the standard as explained below.

For parallel execution, both SyncCharts and Argos offer synchronous parallel, in which all concurrent actions are interleaved to find a sequential and causal order. It is appealing because it allows the user to design parts of a program as if they were truly concurrent, but compile the program into sequential code. It also provides instantaneous broadcast between parallel state-machines. However, this logical concurrency requires costly causality analysis [28] to ensure a causal sequential representation can be found. Causality analysis can be time consuming and may result in the rejection of certain designs which is not possible anywhere else in the IEC 61499 standard. In addition, it would also result in a state-machine that reacts to multiple simultaneous events, and thus not be compatible with the standard.

SyncCharts also supports two mechanisms for the abortion of refined behaviour:
strong and weak abortion. Strong abortion terminates the behaviour of a refined state at the beginning of a tick, while weak abortion would allow the refining ECC to transition (or execute) once before exiting the state. As there is only one type of transition in IEC 61499, HCECCs are only able to employ one type of abortion. Because the refinement operator of HCECCs executes the refined ECC first in a tick, transitions exiting the refined state act as a strong abort. However, because of the fixed order of execution, the refined ECC will not be re-executed if refined ECCs change the values of variables in a tick. As a result, unlike the strong abort in SyncCharts, transitions out of a refined state act as a strong abort with conditions on internal variables based on values from the previous tick.

The synchronous languages Pret-C [75], SC [76] and Reactive-C [77] also use a fixed-order parallel to avoid causality analysis, while still allowing concurrent behaviours to be described. While Pret-C and Reactive-C only allow a textual definition of behaviour, HCECCs offer both algorithms and graphical components for the specification. SC can be generated from SyncCharts [59], simplifying the implementation, however, only HCECCs are specifically tailored to be compatible with IEC 61499.

UML Statecharts [36] offer many features for high-level modelling, including requirements modelling and software architecture design. The UML standard includes many specification diagrams such as sequence diagrams and class diagrams in addition to UML Statecharts. HCECCs, on the other hand, provide fewer and more consistent design elements that are better suited for system specification. In addition, because UML is a high-level language, it is less capable of modelling a plant in conjunction with a controller. Finally, UML is object-oriented, unlike the component oriented IEC 61499 standard.

Other graphical formalisms such as NCES [66] and CNet [64] are based on tokens and places instead of the finite-state-machines presented by other languages. This difference makes them less desirable, as HCECCs offer a better visual representation for hierarchy and concurrency. Both languages are also non-deterministic, which although beneficial in some applications, is undesirable within a basic function block.

A key benefit of HCECCs is deterministic execution, attributable to the fixed order evaluation of transitions and execution of parallel ECCs. This feature is not guaranteed by SyncCharts and Argos. While the surveyed formal graphical languages offer more features for users, HCECCs uniquely offer a compositional synchronous semantics with hierarchy and concurrency while remaining compatible with the standard.

### 4.4 Conclusions

HCECCs simplify the design of control flow by employing a structured state-machine adopting an Statecharts or Argos-like approach for IEC 61499 ECCs. HCECCs present only two of the operators used by Argos allowing for hierarchical and parallel ECCs, retaining...
the compositionality of Argos. A hierarchy of ECCs can be used to efficiently model exception handling, where a single transition can be used to pre-empt multiple behaviours. Similarly, parallelism simplifies the modelling of simultaneous sub-processes, and allows for monitoring of simultaneous events within the developed synchronous semantics. Features such as these greatly improve the maintainability of function block specifications, allowing multiple function blocks with complicated ECCs to be refactored in a succinct way. In addition, as shown in section 4.2, the use of a single state-machine makes it easier to understand the overall behaviour of the conveyor model component. The semantics remain compatible with the standard and use a syntax similar to IEC 61499 ECCs instead of employing a separate graphical language for handling complex state-machine design.

HCECCs are used in the following chapter to simplify the design of a controller for a baggage handling system. A compiler for the extended HCECC semantics is presented later in Chapter 6, along with experimental comparisons with an earlier synchronous semantics [30] and other execution approaches.
Re-Engineering IEC 61131-3 into IEC 61499 Function Blocks

Many existing IEC 61131-3 applications are large and complicated, representing many man-hours of work, spread over a considerable period of time. Because of this existing investment, a new language is not sufficient to encourage better development practices on its own. As a result, despite newer and more capable standards, there is still widespread use of IEC 61131-3 in a range of industries. The reuse of existing code or architectures would motivate the migration process by reducing the required investment of time and simplifying the changes to the application.

This chapter presents an approach for the re-engineering of IEC 61131-3 specifications into IEC 61499. The term, re-engineering, is the alteration of a system in order to reconstitute it into a new form [78]. This refers to the adoption of parts of existing IEC 61131-3 code and creating a new specification in IEC 61499 that implements the same behaviour. In order to facilitate migration, a mapping of IEC 61131-3 elements into IEC 61499 is proposed to reconcile the differences in the two standards. From this mapping, an IEC 61131-3 application can be re-engineered into a model-driven and component-based design using function blocks. Compared to the original implementation, the resulting function blocks are easier to reuse and the function block standard allows for more execution platforms.

This chapter begins with the description of a baggage handling system (BHS) control-
ler designed using IEC 61131-3 in Section 5.1. Section 5.2 describes the proposed map of IEC 61131-3 elements into equivalent IEC 61499 components. Section 5.3 presents the re-engineering approach, using the example BHS controller to demonstrate key aspects. Results obtained from the IEC 61499 implementation of the BHS are discussed in Section 5.4. Finally, Section 5.5 gives some concluding remarks and identifies avenues for further research.

5.1 IEC 61131-3 specifications

First published in 1993, the goal of IEC 61131-3 [4] was to define a consistent syntax and semantics for 5 different programming languages, which had been already been in use in industry. IEC 61131-3 is most often used in industrial control systems and executed on Programmable Logic Controllers (PLCs). The standard is widely accepted by industry, with most PLCs and industrial development environments supporting at least some of the languages. Unfortunately, many manufacturers still retain incompatible aspects of their original proprietary implementations.

By modern standards, the languages of IEC 61131-3: Ladder Logic Diagram (LLD), Function Block Diagram (FBD), Structured Text (ST) and Instruction List (IL) lack many features. In particular, the absence of object-oriented structure complicates component separation, forcing routines to encompass unrelated functionalities. Similarly, code that is specific to a platform or hardware configuration is also integrated with general control logic. Further, PLCs and their programming languages are designed for centralised monolithic designs. As a result, distribution of IEC 61131-3 applications must be factored into the design from the initial planning phase.

The most common platform for IEC 61131-3 languages are PLCs, which are industrial controllers capable of handling many input and output ports. PLCs execute using what is known as a scan-cycle, visualised in Figure 5.1(a). In a scan-cycle, the controller samples the inputs, executes the routines and writes to outputs. This approach simplifies code development, allowing developers to write code that does not account for input values changing during code execution.

IEC 61131-3 encapsulates the code for a PLC in a configuration, illustrated in Figure 5.1(b), which contains all aspects of the control logic and global variables. The controller is divided into Tasks, similar to processes, which are activated periodically or on the occurrence of certain events. When activated, a Task will execute one or more programs, which are a collection of routines, with a main routine which controls the execution of the program. For example, a task may be triggered on the start-up of the PLC and execute an initialisation program that initialises variables used in the controller programs. IEC 61131-3 supports global variables, declared as part of the controller’s
5.1 IEC 61131-3 specifications

configuration, and scoped variables shared as part of the declaration of a program. These program variables can only be used in routines allocated to the program.

5.1.1 Design of a Baggage Handling System

The controller for a baggage handling system (BHS) will be used to illustrate the development of an IEC 61131-3 application, and illustrate the proposed re-engineering approach. A simple BHS consists of a network of conveyor belts which form redundant routes from a single check-in input location to multiple aircraft loading zones. Each physical conveyor is comprised of a rotary encoder, which measures the distance traveled by the belt, and infra-red bed detectors called photo-eyes, which indicate the positions of bags. The controller uses these inputs to identify the leading and trailing edges of bags, for accurate positioning. The control approach adds multiple entry and exit pseudo-devices, where entry devices are responsible for inserting bags into the model of bags on the conveyor and exit devices hand over exiting or diverting bags to an entry device on the next conveyor.

The BHS controller is implemented in two IEC 61131-3 tasks, one for fast and high frequency monitoring of inputs and another for the slower bag tracking and routing algorithms. They will be referred to as the fast and slow tasks respectively. The fast task maintains the absolute distance traveled by each conveyor belt by monitoring the state of the attached rotary encoders. It also detects the leading and trailing edges of bags using inputs from the photo-eyes. To maintain an accurate count of the distance a belt has traveled, the fast task must read every state change of the rotary encoder. When the conveyor is running at its maximum speed of 1.5 ms\(^{-1}\) the rotary encoder changes state once per 26 ms. As such, the fast task is triggered every 25ms, with the slow task given a slower period of 200ms to perform its more computationally intensive operations. The
PLC is configured to execute the slow task in the unused time between the completion of the fast task, and the next 25ms trigger. Thus the slow task is continually preempted by the fast task, as illustrated in Figure 5.2. Communication between the fast and slow tasks use global variables with additional buffers for variables within the slow-task. Buffering of some variables is required so that when execution of the slow task is interrupted by the fast task the variables it uses are consistent.

![Timing Diagram Illustrating Execution Schedule for two tasks](image)

To illustrate IEC 61131-3 programming, an excerpt of the ladder logic routine for allocating a new bag identification number is shown in Figure 5.3. In ladder logic, routines are divided into rungs and instructions. A a rung is a horizontal segment of the ladder logic drawn as a horizontal line similar to a rung on a ladder. Rungs are used to group instructions together, and are executed one at a time, and from top to bottom. Individual instructions are executed sequentially, but only if previous conditional instructions on the same rung evaluate to true. The screen shot is taken from the RSLogix integrated development [79] from Rockwell [23], with the rung numbers listed on the left. The SBR instruction on rung 0 declares the inputs to the function, and the MOV instruction assigns ABID_ResultCode to a default error code, in case of failure. Rung 1 contains an ADD instruction to increment a stack pointer, but the GRT instruction is used to conditionally reset it to 0 if it exceeds the size of the array. Context information is then added to the ErrorContextStore regardless of the GRT instruction due to the empty branch around the conditional statement. Notice that branches act as a type of pseudo-instruction that alters the flow of execution. In Rung 2, if the function has already set the boolean flag ABID_InitComplete then it will jump to the CODE label. Otherwise execution will continue on the next rung.

While the format of LLD is beneficial for electrical professionals who are more used to electronic circuits using relays and contacts, the graphical format is verbose and wasteful for many operations. The mix of horizontal and vertical execution across the syntax is also more complicated than other languages, such as Structured Text or C. For all IEC 61131-3 languages, however, the nature of a flat application comprised solely of routines makes it difficult to identify and reuse system components. Single functions may be used to implement the behaviour for multiple components, obscuring the interactions between the components. This is more common in time dependent tasks, in which many functionalities can be merged into a single function in order to minimise the time spent in
context switching between routines. The next section presents a proposal for the mapping of IEC 61131-3 design features into IEC 61499 components.

5.2 Proposed Mapping of IEC 61131-3 to IEC 61499

The IEC 61499 [15] open standard attempts to address the inadequacies of IEC 61131-3 by introducing a design framework using event-triggered function blocks as distributable components. The hardware independence of a function block application allows a single specification to be compiled for a number of different platforms. A controller can therefore be tested on other platforms, prior to deployment to a PLC, enabling the use of better debugging facilities. Function blocks are also capable of modelling the behaviour of physical hardware, allowing a controller to be tested more rigorously with a simulated system.

While the IEC 61131-3 and IEC 61499 standards have different execution approaches, (cyclic and event-triggered respectively), similarities in some of the design components allow direct translation. Table 5.1 shows a summary of IEC 61131-3 design elements and the nearest equivalent element in IEC 61499. Tasks, which describe the activation of a program, correspond to one possible reaction of a resource in response to an particular external event or periodic interval. Programs correspond directly to an IEC 61499 re-
source, describing part of the behaviour of an application. Routines in any of the IEC 61131-3 languages are equivalent to IEC 61499 algorithms. Although, some routines may implement a state-machine equivalent to an ECC, or a high-level controller equivalent to a network of function blocks in a composite block.

As IEC 61499 algorithms do not support parameters, inputs and outputs from a routine must be implemented using variables within a function block. Depending on how the parameters are used, input parameters and output values can be implemented as variables in the function block interface or as internal variables. Routines may also use INOUT ports that are both read and written, because of this, they must be implemented as internal variables. If other function blocks need to access the INOUT variable, additional algorithms must be created to synchronise the internal variable with copies in the block interface.

The data types of IEC 61499 function block variables are specified using the IEC 61131-3 set of primitives. As a result, variable declarations may be copied directly. In addition, both IEC 61131-3 and IEC 61499 support user defined data types, although they each use a different syntax.

The two standards diverge with regard to variables which are global or scoped to a particular program. The use of global variables within IEC 61131-3 specifications is common due to the ease of use and benefit to performance and memory usage. IEC 61131-3 supports both of these options, while IEC 61499 does not support either, in favour of the arbitrary distribution of self-contained function blocks. Baggage handling systems, in particular, use a significant amount of global data for storing all of the information gathered for each bag. However, global variables are incompatible with the IEC 61499 concept of easily distributable and reusable components. The following sub-section discusses possible approaches for mapping global variables to IEC 61499 design elements.

### Table 5.1: IEC 61499 implementation of IEC 61131-3 elements

<table>
<thead>
<tr>
<th>IEC 61131-3 Element</th>
<th>IEC 61499 Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task</td>
<td>A single reaction of a Resource</td>
</tr>
<tr>
<td>Program</td>
<td>Part of a Resource</td>
</tr>
<tr>
<td>Routine in 61131-3 languages</td>
<td>Algorithm in any language</td>
</tr>
<tr>
<td>Routine IN ports</td>
<td>Function Block inputs</td>
</tr>
<tr>
<td>Routine OUT ports</td>
<td>Function Block outputs</td>
</tr>
<tr>
<td>Routine INOUT ports</td>
<td>Internal Variables</td>
</tr>
<tr>
<td>User-Defined Data Types</td>
<td>User-Defined Data Types</td>
</tr>
<tr>
<td>Global and Program Variables</td>
<td>No equivalent feature</td>
</tr>
</tbody>
</table>
5.2.1 Global Variables in IEC 61499

Although global variables are contrary to the intention of the function block standard, their use is valuable for many applications. Some possible approaches for handling global or scoped variables within IEC 61499 are discussed below:

1. A single basic function block can be used to store the data, as illustrated in Figure 5.4, with inputs and outputs to write and read data respectively. However, the number of connections for such an approach would be significant, as multiple connections would be needed per variable to enable reading and writing both to and from multiple sources. As each block in the network would have a copy of the variable, it is difficult to ensuring a consistent data value. Communication with the data storage block creates a performance overhead before a block can use the data. The communication protocol would also introduce additional states in any block accessing the global data.

![Figure 5.4: Illustration of Option #1](image)

2. Similar to option #1, a single basic function block can be used to encapsulate global data as local variables within the block. However, instead of standard connections between blocks, the interface could provide pointers to the data for instantaneous read and write access in other blocks. From a design perspective, this is appealing as all components of the system can be visualised, and events and data connections are apparent. An illustration is shown in Figure 5.5. As with option #1, this option would still require a data connection to all blocks for all variables and constants used. Although, as there is no protocol to access the data, there would only be a single connection for a variable per block. Another disadvantage of this approach, is that pointers are not supported by the standard. This is because they invalidate the ability for function block applications to be distributed arbitrarily. However,
data could be synchronised between multiple resources in a number of ways with service interface function blocks. For example, a possible service interface function block could implement a type of **critical section**, where the global variable pointer can only be used within the critical section.

![Figure 5.5: Illustration of Option #2](image)

3. As suggested by Sunder et al. [80], the resource or run-time environment could provide global data storage, and service interface function blocks could be used to read and write to the data. Graphically similar to option #1, this approach would provide the user with a complete visualisation of data connections and usage. Sunder et al. do note that difficulties would arise if a block reads and writes to a variable within an algorithm, as the two values would need to be synchronised after every write. Once again like option #1, inter-block connections introduce performance over-head. Also, memory would be wasted on copies of the variables within each block. Synchronisation of data values would also be complicated if resources are able to pre-empt each other. In particular, when a resource is pre-empted before a block has transferred local changes into the global storage.

4. Dai et al. in [81] suggest that global variables can be removed by employing a class-oriented design approach, instead of the traditional component-oriented approach. Using this approach, a class combines all related functions into a single basic function block so that internal variables can be used. This approach still allows for a class to be used in distributed deployment, by the use of configuration parameters for different instances of the same class block in different resources. As behaviours in a class are restricted to a single function block, only one ECC can be used, with other state-like behaviour implemented as algorithms. The model-view-controller design approach is also not possible with class-oriented control.

5. It is possible to link arbitrary C code together with the C code generated by the
(FBC) [30] synchronous compiler for IEC 61499 at compile time. This can be used to provide global variables and constants to the application. This approach means that at the level of the function block design, there is no indication of the global variables used. Algorithms within basic function blocks would simply have direct access to these variables. It would also allow global constants to be used in the input parameters of blocks. As the function block application is separate from variable declarations, spelling mistakes or connections to undefined constants could only be identified at compile time. In addition, accidental variable misuse would be difficult to debug.

Semantically this approach can be thought of as the use of internal variables for a device, instead of just within a basic function block as per the standard. This offers many benefits including: reduced memory usage compared to Option #1 (because there is only a single copy of each value), and improved performance by mitigating overhead from event-data associations and block connections. This option is also appealing because it makes the function block design very simple and it would be easy to change the compiler.

For this work, linking to a library of global variables was chosen because it is simple to implement and does not require any modifications to the translated LLD routines. However, as with most other proposals, it will be incompatible with the standard and other function block compilers. For industrial automation and resource constrained embedded systems, this trade-off is valuable because of the improved performance.

The current implementation assumes the execution semantics provided by FBC and requires that a resource cannot be preempted. This ensures that access to global variables is atomic. Within a resource, the synchronous semantics used by FBC already ensures atomic operations. A different approach is necessary to handle the case where resources can be preempted, possibilities for this future work are discussed in Section 5.5.

5.3 Proposed Approach

A general tool for reverse engineering of IEC 61131-3 code is hindered by the proprietary implementations of the different manufacturers. Instead only a generalised process is possible, with manufacturer specific adjustments to the intermediate steps. This work develops tools for use with Rockwell's RSLogix [79] integrated development environment (IDE). To demonstrate the approach, the BHS controller introduced in Section 5.1.1 is translated into IEC 61499 function blocks.

Figure 5.6 illustrates the proposed re-engineering process from IEC 61131-3 into IEC
61499. The translation of IEC 61131-3 LLD into C code is automated but vendor specific, whilst the other processes remain manual and generic. The function block interfaces of the lowest level components will reflect the required hardware inputs and outputs, and most blocks will include inputs for configuration parameters. Basic function blocks are manually created to execute the algorithms of a single component. The ECC for such a block may be based on an original LLD routine or, alternatively, an ECC can simply execute an algorithm whenever inputs are updated. Composite blocks can be created, as required, to group dependent components. Once the function block architecture is completed, the C functions obtained by translation of the textual LLD code can be inserted into the basic function block of the relevant component. Execution of the IEC 61499 system is achieved by linking the C code generated by FBC with global data declarations stored in standard C files.

Figure 5.6: Migration plan from IEC 61131-3 to IEC 61499

This approach focuses on automated direct translation of the low-level code instead of identifying and re-creating the architecture of the system from code. This approach was chosen because the structure of IEC 61131-3 applications can vary greatly. As such, the translation of routines can be automated, but the automated generation of objects or components is not possible. The integration of generated C code into function blocks results in a platform independent implementation that can be easily reused. The proposed re-engineering process is therefore comprised of the following steps:

- Function block interfaces are created for each component in the system.
- Function blocks for commonly grouped sub-components are encapsulated into a higher level composite block, until there is one function block per physical component.
- LLD code is exported from RSLogix IDE in a proprietary textual format. Rockwell uses the extension L5K for the text file describing every element of the IEC 61131-3 configuration.
5.3 Proposed Approach

- A manufacturer dependent lexical parser creates an Abstract Syntax Tree (a tree representation of the abstract syntactic structure of textual source code) for the complete IEC 61131-3 application.

- The syntax tree is traversed and global variables, data structures and functions are generated in C code according to the manufacturer’s semantics for IEC 61131-3 LLD.

- The generated C functions are manually grouped according to the relevant physical component and imported as algorithms into the basic function block for the component. Some state-based functions can alternatively be manually re-implemented as an HCECC (introduced in Chapter 4).

- System redesign can be managed by altering connections between these top-level component blocks.

In this work, the resulting IEC 61499 application will be executed by using the FBC synchronous compiler developed by Yoong et al. [82] and improved in Chapter 6. This execution approach was selected because the synchronous semantics are similar to the PLC scan-cycle and execution performance is of crucial importance to many industries including BHSs. FBC has also been shown in a number of tests to use less memory and execute more quickly than the available alternatives (see Chapter 6 or [30]).

While function block algorithms can be written in any of the IEC 61131-3 languages, languages such as C, C++ and Java (used in FBC [82], FORTE [40] and FBDK [39] respectively) are preferred as they are better suited for cross-platform development. It is therefore proposed that all of the routines from the PLC implementation are translated into C code for use with the FBC compiler. As basic function block algorithms do not support arguments, a generated C function is imported by first modifying it to use variables from the block’s interface. In the case of routines that use INOUT variables, such variables are implemented as internal variables in the basic function block. If the variable is also used in other function blocks, the algorithm can be modified to read and write to the interface and use a local variable for internal processing.

5.3.1 Function Block Development

For the BHS example, the fast and slow IEC 61131-3 tasks and their respective programs could be created as two resources. However, separation of the control into two different resources would necessitate two function blocks per conveyor section, making it more difficult to manage. Implementing two resources may also be useful if the target hardware for the function block application is not capable of running both programs within the
fast-task’s 25ms dead-line. As an alternative, the two tasks could be created as a single resource, with two events activating different reactions within the function block network.

For this work, it is assumed that executing on a PC and possibly an industrial controller will be sufficiently fast. As such, all functionalities are composed together within a single composite block to represent a single component. If performance deadlines are not met, the function block application can be divided such that a single controller controls fewer components. Communication between multiple controllers can be implemented similar to the IEC 61131-3 implementation, using service interface function blocks to transmit messages over a network connection.

When compared to IEC 61131-3, the IEC 61499 standard provides the most benefit when a closed-loop model of the plant and controller is developed, allowing complete functional testing of the controller. Following the model-view-controller design pattern [11], at the top most specification level, two resources should be defined to encapsulate the plant and controller subsystems. For this chapter, the plant model explained in Chapter 3 will be used. This allows complete functional testing of the controller prior to the construction of the physical BHS. The visualisation of each component further provides a much greater view of the running system, and provides contextual information which is useful for debugging. The re-engineered controller resource will be comprised of a simple network of blocks representing the components in the BHS, into which the generated C code will be inserted.

The components used in an IEC 61131-3 implementation can be determined by analysing the user-defined data types (UDTs) defined in the configuration. UDTs provide all of the variables required by a component in the original implementation, this can be used to define the interface for the function block re-implementation. Some components may also encapsulate other components, shown either by the UDT or knowledge of the hardware. In such cases, a composite function block should also be created for the component, which contains the embedded components (as well as the basic function block for the component itself).

For this implementation, the original UDT for a conveyor section includes many configuration variables so that all conveyors are instances of the same type. Therefore, a single function block was created for a conveyor, with all of the parameters for any conveyor type, including variable numbers of merge and divert points. In this way, all conveyors in the system can be created in a network by using a single component block with different configuration parameters. An alternative approach is to develop a function block component for each conveyor configuration type, pre-setting variables instead of exposing them in the interface of a fully parametrised block. While this would reduce the complexity of the function block network in the controller resource, it would be more difficult to maintain the different configurations.
The data structure for a conveyor section contains information for a number of entry and exit devices, as well as infra-red detectors known as photo-eyes. Representing separate components, each device type is implemented as a separate function block and embedded within the parametrised conveyor. The full conveyor controller is described as a composite function block called ConveyorControllerParametrised, the network of this block is partially shown in Figure 5.7. The interface is composed of constant configuration settings, ports to communicate with neighbouring components and ports for the hardware inputs and outputs of a standard conveyor. In the function block network shown, the variable number of entry, exit and update devices are encapsulated in container blocks, which only initialise the required number of device controllers. The higher-level conveyor processes, including maintaining the record of all bags in the system and controlling the conveyor belt motor are encapsulated in the ProcessConveyorStopStart and ConveyorController_HCECC blocks.

All function block ECCs are manually developed due to the complex and variable nature used to define state-based control in IEC 61131-3. This manual process is quite simple as it just requires analysis of a single LLD routine and reimplementing the states used, the conditions for transitions, and associating the correct actions to each state. To enable graphical reproduction of the more complex state-based ladder logic routines, HCECCs (presented in Chapter 4) are required, allowing hierarchy and concurrent operations within a basic function block. This is especially useful in cases where a component has more than one routine with a state-machine, or multiple independent behaviours that are executed in each tick. In these cases parallel ECCs can be used to succinctly describe this behaviour.
Figure 5.8 shows the ECC for the PhotoEye Controller block, nested within the UpdateDevContainer composite block. This ECC re-implements the routine in Figure 5.9 in a much simpler and easier to maintain format. The data handling from the routine is extracted from the generated C code and separated into algorithms to be called by the various states. The use of ECCs instead of sequential code simplifies control flow design and maintenance, by replacing many lines of code with an intuitive graphical design.

Figure 5.8: PhotoEyeController ECC

Figure 5.9: Section of a State-machine written in LLD

In the absence of a routine that implements a state-machine for a particular component, active and inactive state-like behaviour can be introduced. For the specification of the main controller block, ConveyorController_HCECC, shown in Figure 5.10, HCECCs
are used to represent the concurrent tasks. Without a strict state-machine routine, the top level HCECC is instead used to distinguish between initialisation, error and running states. When running in the NORMAL state, the concurrent tasks execute one of the required C functions to recreate the behaviour of the LLD implementation. These algorithms check and process the global memory written to by the other components in the conveyor controller. The PublishBagData ECC is also added to allow a visualisation to display the model of bags as seen by the controller. The visualisation can be used to detect errors in the control code, by comparing the controller’s perceived bag positions, with actual positions from the plant model.

![Figure 5.10: Conveyor Controller HCECC](image)

The full controller sub-system is created by instantiating and connecting multiple instances of the ConveyorControllerParamatised block. LLD routines which access the PLC’s inputs and outputs are replaced with service interface function blocks, which use sockets to communicate with the plant model in the plant resource. These communication blocks can be easily replaced later to access hardware inputs and outputs instead.

### 5.3.2 PLC Parsing

The LLD routines, user-defined data types and variable declarations are all exported from RSLogix into a single textual format. The textual format, stored in a file with the extension L5K, conserves the rung and instruction order of the routines and includes meta-data about the controller. A parser for the L5K file was developed using a grammar created using JavaCC [83].

Algorithm 5.1 shows Rockwell’s textual representation of the AllocateBagID ladder logic routine shown in Figure 5.3. An additional instruction (line 13) was added to show a
complete routine in L5K, and newlines were added for readability. The name of the routine is obtained as a token after the keyword ROUTINE, other information such as the inputs and outputs must be determined by examining the instructions. All instructions are of the form Identifier(arguments...), and are stored in the AST (Abstract Syntax Tree) as an Instruction node with all arguments stored as an ordered array of child nodes. The SBR instruction on line 2 declares the start of a routine, where the instruction arguments indicate the inputs to the routine. Similarly, the RET instruction on line 13, indicates the outputs for the routine. Any variables identified in both SBR and RET instructions are actually inout ports, which can be both read and written to.

When the start of a rung is detected with the token 'N:', instructions are stored in an ordered list of child tokens for the rung. When outside the context of instruction arguments, the branch pseudo-operation begins with the token '['], as seen at the start of line 5. Each sub-rung of a branch, separated by a comma in the L5K file, is stored in order within the branch node in the AST. A sub-rung is stored as a standard rung with ordered instructions, which may also contain additional branch instructions. For example, in this case the GEQ and MOV instructions are stored in the first sub-rung, and the second sub-rung is empty. The semantics of the branch instruction is discussed in the next sub-section with regard to equivalent behaviour in C. Code comments, starting with the token 'RC:', are also parsed into the AST. Translation from the generated AST is possible by determining an equivalent statement or function for each instruction.

1  ROUTINE AllocateBagID
2   N: SBR(ABID_MinBagID,ABID_MaxBagID,ABID_LastBagID)
3   MOV(RESULTCODE_UNKNOWNERROR,ABID_ResultCode);
4   N: ADD(ErrorContextStore,StackPtr, 1, ErrorContextStore.StackPtr)
5   [GEQ(ErrorContextStore.StackPtr,ErrorContextStore.SIZE)
6    MOV ( 0, ErrorContextStore.StackPtr ) , ]
7   MOV(THIS_PLL_SUBSYSTEM_ID,
8    ErrorContextStore.Entry[ErrorContextStore.StackPtr].SubSystemID);
9   MOV(UNIT_ALLOCATEBAGID,
10    ErrorContextStore.Entry[ErrorContextStore.StackPtr].SoftwareUnit);
11   N: XIC(ABID_InitComplete)JMP(CODE);
12   RC: Additional Rung
13   N: RET(ABID_ResultCode);
14  END_ROUTINE

Algorithm 5.1: Textual format of Ladder Logic Rung
5.3 Proposed Approach

5.3.3 Code generation

An application developed specifically for Rockwell’s LLD semantics generates equivalent C code from the syntax tree created by the parser. This program, called L5K2C, includes a mapping of all LLD instructions into equivalent C statements or functions as well as a map relating standard IEC 61131-3 data types into C primitives. The behaviour of each instruction was obtained from documentation in [84]. More complicated instructions, such as the Branch pseudo-instruction are handled in special functions in order to generate both semantically equivalent and human readable code.

L5K2C walks through the tree and stores different collections of information into separate files. For example, user-defined data types in the L5K file are all stored in a single sequence, these are saved in a single C header file as structs, using the C equivalent of the primitive IEC 61131-3 data types. Within IEC 61131-3, data types, variables and routines can have the same name, causing conflicts in directly translated statements. When generating C code, L5K2C modifies any declaration to ensure global uniqueness. In the case of user-defined data types, the names of the generated structs are modified to include "Struct" as a suffix.

When L5K2C traverses global and program variable nodes in the AST, variables are generated in a file specific to the relevant scope in order to allow controlled inclusion of variables. Unlike user-defined data types and routines, the names of global variables are not changed so that L5K2C does not need to change all occurrences of global variables in translated routines. Variables from a program scope are modified however because they may overlap with variables in other programs. Each generated declaration uses the equivalent C primitive type, or the translated user-defined data type with the suffix "Struct" appended.

Each routine is generated in a separate C file, which begins with some #include directives to include the global variable declarations, as well as variables scoped to the program the routine is part of. As with user-defined data types, the names of routines can also have the same name as routines in other programs. Following the same approach as with variables, all routines are given a prefix of the program name. The modified name of the routine is also used as the name of the C file. As stated earlier, the names of global variables are unchanged to minimise changes to the routines. Local variables are also left unchanged as they cannot overlap with any other set of variables. Only instruction arguments which reference variables from program scopes are generated with the prefix of the program name, as they may overlap with other programs.

Algorithm 5.2 shows the procedure for the generation of C code from a routine. The whole routine is searched for the SBR and RET instructions to determine the inputs and outputs of the function (lines 4-5). INOUT ports are then determined and removed from the sets of inputs and outputs (lines 6-9). The data type for each of the arguments to
procedure L5K2C_Routine(routine)
  
  programme name := program name that routine belongs to;
  name := name of the routine;
  I := the set of inputs obtained from parameters of SBR instruction;
  O := the set of inputs obtained from parameters of RET instruction;
  
  foreach io ∈ I which are also ∈ O do
    append io to new set IO, the set of input-output variables;
    remove io from both I and O;
  endforeach
  
  Get data types for I, O, IO variables from known scopes;
  generate declaration for routine using the format:
  void programme_name(type i ∈ I, type *io ∈ IO, type *o ∈ O);
  
  L := ordered set of rungs and comments in routine;
  
  foreach l ∈ L do
    if l is rung then
      foreach child ∈ l do
        if child is instruction then
          generate C statement for instruction using look-up table;
        end
        else if child is branch then
          generate each sub-rung as a normal rung;
          generate if statement to test if the simplified condition of all
          conditions in any one rung is met;
          print remaining child ∈ l within scope of if statement;
        end
      end
    end
    else if l is a comment then
      generate multi-line C comment using the contents of l;
    end
  end
end procedure

Algorithm 5.2: Algorithm for the translation of LLD to C

the routine are discovered by checking the lists of global and program variables (line 10).
Only then can the definition for the function be generated, with a void return type as all
of the inputs and outputs of the function are stored as arguments (lines 11-12).

The ordered set of rungs and runtimes is iterated over to generate the semantically
equivalent C statements (lines 13-30). Every child of a rung node in the AST represents
an instruction or branch node, and the children of either node represent ordered
arguments. When generating code for an LLD instruction, a look-up table is used to
generate the equivalent C statement (line 18). A custom template syntax is used to
programmatically generate the C code for each instructions. In the case of the ADD in-
struction, the C template is $3 = 1 + 2;$, where the numbered tokens are replaced with
the text of appropriate chilk node. For example, \( ADD(Var1, Var2, Var1) \) is generated as \( Var1 = Var1 + Var2 \). To match the renamed program variables, any instruction arguments that match the name of a program variable are prefixed with the program name. Some instructions need to be handled more dynamically than templates allow. Instructions such as \( JSR(Var1, Var2, ...) \) accept a variable number of arguments, so it must be handled specially. In the case of \( JSR \), a function call is generated in C, which calls the function \( ProgramName_{Var1} \) with any remaining arguments passed as arguments to the C function.

The branch pseudo-instruction complicates compilation due the semantics of the control flow it represents (lines 20-24). Using Figure 5.11 as an example, the semantics of the relatively simple looking LLD require that the MOV instruction must be executed if \( B \) is true, regardless of \( A, C \) and \( D \). Finally, the output \( O \) must only be emitted if \( A \) or \( B \) or \( C \) or \( D \) is true.

![Figure 5.11: Branch in LLD](image)

In C code, this type of behaviour is implemented by in Alg. 5.3. Each sub-rung is generated in the same way as a standard rung, though conditional statements are also tracked for later reuse. Instructions after the branch are encapsulated in an if statement which represents whether control flow in the LLD implementation activated the right hand side of the branch by satisfying the conditions on at least one sub-rung.
if( A ){
    ...
}
if( B ){
    Y = X;
}
if( C ){
    ...
}
if( D ){
    ...
}
if( A || B || C || D ){
    O = true;
}

Algorithm 5.3: Equivalent Behaviour in C

ROUTINE AllocateBagID
N: SBR(ABID_MinBagID, ABID_MaxBagID, ABID_LastBagID)
MOV(RSULTCODE_UNKNOWNERROR, ABID_ResultCode);
N: ADD(ErrorContextStore.StackPtr, 1, ErrorContextStore.StackPtr)
  [GEQ(ErrorContextStore.StackPtr, ErrorContextStore.SIZE)
   MOV(0, ErrorContextStore.StackPtr),]
  MOV(TTHIS PLC SUBSYSTEM_ID,
      ErrorContextStore.Context.Entry[ErrorContextStore.StackPtr].SubSystemID);
  MOV(UNIT_ALLOCATEBAGID,
      ErrorContextStore.Context.Entry[ErrorContextStore.StackPtr].SoftwareUnit);
N: RET(ABID_ResultCode);
END_ROUTINE

Algorithm 5.4: Textual format of Ladder Logic Rung

A sub-section of the routine shown in Algorithm 5.1 is reproduced in Algorithm 5.4 to demonstrate the procedure when generating the C code shown in Algorithm 5.5. The algorithm name is AllocateBagID, within the program SlowProgram. From the SBR instruction (seen only in Algorithm 5.1) the set of inputs, I, contains ABID_MinBagID, ABID_MaxBagID, ABID_LastBagID. Similarly, from the RET instruction (also in Alg. 5.1) the set of outputs, O, contains ABID_ResultCode. Because there are no overlaps in variable names in the sets I and O, there are no elements in IO. The data types for all variables used within routines are determined first by searching the local scope, then the program and global scopes. The C function declaration is then generated as shown on line 1. The first rung in the routine, contains two instructions, SBR followed
5.3 Proposed Approach

```c
void SlowProgram_AllocateBagID(int ABID_MinBagID, int ABID_MaxBagID, int ABID_LastBagID, int *ABID_ResultCode) {
    ABID_ResultCode = RESULTCODE_UNKNOWNERROR;
    ErrorContextStore.StackPtr = ErrorContextStore.StackPtr + 1;
    if( ErrorContextStore.StackPtr >= ErrorContextStore.SIZE ) {
        ErrorContextStore.StackPtr = 0;
    }
    return;
}
```

Algorithm 5.5: C code generated for AllocateBagID

by MOV. The C equivalent of SBR is an empty string, so nothing is generated. The MOV instruction is generated as shown on line 2.

The second rung of the routine is slightly more complicated, due to the branch. The ADD instruction before the branch is generated using the standard template, producing line 3. The branch instruction starts and the rungs within it are printed in order, starting with GEQ in the first sub-rung. GEQ is a conditional instruction which tests if the first argument is greater than or equal to the second argument. All conditional statements are generated within an if statement, as shown on line 4. In the case of multiple conditional statements in a sequence, they are all added to a single if statement with a logical AND between them. The MOV instruction, which occurred after the GEQ instruction, is generated within the if condition (line 5). The next sub-rung in the branch is empty, so nothing is generated. At the end of the branch, an if statement is generated to conditionally execute the two MOV instructions after the branch. From the first sub-rung the only condition is the GEQ instruction. The second sub-rung is always true as it contains no conditions. Because the condition checking either sub-rung one or sub-rung two, it will always evaluate to true and so no condition is generated. The two MOV instructions are generated as normal (lines 7 and 8). Finally, the RET instruction and its arguments, on rung 3, is generated as return; (line 9). This is because output values are passed as pointers to the function, so no translated routine directly returns any data.

Generating C code from a PLC specification results in many C functions and header files which declare global or scoped variables and data types. In order to create a functional controller, separate functions could be developed to handle inputs and outputs and to execute the main function for each program using the same triggers used by the tasks in the original implementation. However, in order to harness the benefits of model-driven engineering, it is beneficial to spend time to manually migrate this code into IEC 61499 function blocks.
The generated C functions are moved into algorithms within the basic function block for the relevant component. The function block compiler used in this work (FBC), also accepts #include directives which can be added into the CompilerInfo tag in the function block's XML. These allow the appropriate global and program variables to be included in the C code generated for the function block.

The accuracy of this LLD to C translation was initially confirmed by manual analysis with experienced IEC 61131-3 developers. After completion of a FB to LLD generator presented in Chapter 6, it was also possible to test that LLD code translated into C code and back into LLD was semantically identical.

### 5.3.4 Summary

The re-engineering of IEC 61131-3 code into IEC 61499 creates a library of function blocks for a set of components. These function blocks are easily reused, as the creation of a new BHS design simply involves the connection of function blocks in a new network.

The proposed process is capable of migrating IEC 61131-3 configurations with any number of tasks. As IEC 61499 does not provide a mechanism for executing different resources at different rates, each function block is executed once per tick, ignoring the execution frequency of the original task. Provided the IEC 61499 implementation is executed on a faster platform than the original PLC, this is not an issue. However, Chapter 6 presents a FB to LLD code generator that allows different aspects of a FB system to be allocated to tasks executing at different rates.

The use of global variables is not ideal, as it complicates the distribution of the function blocks. However, this allows the complete reuse of the automatically generated C code, with only a few LLD routines replaced by the architecture of function block HCECCs and networks. From this initial implementation, global variables may be phased out by implementing any of the other options considered in subsection 5.2.1. However, the performance of the IEC 61499 implementation would be significantly handicapped by the use of standard function block inter-connections. The following section evaluates the code size and provides some performance results.

### 5.4 Results

After completely re-engineering the controller, the function block system is compiled using FBC [30]. Compilation produces 5.74 MB of C code, including the plant and controller. In comparison, the original L5K file was 7.28 MB, which only includes the control code in
a format that is not easily human readable. The C code benefits from the use of function block networks for high level specification, instead of long and hard to follow LLD routines to implement the same behaviour.

Comparing the lines of C code versus the number of LLD rungs for a given routine is a skewed comparison, due to LLD allowing multiple statements per rung. In addition, the generated C code is formatted in an easily readable way, adding lots of white space. For individual algorithms however, C code makes it easier to understand the exact execution flow, while LLD loses clarity because of its graphical nature.

The PLC implementation of the original LLD code must execute the fast task every 26ms, while the slow task is executed in any spare time the PLC has. For PC execution, the benchmark was tested on a 2.5 GHz dual core, 4GB RAM laptop. The system was able to execute 1 million cycles in 4 minutes and 47 seconds. That is an average of 287 µs per cycle of both plant and controller in a single thread and using less than 8 MB of RAM. It should be noted that because of the vast difference in PLC execution power, this benchmark of the function block system on a PC is not necessarily demonstrative of the implementation performance of the same function block network compiled back to a PLC. To allow a more useful comparison, the function block system would need to be compiled and executed on the same PLC. However, this is not possible due to the lack of compiler support. Regardless, the benchmark does show that an industrial PC, such as the WAGO 750-860 (a Linux Programmable Fieldbus Controller [85]) with a 40MHz ARM7 processor, should be able to execute the control resource within the performance dead-lines.

5.4.1 Visualisation

The visualisation tool, developed in Python [86], called MHVIS can create a 2-D OpenGL (OpenGL (2008)) representation of the BHS, together with a graphical interface for user interaction. MHVIS generates a visualisation, as shown in Figure 5.12(a), by parsing all of the plant resources in the IEC 61499 system file to create the layout of conveyors. This approach enables MHVIS to visualise arbitrary baggage handling system layouts. Once the view has been created the user can start the plant simulation using the graphical interface, shown in Figure 5.12(b), which sends data to the SUBSCRIBE blocks in the function block application.

During execution MHVIS animates each component based on data from both the plant and the controller. By subscribing to the PUBLISH blocks in the plant and controller subsystems, the visualisation can show the position of the bags on the conveyor belts, the conveyor movement, and the status of the diverter all in simulated real-time. The visualisation shows the bag positions determined by the controller (blue outline) and the actual positions calculated by the plant model (red rectangles). A developer is therefore
Figure 5.12: Screenshots from MHVIS

(a) Screenshot of MHVIS, BHS visualisation program

(b) Screenshot of the Human-machine Interface

able to clearly see any inconsistencies in the controller’s estimated model and change the control code to reduce the rate of lost bags or errors.
5.5 Discussion

The re-engineering approach presented in this chapter enables complete migration of PLC code into IEC 61499. This method allows the reuse of all platform independent code, including global variables, in a modern industrial framework. The manual creation of the function block architecture is a minor once off cost, which is fairly simple. The proprietary syntax of the original code is automatically translated into standard C code and inserted into basic function block algorithms. The automated translation of code provides confidence in the resulting specification, while manual component design ensures human-readable design with a coherent implementation of one function block per system component. The resulting function block components are much easier to maintain, reuse and distribute. Different systems can be easily created from these components, involving only a change in the top level network, instead of changes to configuration data and routines spread throughout the PLC controller.

The development of a visualisation as part of model-view-controller was hugely advantageous to debugging. Prior to usage of a visualisation, command line print statements and standard debuggers were very limited for debugging high-level functionality. Visualisation of the entire system allows representation of the complete system state, and makes determining the cause of errors much easier.

The chosen synchronous execution of function blocks remains similar to PLC execution, which results in an easier translation process. By using C for execution, arbitrary declarations can be added into the executable, enabling global variable access. Further, removal of global variables from the specification would require additional development to re-design the function block architecture, as well as the generated algorithms.

While the BHS developed in this work is targeted to a centralised controller, the distribution of function blocks which use global variables can be handled in the same way as in the IEC 61131-3 implementation. Within a device, function blocks can communicate directly with each other using global variables, and between separate devices, a message passing protocol can be implemented to transfer the required data. This approach improves performance and memory usage within a single device, and distributed data handling can utilise aspects of any message passing LLD routines which are also translated into C.

Future work may focus on removing or reducing the manual function block and ECC creation steps in the presented process. Many aspects, particularly the interface of blocks, could be automatically created by analysing user-defined data types. Although, the user would still need to check and modify the generated interfaces depending on the exact IEC 61131-3 implementation. The ECCs within components will likely remain manually created, as LLD allows for a variety of methods of state machine specification which are
not easily automatically translated. Given a standard specification approach for state-
machines in LLD, the translation tool could be extended to create ECCs from appropriate
code.

While this work is concerned with modernising control software, it is still important
to manage classical PLC hardware. PLCs will likely be integrated alongside newer con-
trol hardware as new standards solidify their position, and older technology is replaced.
Implementation of PLC applications in IEC 61499 allows for configuration of both new
and legacy controllers, thus translation of IEC 61499 function blocks into IEC 61131-3
languages for execution would be desirable.

As mentioned previously, the use of global variables and constants is important for
simplifying designs and providing communication between blocks with no overhead. The
current approach is usable for most implementations, except for when truly parallel re-
sources are used, where the execution of a resource can be preempted by another. A
possible more general implementation may include communication function blocks which
send and receive a full copy of the data between resources. If the data is not received,
it is up to the designer of each block to not access the data, or at least be aware not to
write to the stale local copy. In a tick where the local copy of the global data is updated,
then local function blocks are notified via an event that these variables can be used.
6

Compilation of IEC 61499 Function Blocks

The hardware abstraction of the IEC 61499 standard allows for a single function block specification to be implemented on a range of controllers. In addition, function block applications are saved as an easily readable XML file, enabling inter-operability between tools. To execute function block applications, code generators translate this XML specification into an executable format. Service interface function blocks are a special exception to this translation, as the implementation of such a block is hardware or platform dependent, and not included within the definition of the function block itself. This target specific implementation must be managed by the code generator, by retaining a library of known service interface function blocks for a specific platform.

At present, function block applications are most commonly executed using a run-time. The run-times, such as FBRT [39] and FORTE [40], schedule block execution and manage event propagation. The code generator for the associated run-time generates a class object for each function block in the application. Arbitrary function block applications are then executed by instantiating objects of the required class within the run-time.

Using the synchronous semantics presented in Chapter 4, blocks are statically scheduled and events are periodically sampled. This allows for the generation of directly executable and deterministic code. A synchronous compiler for function blocks is therefore able to produce compact code with less execution over-head than run-time alternatives.
The reduced code size and performance requirements also allow for execution on more resource constrained hardware platforms.

This chapter presents two code generators which produce deterministic and deadlock free executable code from IEC 61499 specifications, including applications containing HCECCs, for two types of platforms. The first tool is an extension to FBC, from Yoong et al. [30, 34], to support direct code generation from HCECC specifications. As part of this development, FBC was also extended to support algorithms written in IEC 61131-3 Structured Text and IEC 61499 Adapters. The second tool uniquely generates an IEC 61131-3 configuration from an IEC 61499 system, for execution on Rockwell PLCs. The C code generated by FBC allows targeting of current and future controllers, while PLC code generation allows IEC 61499 to replace IEC 61131-3 in current industrial controllers.

Generation of executable code will be demonstrated using a baggage handling system with 16 conveyors following the model-view-controller design approach. Figure 6.1 shows the layout of the BHS, including the two merge and two divert points, using a screen-shot from the developed visualisation program.

Figure 6.1: Visualisation of example BHS
The IEC 61499 system description of the BHS is shown in Figure 6.2. It is allocated to a single device with two resources, one for the plant, and a separate resource for the controller. The plant and controller resources use a composite function block for each conveyor section, simulating and controlling the section respectively. The plant model is taken from the case study in Chapter 3, while the controller is based on the re-engineered BHS controller in Chapter 5. The two resources communicate with each other, and with the external visualisation program, using UDP sockets.

![Figure 6.2: System Architecture for BHS example](image)

Section 6.1 presents the generation of C code from IEC 61499 specifications, extending the compiler originally presented in [33]. Section 6.2 then introduces a compiler for the generation of ladder logic diagrams, for execution on Rockwell [23] PLCs. Section 6.3 concludes this chapter with some discussion of the developed tools.

## 6.1 Compilation to C

This section presents developments to enable the direct compilation of basic function blocks with HCECCs into C code. Compilation of most of the other aspects of the IEC 61499 standard is achieved by using FBC [30] as the basis for this compiler. However, support is also introduced for algorithms using Structured Text (ST) and IEC 61499 Adapters which were previously not handled.

Using the original synchronous semantics for IEC 61499 developed by Yoong et al., FBC generates C code for arbitrary IEC 61499 models which can then be compiled into a stand-alone application. As a result of the semantics and code generated, function block applications compiled by FBC have the fastest reaction time and lowest memory usage among various existing IEC 61499 compilers [30]. The use of C code allows FBC to generate code which can be compiled and executed with or without an operating system. Further, the direct execution of the function block application allows resource-constrained platforms to be targeted that would not be possible using a run-time approach.

This section describes the generation of C code using the plant resource from the example BHS system. The approach used by FBC is introduced first, followed by ex-
tensions for HCECCs, algorithms written in Structured Text, and IEC 61499 Adapters. The modified compiler is also evaluated against FBRT [39], the commercial run-time nxtFORTE [45] and the original FBC [30] using a series of benchmarks in Section 6.1.3.

6.1.1 Code generated by the Original FBC

The original FBC compiler, presented most recently in [30], synthesises C code following a depth-first traversal from the XML file of the IEC 61499 component passed to it. All XML elements are parsed into an internal format, and code is generated for the nested elements first. In the case of the BHS_1 baggage handling system shown earlier, the path of the XML file for the system illustrated in Figure 6.2 is passed to the compiler as a command-line argument. From this system file, code for the plant and controller resources is generated first, followed by code for the device.

Approach

FBC creates a unique type, called an FBType, associated to each function block, resource or device used. For every unique FBType, a unique set of files is generated containing a C structure and set of functions.

The C data structure for a block is used to store all of the data related to an instance of the block. It stores the interface for the block including input and output event and data ports with extra variables added depending on the type of block. The FBType structure is given two copies of any input or output data port in order to implement the semantics of event-data associations (where data values between the interface and block internals are buffered).

At least two functions are generated for each FBType. An initialisation function, FBTypeinit, is generated for each block type, to serve as a constructor for the block. The reactive function, FBType run, implements the execution semantics of the block for a single tick. In each tick, the reactive function is also responsible for the semantics of event-data associations, synchronising the two copies of each data value, on the presence of an associated event. These updates are performed at the start of the tick for input ports, and the end of the tick for output ports. All of the functions for a block have a single argument that is a pointer to the FBType structure for the instance, and it is given the variable name me. Algorithms within basic function blocks are also compiled into functions, using the same instance data pointer to access interface and internal variables.

Main Function

FBC generates a main function for the top-level IEC 61499 component passed as a parameter to FBC. Although, if FBC is instructed to compile a function block system file, a
main function is generated for each device in the system. The purpose of the main function is to declare the top-level instance data, and call the appropriate reactive function in an infinite loop. All of the C source and header files generated by FBC can then be easily compiled into an executable for deployment.

Translating Devices and Resources

Each device in a system is compiled into a single executable, intended to be deployed onto separate hardware. A device is solely responsible for the execution of multiple resources, which are sequentially executed once per tick. The single device in the BHS example is compiled into a reactive function that executes the plant and controller resources. As resources do not have an interface like other blocks, execution of resources is simply the execution of the embedded network of blocks.

Translating Block Networks

FBC parses block networks within composite blocks and resources into a netlist before generating the reactive function. Declarations for the instance data of embedded function blocks are also added to the generated FBType data structure.

From the semantics used by FBC, function blocks within a function block network are executed in synchronous parallel with each other. To generate executable code, the ordering of block execution must be sequentialised, but this incurs some well-known compositional issues requiring causal analysis [28]. In the synchronous approach, all computations are conceptually instantaneous within a tick. Due to this, arbitrary connections of blocks can result in deadlocks, for example where two blocks both wait for the completion of each other. Intuitively, outputs are said to be produced from inputs, but feedback loops create non-causal cycles [67] as the separation of cause and effect is blurred. FBC takes the approach presented by Maraninchi et al. [67], which defines a notion of correct compositions, such that not all block execution orders are valid. FBC provides two implementations for generating executable code, selectable using arguments given to the compiler.

Delayed Communication - All communication between blocks is forced to be delayed until the next tick, simplifying code generation.

Instantaneous Communication - After analysing the structure of a network, blocks are executed in a sequential order that only executes a block after the blocks connected to its inputs have all been executed.

Acyclic Communication - If completely instantaneous communication is not possible due to non-causal loops, communication chains can be broken by arbitrarily forcing
Delayed Communication for one of the connections.

FBC generates a reactive function for a network using delayed communication by assigning all input ports to the values of outputs before outputs are written to in the current tick. In this way, the network composition is guaranteed to be causal, because inputs in the present tick use the value of the outputs in the previous tick.

In contrast, instantaneous communication offers improved performance by allowing blocks to read the outputs provided by connected blocks in the current tick. In this approach, if cyclic execution dependencies are identified in the block network, FBC produces an error as valid code cannot be generated. Acyclic Communication is similar, but ensures any function block network can be compiled by arbitrarily breaking non-causal loops.

Non-causal cycles are identified using a modified topological sort shown in in Algorithm 6.1. Every function block is iterated over by the EnsureAcyclic function, using an initially empty stack, S, to track connection paths. If a function block, fb, is already found in S, then this indicates a connection loop back to the fb block (line 2). If Acyclic Communication is selected, any cyclic dependencies identified are broken by forcing delayed communication for a single connection in the loop (line 4). For instantaneous communication, networks with cyclic dependencies are rejected and code generation fails with an error (line 7).

The main sorting algorithm occurs on lines 10-20. All of the predecessors of a block are recursively iterated over in order to produce a flat sorted list L. Where predecessors are the source blocks of any input connection to the block. If the function recursion on line 14 returns true, it means that a unit delay has been inserted to break a cycle. This indicates that the topological order for the fb block has been determined and iteration over predecessors can stop (line 15). If line 14 returns false, the sorting algorithm will continue iteration over remaining predecessors of fb. Finally, fb is appended to the end of the sorted list L (line 19). Where FBC inserts a unit delay, this corresponds to placing a block earlier in the list L than one of its predecessor blocks. In this way, the delayed block will read outputs from the predecessor block which were calculated in the previous tick.

The code generation procedure for a composite block is shown in Algorithm 6.2. Lines 4-7 declare interface and internal copies of variables for input and output data ports. Code is generated to implement event-data associations on lines 10-12 and lines 34-36. Previous input and output events do not need to be cleared by a composite block, as the execution of embedded blocks will provide this behaviour.

The EnsureAcyclic function call on line 8 creates the ordered list L. In the case where EnsureAcyclic fails to create an ordered list of all blocks in the network, FBC will report an error and exit. The ordered list of blocks, is processed on lines 13-24. The input ports for each block are updated (lines 14-22), and the block is executed (line
6.1 Compilation to C

```c
function EnsureAcyclic(fb, S)
    if fb is in S then
        if configural to break cycle then
            arbitrarily insert unit delay here;
            return true;
        else
            exit with error: cycle detected;
        end
    end
    if fb has not been visited then
        add fb to visited set;
        push fb onto S;
        foreach predecessor, n, of fb do
            if EnsureAcyclic(n, S) = true then
                break;
            end
        end
        pop fb from S;
        insert fb into sorted list, L;
    end
    return false;
end function
```

Algorithm 6.1: Function to topologically sort a function block network and ensure there are no connection loops [30].

23). The function connectionSet() on lines 16 and 27 is used to return the set of connections to a port. Multiple connections can be made to and from event ports and from data ports. However, only a single value may be connected to a data port. This distinction requires separate handling of event and data connections (lines 17-21). After the embedded network has been executed, the output ports of the composite block are updated (lines 25-33).
procedure GenerateCFB(fb)
  IE := set of all input events in fb;
  OE := set of all output events in fb;
  ID := set of all interface input data in fb;
  ID' := set of all internal input data in fb;
  OD := set of all interface output data in fb;
  OD' := set of all internal output data in fb;
  EnsureAcyclic(an arbitrarily first block from the network, empty stack);
  L; // ordered list of function blocks (created by EnsureAcyclic)
  foreach ie ∈ IE associated with id ∈ ID do
    generate code to update id' with id whenever ie occurs;
  end
  foreach b ∈ L do
    I := set of all input ports of b;
    foreach i ∈ I do
      IC := i.connectionSet();
      if i.type = EVENT then
        generate code to assign i to the logical OR of all events in IC;
      else if i.type = DATA then
        generate code to assign i to data, ic ∈ IC;
      end
    end
    make call to execute b;
  end
  O := set of all output ports of fb;
  foreach o ∈ O do
    OC := o.connectionSet();
    if o.type = EVENT then
      generate code to assign o to the logical OR of all events in OC;
    else if o.type = DATA then
      generate code to assign o to data, oc ∈ OC;
    end
  end
  foreach oe ∈ OE associated with od ∈ OD do
    generate code to update od with od' whenever oe occurs;
  end
end procedure

Algorithm 6.2: Algorithm to generate C code for a function block network.

To demonstrate this procedure, the network from the ConveyorModel composite
function block, shown in Figure 6.3, will be used. The ordered list of blocks, L, will consist
of the Conveyor_Belt_Model block followed by the Conveyor_Photoeyes_Model
block, because of the flow of events and data described by the connections. The procedure
with therefore generate code as follows:

1. Connections from the interface of ConveyorModel to Conveyor_Belt_Model
are read.

2. `Conveyor_Belt_Model` is executed, computing its new outputs.

3. Connections to `Conveyor_Photoeyes_Model` are read, providing the block with the recently calculated output values from the `Conveyor_Belt_Model` block.

4. `Conveyor_Photoeyes_Model` is executed.

5. Finally, all output ports of the composite block interface are updated from the outputs of the internal blocks.

![Figure 6.3: Network of Block Modelling Conveyor Section](image)

FBC uses a similar procedure for generating code for the function block network within a resource. However, as resources do not have a block interface, port declarations (lines 2-7), output connections (lines 25-33), and event-data associations (lines 10-12 and 34-36) are omitted. Instead, only block executions and inter-block connections are generated.

**Translating Basic Function Blocks**

The translation of a basic function block generates a reactive function to execute the behaviour of the ECC. Additional C functions are created for each of the embedded algorithms, to be called by the reactive function. The data structure for basic function blocks also includes a variable to store the index of the current state (_state). An additional variable, _entered, is added to indicate whether or not the current state has been entered, because ECCs are Moore-machines.
The reactive function for a basic function block is generated by the procedure shown in Algorithm 6.3. Lines 5-8 declare the variables for input and output data ports, including an internal copy for buffering each port. At the start of each tick, the previous output events are cleared (line 9). Code is then generated to check input events and update the local copies of associated data ports (lines 10-12). An infinite for loop is generated (lines 13 and 37) around a switch statement generated by lines 14-36. This is used to allow a new state to be entered after taking an exiting transition. For each state, a unique case statement is generated within the switch statement (line 15). A condition within each case statement detects if the state has already been entered (line 16). The actions of the state are executed (lines 17-24) on entry, or exiting transitions are tested and taken (lines 26-34) if it has already been entered. If an exiting transition is taken, the state variable is updated and the infinite for loop is restarted (line 29). After a state has finished its actions, or if no exiting transition is taken, control flow exits the infinite for loop (line 35). After the ECC has completed execution for a given tick, the input events to the block are cleared (line 38). Finally, code is generated to update the interface copy of output variables of variables associated to emitted output events (lines 39-41).

An excerpt of the reactive function generated for the ECC in Figure 6.4 is shown in Algorithm 6.4. Upon entry into the TICK state (lines 4,5), the algorithms TICK and CheckDiverters are executed, and the output event CNF is emitted (lines 6-8). Line 24 breaks out of the switch statement and line 27 exits the infinite for loop. Subsequent executions of this reactive function will test the conditions of exiting transitions (line 11-23) until a new state is entered. Because TICK has an always true exiting transition, if no other transition is taken the ECC returns to the IDLE state. This can be seen in lines 18-22, which changes the state index and clears the entered flag. The continue statement on line 21 utilises the infinite for loop to restart the evaluation of the switch statement at line 2.

Code for algorithms is generated by encapsulating the user defined algorithm text in a C function definition which includes a pointer to the instance data of the block. As a result, the algorithm content entered by the user must be written in C. In order for the code to compile, the user must also use the instance data pointer, me, to access block variables.
6.1 Compilation to C

1 \textbf{procedure} \text{GenerateBFB}(fb) \\
2 \hspace{1em} S := \text{set of all states in } fb; \\
3 \hspace{1em} IE := \text{set of all input events in } fb; \\
4 \hspace{1em} OE := \text{set of all output events in } fb; \\
5 \hspace{1em} ID := \text{set of all interface input data in } fb; \\
6 \hspace{1em} ID' := \text{set of all internal input data in } fb; \\
7 \hspace{1em} OD := \text{set of all interface output data in } fb; \\
8 \hspace{1em} OD' := \text{set of all internal output data in } fb; \\
9 \hspace{1em} \text{generate code to clear all output events in } OE; \\
10 \hspace{1em} \textbf{foreach} \ \text{ie} \in IE \ \text{associated with id} \in ID \ \textbf{do} \\
11 \hspace{2em} \text{generate code to update id'} \text{ with id whenever ie occurs;} \\
12 \hspace{1em} \textbf{end} \\
13 \hspace{1em} \text{generate code to start an infinite } for \text{ loop;} \\
14 \hspace{1em} \textbf{foreach} \ s \in S \ \textbf{do} \\
15 \hspace{2em} \text{generate new case for } s \ \text{in switch-statement;} \\
16 \hspace{2em} \text{generate condition to execute actions if not already entered;} \\
17 \hspace{2em} \textbf{foreach} \ \text{action, a, of s do} \\
18 \hspace{3em} \text{if a has algorithm, alg then} \\
19 \hspace{4em} \text{generate call to function[alg];} \\
20 \hspace{3em} \textbf{end} \\
21 \hspace{3em} \textbf{if a has oe} \in OE \ \textbf{then} \\
22 \hspace{4em} \text{generate code to set oe;} \\
23 \hspace{3em} \textbf{end} \\
24 \hspace{1em} \textbf{end} \\
25 \hspace{1em} \text{generate condition to evaluate and take exiting transitions;} \\
26 \hspace{1em} \textbf{foreach} \ \text{transition condition, t, of s do} \\
27 \hspace{2em} \textbf{if t leads to next state } n \in S \ \textbf{then} \\
28 \hspace{3em} \text{generate code to test for } t \ \text{and if true:} \\
29 \hspace{4em} \text{update } _-\text{state} \text{ and enter the next state } n \text{ by restarting the } for \text{ loop;} \\
30 \hspace{2em} \textbf{end} \\
31 \hspace{2em} \textbf{if t is always true} \ \textbf{then} \\
32 \hspace{3em} \text{break;} \\
33 \hspace{2em} \textbf{end} \\
34 \hspace{1em} \textbf{end} \\
35 \hspace{1em} \text{generate code to exit the infinite } for \text{ loop;} \\
36 \hspace{1em} \textbf{end} \\
37 \text{generate code to end the infinite } for \text{ loop;} \\
38 \textbf{foreach} \ \text{oe} \in OE \ \text{associated with od} \in OD \ \textbf{do} \\
39 \hspace{2em} \text{generate code to update od with od'} \text{ whenever oe occurs;} \\
40 \hspace{1em} \textbf{end} \\
41 \textbf{end procedure}

Algorithm 6.3: Algorithm to generate C code for a basic function block.
for (; ; ) {
    switch (me->_state) {
    ... 
    case 2: // TICK State
        if (!me->_entered) {
            ConveyorModel_Photoeyes_TICK(me);
            me->_output.event.Cnf = 1;
            ConveyorModel_Photoeyes_CheckDiversers(me);
            me->_entered = true;
        }
        else {
            if (me->BagExited) {
                me->_state = 5; ...
            }
            else if (me->BagDiverted) {
                me->_state = 6; ...
            }
            else {
                me->_state = 0;
                me->_entered = false;
                continue;
            }
        }
    } break;
    ... 
    break;
}
Service Interface Function Blocks

Service interface function blocks are executed in the same way as other blocks, but as the corresponding XML files contain no executable specification, executable code is generated using pre-written templates instead. A number of service interface function blocks are also prescribed in the standard [15], such as PUBLISH and SUBSCRIBE communication blocks. During compilation, FBC identifies the C implementations for known service interface function blocks using a directory structure. The C implementations are manually developed before compilation, following the same code structure as generated code, using functions based on an FBTType data structure. As such, the appropriate template files for a block are simply copied to the output directory.

FBC also allows for programmatic commands within the C code template, including insertion of dynamic values into the templates and copying of addition files. This allows the compiler to generate multiple block implementations from a single C source file. For instance PUBLISH blocks have a variable number of ports, declared in the suffix of the block type. Using this suffix (for example 1 for PUBLISH_1 and 5 for PUBLISH_5) instance specific C files are generated with structures and reactive functions with one and five ports respectively.

6.1.2 Further FBC Development

The work from Yoong et al. did not generate code for IEC 61499 Adapters, and algorithms not written in C code were generated as empty C functions. The original FBC was therefore unable to compile the majority of IEC 61499 specifications developed with tools that do not natively support algorithms written in C code. This thesis developed FBC further to support HCECCs introduced in Chapter 4, adapters, and algorithms written in Structured Text (ST).

HCECCs

Generation of C code for HCECCs was implemented by modifying several aspects of FBC. Firstly, the compiler was modified to parse the custom XML syntax created for HCECC elements within a basic function block. Each custom XML element is handled by a separate function to parse the required data. Synchronous C code is generated from an internal representation for each HCECC by extending the algorithm that generates code for basic function blocks. While Chapter 4 describes the IEC 61499 compatible equivalent of HCECCs, for direct execution the semantics are implemented more efficiently.

When using the FBC compiler, code for an HCECC is generated as a set of nested and sequentially executed state-machines in a single reactive function. Because an HCECC block contains multiple state-machines, as the block data structure is generated, extra
The _state and _entered variables are added, using a unique prefix for each ECC. The algorithm shown in Algorithm 6.5 is used to generate code for the HCECC. This function replaces the generation of an ECC in the original GenerateFBE procedure used by FBC, such that event-data associations and clearing of events is still handled in the same way.

\begin{verbatim}
procedure GenerateHCECC(hecc)
  S := set of states in hecc;
  PARA := set of HCECCs in parallel with hecc;
  generate code to start an infinite for loop;
  foreach s ∈ S do
    generate new case for s in switch-statement;
    if not entered then
      execute state actions;
      if state is refined then
        generate code to reset refining HCECC;
      end
    else
      foreach transition condition, t, of state s to next-state n do
        generate code to test for t and restart switch statement with state := n;
        if t is always true then
          break;
        end
      end
      if state is refined then
        GenerateHCECC(state.refiningHCECC);
      end
      generate code to exit the infinite for loop;
  end
  generate code to end the infinite for loop;
  foreach para ∈ PARA do
    GenerateHCECC(para);
  end
end procedure
\end{verbatim}

Algorithm 6.5: Algorithm to generate C code for an HCECC.

As an example, for the HCECC in Figure 6.5, GenerateHCECC creates a switch-case statement for each state in the top level hecc passed as an argument to the function (line 6). Each case statement checks if the executing block has already entered the state and if not, its algorithms are called, and outputs are emitted completing the tick for this ECC (lines 7-12). In addition, if the state is refined, such as the Running state, the refining ECCTickECC and any ECCs in parallel with the refining ECC are all initialised to state 0 and their state entered flags are cleared (lines 9-11). If the state has already been entered, exiting transitions are evaluated in order of their priorities (lines 13-18). If a transition is
taken, as with standard ECCs, the switch-case statement is restarted. This immediately enters the new state, strongly aborting the execution of refined ECCs. If no transition is taken, then nested ECCs are executed in the fixed order defined by the hierarchy and parallel operators. For states which are refined, GenerateHCECC is called to generate the code for the refining HCECC (line 21). Similarly, after any HCECC is printed, if it has other HCECCs in parallel, they are also generated using GenerateHCECC (line 27).

In this way, the overheads of the equivalent composite function block are avoided by executing the ECC product on the fly. The use of a single function block with a single copy of each algorithm reduces the memory used by the specification. By generating a single reactive function instead of a set of reactive function for the composite function block implementation, performance is also improved. Section 6.1.3 presents some performance benchmarks, comparing HCECC with standard implementations of the same behaviour.

Translation of Structured Text

The IEC 61499 standard allows any language for algorithm specification, but explicitly mentions the IEC 61131-3 set of languages for backwards compatibility. Originally FBC only accepted algorithms written in C, because of a lack of translation tools. This restriction made FBC incompatible with function block models developed with many IEC 61499 tools. FBDK [39] supports Structured Text, Ladder Logic Diagrams and Java while 4DIAC [40] and nxtStudio [1] both currently only support Structured Text. Introducing translation of IEC 61131-3 Structured Text (ST) into C code therefore allows specifications from other IEC 61499 tools to be compiled with FBC.

The translator was implemented based on a grammar for ST, and ANTLR [87] was used to produce a parser and lexical analyser. The parser generates an abstract syntax
tree from arbitrary ST code, and from this tree, C code is generated. The generation of code follows the PLCOpen [69] semantics of structured text, creating semantically equivalent and human-readable C code. The excerpt in Algorithm 6.6 is taken from the conveyor model, which simulates the photo eye detection by comparing positions with the bags on the conveyor. The generated C code is shown in Algorithm 6.7 and remains very similar to the ST implementation due to the simple mapping of statements. To ensure the evaluation order of expressions is maintained, brackets are added around all relational and mathematical operators. An option was added to the translator to insert the required pointer dereference for the instance data generated by FBC (the variable me).

```
1 (* Code Comment *)
2 FOR pePtr := 0 TO PECount-1 DO
3   FOR bagPtr := 0 TO 19 DO
4     IF ( bagDSPos >= PEPositions[pePtr] ) AND
5       (bagUSPos <= PEPositions[pePtr]) THEN
6       (* this PE needs to 'detect' *)
7       PEDetects[pePtr] := TRUE;
8       EXIT;
9     END IF;
10    END FOR;
11 END FOR;
```

**Algorithm 6.6:** Sample ST code from Photo-Eye Simulator

```
1 /* Code Comment */
2 for(pePtr = 0; pePtr <= me->PECount-1; pePtr++){
3   for(bagPtr = 0; bagPtr <= 19; bagPtr++){
4     if( (bagDSPos >= me->PEPositions[pePtr]) &&
5       (bagUSPos <= me->PEPositions[pePtr]) ){
6       /* this PE needs to 'detect' */
7       me->PEDetects[pePtr] = true;
8       break;
9     }
10   }
11 }
```

**Algorithm 6.7:** Equivalent C code for Photo-Eye Simulator

When compiling a basic function block, if an algorithm is specified using ST, FBC invokes this parser to generate C code with the required pointer dereference. The generated C code can then be copied directly into the C function corresponding to the algorithm, in the same way that user written C code is inserted.
6.1 Compilation to C

![Figure 6.6: Illustration of Adapters](image)

**IEC 61499 Adapters**

Support for IEC 61499 adapters was also added to FBC to improve compatibility with the standard and other tools. Adapters are used in larger applications to reduce the number of inter-block connections, by grouping events and data together. The interface for an adapter, shown in Figure 6.6(a), looks the same as a normal function block. But from this single interface, two block types are created: the socket and plug, shown in Figure 6.6(b). The socket is the default configuration which is identical to the adapter’s design, whereas the plug is the mirror of this configuration, where inputs become outputs and vice versa.

Sockets and plugs are used within a network to describe the connections to and from the adapter. In a function block network, the plug or socket instances appear like any other block. At the interface level of the composite block however, the embedded adapters become ports. Instances of the socket sub-type become an input to the block, and instances of the plug sub-type become an output. This hides the exact connection details from higher-level block networks. Figure 6.7(a) shows a function block network with connections passing bag data from one conveyor to another in a loop. Figure 6.7(b) shows the same behaviour, but using adapters to simplify the network.

When a composite block containing a plug or socket is parsed, the adapter type is loaded from its own XML file. As both sub-types are expected to be used in the same system, the compiler generates the C `struct` for the socket from the interface data in the XML file. The inputs and outputs are then mirrored to create the C `struct` for the plug. The adapter C files are given the name of the adapter with the suffix of the specific sub-type, such as `BagDataAdapterSocket` and `BagDataAdapterPlug` for the adapter `BagDataAdapter`. Instance data for the correct sub-type is added to the C `struct` of the encapsulating composite block just as with other function block types.

The multiple arrangements for adapter connections in a composite block lead to several code generation procedures. Connections between adapter ports and ports of other function block types are declared in the standard XML element, so no change is needed.
to generate code. Within function block networks, connections between pairs of adapters are declared in a separate XML element which is also parsed into an internal representation. All types of adapter connections are examined by an extended EnsureAcyclic function, ordering them similar to standard event and data connections. After an acyclic execution order is found, code is generated for the two additional connection types: connections between two adapters and connections between an adapter and port.

Adapter connections between a socket and a plug represent bi-directional communication, and therefore require two segments of code. Before the function block containing the plug is executed, code is generated to update the input of the plug from the output of the socket. Similarly, code is generated to update the input of the socket from the output of the plug before for the function block containing the socket is executed. As both segments of code cannot occur before each other, one segment of code will read the delayed output values from the other block. The exact ordering is determined based on the extended EnsureAcyclic function. In the case where no other connections exist to force an execution order, one side of adapter connections will be arbitrarily delayed.

Connections between two adapters of the same sub-type occur when a composite block just exposes an embedded socket or plug port to its own interface. Figure 6.8 illustrates this behaviour, where the interface for Conveyor_Model contains plugs, prefixed with >> and sockets suffixed with >>. The adapter ports of Conveyor_Model are connected to this block’s interface, replicating the same plugs and sockets. In this scenario, data is copied from inputs in the composite block’s adapter to inputs of the embedded block’s adapter before the block is executed. After execution, output data from the block’s adapter is copied to the outputs of the composite block’s adapter.

For connections from an adapter to a block input port, code is generated to update the port with data from the output of the adapter before the block is executed. Similarly, for connections from a block output port to an adapter, code is generated to update the
input of the adapter after the block is executed.

### 6.1.3 Compiler Performance Results

To evaluate the impact of the changes, notably the addition of HCECCs, a suite of test applications was used to compare both quantitative and qualitative factors. The test applications, available from [88], differ in the number of blocks used and the complexity of algorithms, to provide a variety of implementations. HCECC implementations of each application are compiled with the extended FBC compiler, FBC–HCECC, and compared with ECC implementations using the research and academic de-facto standard FBRT [39], the commercial and industry proven nxtFORTE [45] and the original FBC [30].

Table 6.1 shows the applications tested, with the quantities of function block network and basic block design elements used. Unique blocks refers to the number of unique function block types, as some examples such as the Water Monitor use multiple instances of the same block. All other design elements are counted from these unique block types. Connections are the number of event and data connections used in any function block network. States and Transitions are the total number of states and transitions used in all basic blocks. With the exception of the ConveyorController example, the ECC and HCECC implementations for each application were developed independently according to the requirements of the application. The ConveyorController was developed solely as an HCECC, and then converted to a composite function block by applying the semantics of each operator. For HCECCs refined states are recorded as one state plus the number of states embedded within it.

In all of the examples, the number of states and transitions are the same or fewer in the HCECC implementation. The operators given to HCECCs allow more behaviours to be compressed into a single block, making them faster and easier to develop and maintain for the same behaviour. The reduction in the number of blocks used for many of the applications has the most significant impact on the ease of development and maintenance.
of an application. This is because a change in a single block’s interface will result in changes in the connections of the encompassing composite block, and possibly in the connected blocks.

To compare performance, a test file was generated for each application containing one million pseudo-random input vectors. Each vector contains a random single event, as well as random values for each data input. For each application, a test bench was developed that is specific to each implementation. The results obtained exclude the time taken to execute the test bench without the application under test. The run-times and synchronous applications were executed on a dual-core 2.5Ghz PC, with 4GB RAM. The Java implementation synthesized for FBRT by FBDK was compiled using javac from Oracle’s Java Development Kit 6 (JDK) [89] and executed using the included Java Virtual-Machine (JVM) with default settings. nxtStudio’s implementation was deployed to nxtForte version 1.5. Because of the compilation and deployment method used by nxtStudio, the object code size for nxtForte is approximated by using FORTE [40], the open source runtime on which nxtForte is based. The C code from FBC and FBC-HCECC was compiled using Visual Studio’s [90] bundled C compiler, with the -O2 optimization switch.

Figure 6.9 shows the execution time for 1 million input event reactions for each application. Due to the scale of the differences, a logarithmic scale is used. The values shown on the graph were obtained by calculating the average after 10 consecutive runs of each program. In order to negate the effects of caching and just in time compilation by Java, the time for the first execution for each program was not used.

\(^{1}\) 16 Controllers are used together in benchmarks to form a larger BHS example

<table>
<thead>
<tr>
<th>Application</th>
<th>ECCs</th>
<th>HCECC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unique Blocks</td>
<td>Connections</td>
</tr>
<tr>
<td>Watch</td>
<td>3</td>
<td>21</td>
</tr>
<tr>
<td>Distribution Station</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>Drill Station</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Water Monitor</td>
<td>4</td>
<td>31</td>
</tr>
<tr>
<td>Cruise Controller</td>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>MP3Player</td>
<td>4</td>
<td>22</td>
</tr>
<tr>
<td>Conveyor Model</td>
<td>3</td>
<td>25</td>
</tr>
<tr>
<td>Conveyor Controller</td>
<td>5</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 6.1: HCECC and ECC Size Comparison
To better visualise the differences, the results were normalised in Figure 6.10 such that FBC-HCECC has the value of 1, and the vertical scale is logarithmic because of the range of results.

These results show that both synchronous approaches, FBC and FBC-HCECC, are always faster than the two run-time approaches from FBRT and nxtForte. This is due to the synchronous approach, which removes the need for complicated run time scheduling of events. The FBC-HCECC implementation is between 1.5 and 8 times faster than FBRT, and much faster than nxtForte, ranging from 32 to 177 times faster. Comparing FBC using ECCs, and FBC-HCECC using HCECCs, it appears that the predominant performance benefit, comes from the reduction in the number of blocks that are used and therefore executed. The FBC-HCECC implementation ranges from 1% faster in the large BHS example, to over 2 times faster in the MP3Player application.

Figure 6.11 shows the size of the code produced by each approach for each application.
The code synthesized by FBC is always smaller than FBRT, and even more so compared with FORTE. FBC-HCECC extends this improvement, generating code that is smaller than all other implementations. As with performance comparisons, this benefit likely arises from the smaller number of function blocks needed to represent the same behaviour. The cruise controller example benefiting the most, due to a reduction from five blocks down to one.

![Code Size](image1)

**Figure 6.11: Test-bench Object Code Size**

To better visualise the differences, the same code size measurements are normalised in Figure 6.12 such that FBC-HCECC has the value of 1. Ranging from 2 to 12 times the size of code from FBC-HCECC, the results show the consistently larger code generated by FORTE. Code from FBRT and FBC are comparatively similar, averaging 3.5 and 3 times larger than FBC-HCECC.

![Normalised Code Size](image2)

**Figure 6.12: Normalised code size compared to FBC-HCECC**

These experiments show that the reduction in complexity afforded by HCECCs improves IEC 61499 designs, making them simpler and more maintainable. The semantics
of HCECCs also allow efficient execution, such that while an HCECC reflects a larger IEC 61499 compliant specification, direct compilation of HCECCs produces faster and smaller code than alternative approaches.
6.2 Compilation for Rockwell PLCs

The IEC 61131-3 set of languages intended to define a unified standard for the industrial control languages: Structured Text (ST), Ladder Logic Diagrams (LLD), Instruction List, Sequential Function Chart and IEC 61131-3 Function Block Diagrams (FBDs). PLC manufacturers however, failed to agree on a single implementation, resulting in specifications which are limited to a single manufacturer.

This work targets the popular range of PLCs from Rockwell Automation [23]. The developed compiler therefore assumes Rockwell execution semantics when translating function block components into ladder logic diagrams. It would also possible to generate code for PLCs from other manufacturers, by modifying the output file format and the generated LLD instructions.

Rockwell’s IDE for PLC programming, RSLogix [79] uses an XML format with the extension L5X to represent the configuration for a controller. The developed compiler, called FB2L5X, generates a full L5X file for a single controller from a IEC 61499 system description. The generated L5X file can be deployed and debugged using RSLogix.

6.2.1 Synchronous IEC 61499 to PLC Semantics

Recall from Chapter 5 that PLCs execute using the concept of a scan-cycle, identical to the synchronous tick. This similarity allows generation of code for PLCs to follow the approach used by FBC, creating a data structure and reactive function for each function block.

Table 6.2 summarises the relationships between function block elements and equivalent PLC design components. A device in IEC 61499 corresponds well with a PLC configuration for a single controller. Similarly, resources are equivalent to a periodically executed task. However, due to performance requirements it may be required to divide a single resource into multiple tasks and programs. Additional configuration is therefore required to define the allocation of blocks within a resource to various tasks, the next subsection describes how this is achieved.

Rockwell Add-on Instructions are an extended form of an IEC 61131-3 routine that includes instance data. They are utilised to implement the reactive function for each block type, as well as basic function block algorithms. The associated instance data is used to store the data structure for each instance of the block.

The data structure for each FBType is constructed using the same approach used by FBC. Instead of C structures, the IEC 61131-3 equivalent User-Defined Data types (UDTs) are used. In the UDT for a block type, variables for the event and data interface of the block are declared, with two copies of each data port for event-data associations. An instance of this UDT is then passed by reference to all of the Add-on Instructions
created for a block. The translation of data port types is greatly simplified because the XML for IEC 61499 function blocks uses IEC 61131-3 data types for primitives. FB2L5X is therefore able to declare data ports using the same data type in the UDT.

As shown in Chapter 5, to minimise re-development time and improve performance, global declarations and variables were maintained during export from the original PLC implementation. FB2L5X supports this usage, by allowing arbitrary C files to be parsed along with the function block system. Variable declarations are stored in the IEC 61131-3 configuration as either global or program specific variables. IEC 61499 User defined data types, or C structures in an included C file are generated as user-defined data types in IEC 61131-3.

An important caveat of global variables is that an IEC 61131-3 task can be pre-empted, making accesses to global variables non-atomic. As such, the user must either not split a resource into multiple tasks, or design the function block specification taking this into account. For example in the re-engineered baggage handling system, the slow task samples the variables from the fast task, creating a new temporary copy. In this way, should the task be pre-empted, the slow task will continue execution using consistent values.

### 6.2.2 Task Allocation of Blocks

IEC 61131-3 controller configurations are divided into *Tasks* and *Programs*. Each task contains information detailing its execution, including execution period, and a list of programs which are called when the task is executed. A program is a set of functions, including a main function to be called by a task. From an IEC 61499 system, a translation map is required to form the equivalent execution architecture on the PLC, allocating IEC 61499 resources and blocks into *tasks* and *programs*.

In the ideal case, IEC 61499 resources should be directly mapped to programs, although this is possible by manually allocating blocks in this way. Because the developed
controller block includes functions from both fast and slow tasks, it is necessary to split up a resource for efficient redeployment to a PLC. As global variables are used to communicate between the controller blocks originally sourced from the fast and slow tasks, there are no connections between these blocks in the network for the controller resource. This allows the compiler to split these components without affecting the behaviour.

To automate reallocation, a custom XML format is used to allow the user to map function blocks and sets of global variables to programs, and programs to tasks. This allows the compiler to open a complete IEC 61499 system, which may also include the plant model, and only generate code for required function blocks. Global variables have been allowed due to their use in the original IEC 61131-3 implementation, and in the re-engineered BHS developed in Chapter 5.

![Figure 6.13: Translation of IEC 61499 into IEC 61131-3](image)

Figure 6.13 illustrates the configuration for the BHS example. This configuration describes the mapping of the blocks within the controller resource into two tasks to ensure that timing requirements can be met. Blocks from the controller resource which are allocated to the Fast task execute once per 26 ms, while other blocks execute in the slow task in the remaining execution time. These tasks communicate with each other using global variables. Any additional C files that are required must be specified in the configuration file, including whether they are to be generated as global or program-scoped variables.

The following sections describe the generation of user defined data-types and reactive functions for blocks networks and basic function blocks.

### 6.2.3 Generating UDTs for Function Blocks

An instance of the EncStateToEncCount block shown in Figure 6.14 is used for each conveyor controller. It translates the output signal of the rotary encoder attached to the conveyor section into a value representing the distance travelled by the conveyor belt.
This block runs in the fast task of the controller, because at the belt’s top speed the value can change once every 26ms.

Algorithm 6.8 shows the L5X representation of the User Defined Data-Type (UDT) for the EncStateToEncCount block. Similar to the generation of C code by FBC, every event and data port is created as a variable in a user defined data-type. The storage element for input events is declared on line 3, then the input event, Req, is declared as part of the element on lines 4-5. The input variable ConvID is declared on line 9, where the prefix ‘_’ indicates that it is the interface copy. The internal buffered version, used in algorithms and transition conditions is declared on line 10. Because the block contains an ECC, the additional variables _state (line 17) and _entered (line 18) are added to the UDT. For blocks with HCECCs, both of these variables are duplicated for each ECC, using a unique prefix for each ECC. Alternatively, for blocks containing a function block network, instances of the required block types are declared in the UDT.

![Figure 6.14: Encoder State to Encoder Count Block](image)

### 6.2.4 Translating Block Networks

The current procedure for generating LLD for function block networks adopts the original approach used by FBC, where all inter-block communication is delayed until the next tick. Algorithm 6.9 shows the procedure for generating code from a function block network. Where all inputs are updated from outputs (lines 5-7), all blocks are executed (lines 8-10), then all outputs are written (lines 11-13). This approach is the simplest to implement, because execution of network blocks can be arbitrarily ordered with no consequence to application behaviour. There is only a minor performance cost compared to the instantaneous communication approach presented by FBC.

Figure 6.15 shows a part of the conveyor controller network used in the baggage handling system. The L5X generated for the network is shown in Figure 6.10. The Addon-Instruction begins with meta-data and input parameters (lines 1-9). The generated
Compilation of IEC 61499 Function Blocks

<DataType Name="EncStateToEncCountStruct" Family="NoFamily">
  <Members>
    <Member Name="_input" DataType="SINT" />
    <Member Name="Req" DataType="BIT"
      Target="_input" BitNumber="0" />
    <Member Name="_output" DataType="SINT" />
    <Member Name="Cnf" DataType="BIT"
      Target="_output" BitNumber="0" />
    <Member Name="_ConvID" DataType="INT"/>
    <Member Name="ConvID" DataType="INT" />
    <Member Name="EncoderIncr" DataType="INT" />
    <Member Name="EncoderIncr" DataType="INT" />
    <Member Name="EncoderState" DataType="INT" />
    <Member Name="EncoderState" DataType="INT" />
    <Member Name="_EncCount" DataType="LINT" />
    <Member Name="EncCount" DataType="LINT" />
    <Member Name="_state" DataType="INT" />
    <Member Name="_entered" DataType="BOOL" />
  </Members>
</DataType>

Algorithm 6.8: Encoder State Data Structure in L5X

procedure LLDGenerateNetwork(fb)
  IC := set of all connections to embedded block inputs;
  instances := set of all embedded block instances;
  OC := set of all connections to outputs of fb;
  foreach (source, destination) ∈ IC do
    generate code to assign destination to the value of source;
  endforeach
  foreach (blockType, instanceData) ∈ instances do
    generate code to execute blockType with a reference to instanceData;
  endforeach
  foreach (source, destination) ∈ OC do
    generate code to assign destination to the value of source;
  endforeach
end procedure

Algorithm 6.9: Algorithm for the compilation of function block networks into LLD

Rung 0 (lines 10-15), which sets the input event me.CONV0._input.Init, includes a branch to enable the event on the presence of either me.START._output.WARM or me.START._output.COLD. Because this is generated before the execution of the blocks, this provides inputs with the value of outputs from the previous tick. Rungs 7-9 (lines 18-30) execute the Add-on Instructions for the three function blocks sequentially and in an arbitrary order. As this network is within a resource, no code is generated for output connections.
6.2 Compilation for Rockwell PLCs

6.2.5 Translating ECCs

Algorithm 6.11 shows the procedures used to generate LLD for the execution of a basic function block, including either an ECC or HCECC. LLDGenerateBFB generates code that clears previous output and input events (lines 7 and 13) and manages event-data associations (lines 9-11 and 14-16). The LLDGenerateHCECC procedure, also shown in Algorithm 6.11, is called on line 12 to produce code for the ECC or HCECC.

The HCECC or ECC generated by LLDGenerateHCECC uses an if-else statement for each state. The infinite for loop generated by FBC is instead created with start and end labels around the if-else statements. This allows the execution to jump back to the start when a new transition is taken, and jump to the end after a state has been entered. As in FBC, the HCECC for a refined state is printed after the exiting transitions are tested, and parallel HCECCs are printed after the END label of the original HCECC.

Figure 6.17 shows the LLD generated from the ECC in Figure 6.16, where the numbers down the left-hand side indicate the rung number. Run 0 clears the output events, and Run 1 updates the three variables associated with the input event REQ. Rungs 2-5 are generated by LLDGenerateHCECC, with a single rung per state. The rung for each state
\begin{algorithm}
\begin{verbatim}
<AddOnInstructionDefinition Name="DEV1Controller"
    CreatedBy="UoA\gsha041" >
    <Parameters>
        <Parameter Name="me" DataType="DEV1ControllerStruct"
            Usage="InOut" />
    </Parameters>
    <Routines>
        <Routine Name="Logic" Type="RLL">
            <RLLContent>
                <Rung Number="0" Type="N">
                    <Text>
                        [XIC(me.START._output.WARM),
                        XIC(me.START._output.COLD)]
                        OTL(me.CONV0._input.Init)
                    </Text>
                </Rung>
                ...
                <Rung Number="7" Type="N">
                    <Text>E_RESTART(me.START)</Text>
                </Rung>
                <Rung Number="8" Type="N">
                    <Text>
                        ConveyorControllerParamatised_WithNetwork(me.CONV0)
                    </Text>
                </Rung>
                ...
            </RLLContent>
        </Routine>
    </Routines>
</AddOnInstructionDefinition>
\end{verbatim}
\end{algorithm}

Algorithm 6.10: Excerpt of generated controller network Add-on Instruction

Figure 6.16: ECC for Encoder State to Encoder Count
6.2 Compilation for Rockwell PLCs

```plaintext
procedure LLDGenerateBFB(fb)
  IE := set of all input events in fb;
  OE := set of all output events in fb;
  ID := set of all interface input data in fb;
  IDt := set of all internal input data in fb;
  OD := set of all interface output data in fb;
  ODt := set of all internal output data in fb;
  generate code to clear all output events in OE;
  foreach ie ∈ IE associated with id ∈ ID do
    generate code to update idt with id whenever ie occurs;
  end
  LLDGenerateHCECC(fb, hcecc);
  generate code to clear all input events in IE;
  foreach oe ∈ OE associated with od ∈ OD do
    generate code to update od with odt whenever oe occurs;
  end
end procedure

procedure LLDGenerateHCECC(hcecc)
  S := set of states in hcecc;
  PARA := set of HCECCs in parallel with hcecc;
  generate unique label for start of hcecc;
  foreach s ∈ S do
    generate new if or else-if clause for s;
    if s has not yet been entered then
      foreach action a of s do
        generate code to call algorithm and emit output event;
      end
      if s is refined then
        generate code to initialise refining HCECC;
      end
      set state to entered;
    end
  endforeach transition condition, t, of s to n do
    generate code to test for pre(t) and assign next state to n;
    Restart execution of the hcecc’s if(state) statement;
  endforeach if s is refined then
    LLDGenerateHCECC(s, refiningHCECC);
  end
  foreach para ∈ PARA do
    LLDGenerateHCECC(para);
  end
  generate unique label for end of hcecc;
end procedure
```

Algorithm 6.11: Algorithms to compile basic function blocks to LLD
contains a branch with two sub-rungs for the first and subsequent entries into each state. Where the instruction \textit{entered}[/[ tests the state has not yet been entered, and \textit{entered}][ tests that it has already been entered. Finally, Rung 6 clears the input events and Rung 7 updates the variable associated with the output event \textbf{CNF}.

When a state is first entered ([/entered), the algorithms are called as separate Add-on Instructions, and output events are set to true and the \textit{entered} flag is set to true (L) entered). After the state has been entered ([ entered), the sub-rung with exiting transitions will be processed in the next tick or scan-cycle. When a state has multiple exiting transitions, each transition is generated as a new sub-rung within the nested
6.3 Discussion

The open XML format and platform independence of the IEC 61499 standard simplifies code generation and enables multiple execution targets from a single specification. These compilers allow a single IEC 61499 specification to be executed either on a platform that supports C code or on Rockwell PLCs currently in use by industry. This improves system development, as developers can functionally test code before implementation, ensuring that software functions according to specifications before the hardware system is built. In this work the synchronous semantics have been adopted to allow for the generation of directly executable code, improving execution performance. Direct execution also provides the ability to target more platforms without the need for a pre-compiled run-time.

Specifications using HCECCs with the updated FBC compiler have significant performance advantages over not only run-time approaches, but also the original FBC compiler. Instead of flattening an HCECC into a collection of basic function blocks, executing the ECC product on the fly reduces performance and memory overheads. The addi-
tion of support for structured text algorithms and adapters into FBC makes it almost fully compatible with the standard, and other IEC 61499 tools. Additional work on the interoperability of the FBC compiler is also presented in Chapter 7.

IEC 61499 with HCECCs aid the development of complex and reliable systems, and FB2L5X allows manufacturers to utilise the new standard for design and testing on a PC before implementation on a PLC. FB2L5X currently lacks support for service interface function blocks and adapters, due to its early development status. Support for such features can be added in the future following the approach used by FBC. The generated PLC configuration will therefore need to be manually modified to include routines which connect hardware input and outputs with the function block network. Because the controller used in this case study was re-engineered from an original LLD implementation, the original LLD code can be imported to implement this as well as inter-device communication protocols.

Despite the resulting non-compliance with the IEC 61499 standard, global variables have been allowed in applications such as the BHS case study due to the data intensive nature of such specifications, and the importance of execution performance. Refer to Chapter 5 which discussed some alternatives to the current usage of global variables. Both compilers support the use of global variables by including or translating files containing definitions of global variables written in C code.
Development of IEC 61499 Tools

The lack of mature software and hardware technologies supporting the standard make development using function blocks more difficult. To fully utilise IEC 61499 for the development of complex systems, user friendly tools are required to encourage its use. The standard is now in the beginning of its industrial uptake, currently supported by two major industrial software vendors; ISaGRAF [16] and nxtControl [24], and a number of hardware platforms, including SIEMENS, Beckhoff, Wago and Advantech.

However, as presented in Chapter 2, IEC 61499 run-times, such as nxtForte used by nxtControl, are non-deterministic and use buffers for event propagation. In some function block network topologies, event buffers may also over-flow and lead to lost events and incorrect behaviour. This makes the development of reliable software very difficult. The synchronous function block compiler, FBC [30], on the other hand, provides efficient deterministic execution for function block applications, without the above problems. The generation of C code then allows an application to be cross-compiled for many embedded platforms.

In this chapter, the FBC compiler (Chapter 6) is extended and integrated into an existing commercial function block IDE, nxtStudio, as well as a new IDE called TimeMe. Integration with nxtStudio will give users of the FBC compiler access to unique IEC 61499 based HMI blocks for the design of graphical human-machine-interfaces. Using HMI blocks, users can graphically create visualisations for function block applications. Integration will also allow nxtStudio users the option to deploy to nxtControl’s nxtForte run-time or to
generate directly executable code depending on the target platform. The TimeMe IDE, developed by the University of Auckland [91], was designed with FBC as the core tool for IEC 61499 code generation. Because of the connection with the semantics used by the FBC compiler, HCECCs are also supported. Co-development of both FBC and TimeMe also allowed the implementation of a visual simulation mode for IEC 61499.

This chapter is organised as follows: Section 7.1 presents the integration of the compiler into nxtStudio, along with the implementation of extra functionality to interface with some of nxtControl’s unique features. Section 7.2 presents an introduction to TimeMe followed by the implementation of a simulation interface for synchronous function block applications.

### 7.1 Integration with nxtStudio

nxtStudio [1] from nxtControl [24] is a leading commercial IDE for the IEC 61499 standard. Through a relationship with nxtControl, the University of Auckland has been working to integrate the synchronous function block compiler into nxtStudio. This introduces the benefits of the synchronous semantics for IEC 61499 into the commercial IDE and allows complete model-view-controller design [11] for deterministic function block applications. As a corollary, increasing the exposure of FBC could potentially lead to favorable changes in the IEC 61499 standard, in particular, a well-defined and deterministic execution semantics.

nxtStudio began development in 2007 and offers a very easy to use interface for the development of all function block types. The IDE is bundled with nxtControl’s function block runtime nxtForte and features a graphical interface allowing the IDE to also deploy to remote instances of the runtime. nxtStudio is also able to automate aspects of re-mapping application components to different devices, automatically inserting the required service interface function blocks for communication.

An exceptional feature of nxtStudio is the integrated designer for component visualisations and human-machine-interfaces. Using a custom HMI block, a user describes the information flow between function block systems and their visualisation. In a separate view for the same block, the user can graphically design the visualisation of the component and write C# code to control its behaviour. When the HMI block is deployed, the visualisation, which can be executed on a Windows PC, connects to the run-time using a custom protocol to send and receive data.

Using nxtStudio the complete M.V.C. of a component can be developed and deployed. Integration of FBC into nxtStudio then allows efficient and deterministic implementation of the function block applications on a wide range of targets. A plugin was developed, providing a graphical interface for generating C code using FBC from within nxtStu-
dio. Changes were made to FBC to support nxtStudio projects and many of the service interface function blocks that nxtControl provides in the base nxtStudio library. The generated C code can then be compiled for any platform.

7.1.1 Plugin Development

The plugin shown in Figure 7.1 was developed by another PhD Student at the University of Auckland to provide an interface for the FBC compiler. The plugin, launched from within nxtStudio, allows the user to use FBC to generate C code and use Visual Studio tools to compile the code into an executable. The Compile button calls FBC in order to generate C code for the currently open function block or system. Make is used to compile the code for execution on a 32bit x86 processor using the C compiler provided by Visual Studio 2010 [90]. This executable will run on any Windows machine. Linux is also supported by manually running a different make file which is also generated by FBC. Run then launches the resulting executable, allowing quick testing on a PC, similar to using nxtStudio to deploy to a local instance of the nxtForte run-time.

The options ccode, run and acyc are appropriate for all applications within nxtStudio, but other FBC features are exposed to the user for completeness. C code is almost solely used by the plugin, because the work on integration of nxtStudio specific features presented in the following subsections were only implemented in C code. The run option generates code for the function block application, as well as a main function which calls the reactive function in a tight loop. This code can then be compiled and deployed on the desired processor. acyc automatically removes causal event-loops by automatically introducing a delay in one of the connections.

![Figure 7.1: nxtStudio Plugin Interface for the FBC Synchronous Compiler](image)

The remainder of this section explains the changes made to FBC as part of this thesis
to support compilation from nxtStudio projects, including implementation of nxtStudio’s HMI and inter-resource communication protocols.

7.1.2 Compiler Development

To enable FBC to compile a nxtStudio IEC 61499 project, several proprietary formats and protocols used by nxtControl needed to be implemented.

While the XML format of all standard function blocks is specified, nxtStudio uses some additional elements and attributes for many of its own custom functionalities. The NameSpace attribute was also added by nxtStudio to help differentiate identical block names from different projects. This tag is currently ignored by the compiler, as only applicable file paths should be given to the compiler.

Dynamic Service Interface Function Blocks

The most complicated change for nxtStudio support is the addition of custom tags and naming conventions to define service interface function blocks which have dynamically re-sizeable ports. This is used in several blocks, including some simple boolean function blocks provided as part of a nxtStudio library. A user is also able to define dynamic service interface function blocks (SIFBs), by following a design pattern described in the user manual. Graphically, dynamic SIFBs such as the ADD block pictured in Figure 7.2, have an icon in the bottom right that allows the user to drag in order to resize instances of the block.

![Graphical Representation of a Dynamic Service Interface Function Block](image)

Figure 7.2: Graphical Representation of a Dynamic Service Interface Function Block

Algorithm 7.1 shows the relevant part of the ADD block’s XML syntax. An extra Attribute node is added to the InterfaceList node, which indicates that the variable, in this case CNT, can be resized. The numbers after the variable name indicate the maximum and minimum values, 16 and 2 respectively, and a default value of 2 for the variable. Ports can now use the variable by adding the suffix `\${CNT}` to the variable name.
7.1 Integration with nxtStudio

1 <InterfaceList>
2   <Attribute Name="Configuration.GenericFBType.Counters"
3     Value="CNT;2;2;16" />
4 <InputVars>
5   <VarDeclaration Name="IN{CNT}" Type="DINT" />
6 </InputVars>
7 </InterfaceList>

Algorithm 7.1: Excerpt of XML for the declaration of dynamic ports

As service interface function blocks are just interfaces for arbitrary code, FBC includes a library of C code for each SIFB. The library was initially extended to include the additional blocks defined in nxtStudio. Then further changes were made to allow more dynamic SIFB code generation.

When the XML of a service interface function block is parsed, and dynamic ports are discovered, the compiler stores name and type information for the port. As the XML of the network containing the dynamic block does not directly contain information of the port’s size, the size of dynamic ports is determined based on connections to the block with the appropriate prefix and numbered suffix. Once the size of the port is known, code for the specific instance of the block can be generated. In general, FBC will use a single C file per function block type, but in several service interface function blocks, different instances will have different port types or sizes. As such, each instance of dynamically sized function blocks are given a unique sub-type, even if the same number and types of ports are used in another instance.

The library of service interface function block C code, used by FBC, already utilised a custom syntax for inserting programmatically generated code. These commands were extended to support dynamic port sizes and functions. A subset of the syntax for dynamic ports is shown in Algorithm 7.2 below, where IN is the name of a port with the suffix ${CNT}$. Each token between two @ symbols is a command to the compiler. This syntax follows the style of previous code templates. The begin_varlist token, followed by a # and the name prefix of the variable or port, changes the mode of the compiler. When the token is found, FBC loads the list of variables for the defined port name of the current instance and enables several extra tokens. @var@ inserts the full port name. @v#@ inserts the suffix of the variable. For instance, for port IN2 it will insert 2. The command also supports the variant @v#@ - 1@ for array indexing. The templating also allows for dynamic ports such as IN1 and IN2 to be of different types. This is implemented by determining port types based on the connections to the block instance.
HMI Communication Protocol

Integration with visualisations developed within nxtStudio is very beneficial for fast function block development using the model-view-controller design pattern. nxtStudio provides two blocks for Human-Machine-Interface (HMI) specification, one for visual design with a block interface and another for connection to the rest of the function block network.

- The CAT or Composite Automation Type block is similar to a standard composite function block, but it encapsulates the definition of an HMI block as shown in Figure 7.3(a). The HMI block is instantiated within the function block network of the CAT to connect the visual component with standard function block components.

- The HMI block, is a customised function block similar to a SIFB that can only be used within a CAT block. This block defines the interface for inputs and outputs to the visual representation of the component. Like a standard service interface function block, the HMI block can contain arbitrary code, but nxtStudio presents

```c
1  // Template Source Code:
2  @begin_varlist/#IN@
3  me->_OUT[v#/1] = me->_var@;
4  @end_varlist@
5
6  // Generated Code:
7  me->_OUT[0] = me->_IN1;
8  me->_OUT[1] = me->_IN2;
```

**Algorithm 7.2:** Excerpt C code in FBC’s service interface function block library

![CAT Block Structure](image1.png) ![Visual Editor for HMI Block](image2.png)

**Figure 7.3:** Human-Machine-Interface design elements in nxtStudio
a graphical interface for the visual design of the HMI as shown in Figure 7.3(b). nxtStudio also provides a C# editor to define the control logic for the visualisation.

In a project, all CAT blocks are tracked, so that an instance of a CAT block can be dragged into a position on a canvas for rendering. Once instantiated on the canvas, the visual representation defined by the HMI block is shown and can be moved around.

The function block system is then deployed and executed on a pre-defined IP address saved in the XML of the system. When the HMI canvas is launched from nxtStudio, it connects to a port which the nxtForte run-time is listening to. The canvas or client then subscribes to the instances it contains. Communication of changes to input ports of HMI block are then sent to the subscribed HMI client and user input from the HMI is transmitted to the output ports of the HMI block.

To allow systems compiled with FBC to communicate with the visualisation, nxtStudios HMI protocol was implemented by creating a separate HMIServer function. The server implements the communication protocol and initialises the HMI server socket. Figure 7.4 shows how communications to and from the HMI client are propagated to and from HMI blocks. For the Conveyor_CAT block, connections to the inputs of the embedded HMI block are written to the interface as with other blocks. When the HMIServer function is executed once per tick, it checks for the presences of an input event, reads the required data, and sends it to subscribed clients. The HMIServer then parses all messages in the UDP socket buffer and writes any received data to the output of the correct HMI block instance. Other blocks in the Conveyor_CAT function block network can be connected to these output ports, and thus receive data from the HMIClient.

![Diagram of HMI Communication](image)

Figure 7.4: Architecture of HMI Communication

When the canvas is configured and connected, the full baggage handling system can be visualised as shown in Figure 7.5. The colour of the conveyor sections represents the status. The top left of the HMI allows control over the execution speed of the simulation. The execution period defines how often the simulation should progress by a single tick.
The 'Progress Time By' input allows the user to specify the time elapsed between ticks. Together, these inputs allow the user to control the simulation speed relative to real time. The MCP or Main Control Panel button in the top right provides the user with controls over the system such as emergency stop, start and reset.

![Figure 7.5: HMI Visualisation of a simple Baggage Handling System](image)

An interface to the three bag sources is also shown at the top of the HMI canvas. The name string is set at compile time, and includes the ID of the conveyor the bag will enter. The bag source for conveyor 101 is automatic, and will continue to add non-overlapping bags while enabled. The other bag sources are manual, allowing the user to add a bag at any time with a given length but fixed width.

Upon entering a value for the bag length and clicking 'Add Custom' within the '201 Source' input section, the HMI client sends a packet of data to the HMIServer. The packet contains the full instance name of the block, in this case RES0.BagSourceLine2, and the included data and an identifier for which event was sent. The HMIServer function parses this data, and set the outputs of the correct block instance. In the next tick, the connected conveyor plant model block will receive this data and add a new bag to its model.

**Inter-resource Communication**

As a mature IEC 61499 development environment, nxtStudio is able to automatically insert service interface function blocks for communication when an application is distributed to separate resources. A group of blocks can be selected and the target device and resource can be changed to a resource on another device. In the case of the baggage handling system, Figure 7.6 shows the re-mapping of parts of the application from one resource to another. When a system is compiled for deployment, nxtStudio generates a separate system file which includes extra blocks which communicate events and data between resources. This allows simple re-distribution of function blocks for different deployment scenarios.
Figure 7.6: Redistributing some Conveyors within the Baggage Handling System example

Figure 7.7 shows the extra blocks added by nxtStudio. A resource may contain multiple CommChannel blocks, as they send data to a single remote resource. As all incoming data is sent to a single listening socket per resource, there is only one CommChannelUDPRecv block. The LINKPUBLISHER block is used to send a packet of event and data values. On the receiving side of a channel, events and data are separated into EVENTRECEIVER and DATARECEIVER. Importantly, there are no connections between the Communication Channel blocks and the related publish and receive blocks. The developed synchronous compiler plugin for nxtStudio, passes the deployment system, with explicit communication blocks, to the compiler.

Figure 7.7: Communication Blocks Automatically Inserted by nxtStudio
FBC’s implementation, illustrated in Figure 7.8, shows how the inserted blocks are used to communicate events and data with other resources. The system file sets the ID parameter for all publish and receive blocks to the correct channel for sending or receiving data. This information is then statically compiled into two functions. One function transfers publish data to the correct send buffer, and another function propagates received events and data to the correct port in the correct block. If the same event and data need to be sent to multiple resources, then multiple such publish blocks will be used.

![Figure 7.8: Architecture of Inter-Resource Communication](image)

When the REQ event of a LINKPUBLISHER block is triggered, the event and data information is stored in a custom binary format in the appropriate channel’s buffer. Then, once per tick, the communication channels sends all information in their associated buffers to their configured remote resource. The channel UDP receiver for the resource also decodes any received binary data and writes it to the interface of appropriate RECEIVE blocks.

Because there can be multiple receive channels on one resource, the SIFB library template which implements CommChannelUDPRecv supports a unique name per instance. Similarly, the LINKPUBLISHER and RECEIVER blocks are uniquely generated because of the variable number of data ports and data types.

This implementation of nXtForte’s protocol allows FBC to compile applications which can be quickly modified and deployed on different hardware. It also allows a multi-device system to be deployed to a mixture of nXtForte run-times and synchronous executables, which can then communicate with each other. Each device can therefore be executed on the most suitable platform.
7.1.3 FBC and nxtStudio Summary

With this level of compatibility with nxtStudio, the FBC synchronous compiler for IEC 61499 generates code to replace an instance of nxtForte. Because standard C code is generated, it can be compiled into an executable for many platforms. FBC, therefore, offers nxtStudio a compiler for generating deterministic code for safety critical or resource constrained applications. Similarly, FBC benefits from a commercial tool for complete model-view-controller component design. Fast re-distribution of the function block application also improves development time, allowing less focus on the deployment architecture.

One caveat arises however, because users of nxtStudio expect event-triggered execution semantics, whereas synchronous function blocks give different behaviour to ECCs (see Section 2.3.2). As such, the function block designer must be conscious of these differences when designing ECCs which will be executed using FBC. HCECCs are also not supported, making the description of complex behaviour more difficult.

7.2 TimeMe

TimeMe Studio is an integrated development environment based on Microsoft Visual Studio [90]. Developed at the University of Auckland [91], it aims to facilitate the design of safety-critical embedded control systems by adopting a synchronous semantics for IEC 61499. IEC 61499 is utilised for component oriented design of distributed systems and the synchronous semantics for function blocks provides the user with guaranteed determinism and absence of deadlocks [34]. Because TimeMe was developed in house, it has allowed complete control over feature development and tool integration.

The integration of the FBC compiler serves a vital role, allowing the execution of IEC 61499 specifications developed in TimeMe. Modifying the generated application allows a customised IEC 61499 simulation for synchronous function block applications which integrate with TimeMe's debug engine.

This Section explains the development and integration of FBC as an integral part of TimeMe. The generation of an interactive tick-by-tick simulation application is introduced enabling TimeMe to simulate models, similar to commercial synchronous design environments such as Esterel Studio [92] and SCADE [13].

7.2.1 Integration of FBC

The compiler is integrated into TimeMe as part of the Visual Studio concept of a Project Template. TimeMe introduces several project types which enable development of IEC 61499 applications, as shown in Figure 7.9. Each of the new templates automatically initialise many of FBCs parameters for a particular purpose, focusing on the type of code
generated.

![Figure 7.9: TimeMe Project Templates](image)

Project parameters are propagated to Visual Studio’s *Configurations*, which define environment settings for the build process. TimeMe’s configurations in the tool bar shown in Figure 7.10, further configures TimeMe to call different build tasks. The Run configuration calls FBC with arguments to compile the currently open IEC 61499 file and generate the main function to run it. Simulation mode also calls the FBC compiler, but with an argument which makes FBC generate code for an interactive simulation of the current function block file. More detail on simulation code follows in Sub-Section 7.2.2. The other configurations use separate tools not covered in this work.

![Figure 7.10: TimeMe Configuration Selection](image)

Because TimeMe stores more information per function block than the standard XML syntax allows, a custom XML file is used to save and load function blocks. To use IEC 61499 standard tools, such as FBC, TimeMe generates standard XML files for each file in the Project before invoking the tool.

During code generation, FBC traditionally printed status and error messages to the console. To aid integration with Visual Studio’s error and warning notice panel, an XML
7.2 TimeMe

format was defined to simplify parsing of these messages. The XML format is only printed when FBC is called by another program. In both cases compilation notices are written to std out and errors are written to stderr. The messages are shown to the user in the graphical form in Figure 7.11. Integration of messages allows completely hidden compilation by FBC, and by including the name of the block causing the error, TimeMe allows the user to double click on the message to open the block.

![Figure 7.11: Code Generation Error Messages shown in TimeMe](image)

When generating C code, FBC creates a Makefile and batch script which uses Visual Studio’s own nmake and cl, to compile the function block application. A Makefile for GNU make is also generated for compilation to other targets using a compiler such as gcc. For both run and simulation mode, the batch script is called by TimeMe to compile the generated C code. This provides the user with a single button interface to create an executable for any IEC 61499 application.

### 7.2.2 Implementation of Simulation Mode

TimeMe implements step-wise interactive simulation of IEC 61499 applications. This is unique in the function block domain, but is similar to simulation in Esterel Studio [92] and SCADE [13]. Visualisation and control of the simulated application is implemented within TimeMe. While FBC generates an application for the user selected function block model. The generated application connects to TimeMe and responds to commands with the state of the simulated mode.

Figure 7.12 shows the simulation process in terms of interactions between TimeMe, FBC and the simulated model. TimeMe calls FBC with a flag to generate simulation code. The generated code is then compiled using gcc or Visual Studio’s C compiler, and the resulting executable is launched. Upon execution, the simulation application connects to the IDE using a TCP socket, sending information of the initial state. TimeMe then provides the user with a graphical interface for controlling the execution, thus allowing the user to progress to the next tick or stop the simulation, as required.

The two components communicate using a custom application level XML protocol, allowing control requests from the IDE to the application, with detailed responses of
execution state information. The `Conveyor_Model` composite block, shown in Figure 7.13, will be used to illustrate simulation.
Simulation Protocol

After sending the initial tick response packet, the simulation application responds to requests from TimeMe. The two possible request packets are Tick Request and Stop.

The Tick Request Packet

The Tick Request packet contains the user defined values for all of the event and data inputs to the simulated block. Part of the XML for a tick request for the Conveyor_Model block is shown below.

```xml
<Tick>
  ...
  <Event Name="Tick" Value="ABSENT" />
  <Event Name="BagIn" Value="PRESENT" />
  ...
  <Var Name="BagInAngle" Value="0" />
  <Var Name="BagInID" Value="5" />
  <Var Name="BagInLength" Value="800" />
  <Var Name="BagInWidth" Value="200" />
  <Var Name="BagMergeAngle" Value="0" />
  <Var Name="BagMergeID" Value="10" />
  <Var Name="BagMergeLength" Value="750" />
  <Var Name="BagMergeWidth" Value="220" />
  <Var Name="MotorForward" Value="TRUE" />
  <Var Name="MotorReverse" Value="FALSE" />
  <Var Name="msSinceMidnight" Value="19550" />
  ...
</Tick>
```

The function block application parses the XML, setting the appropriate events and data, then it executes a single tick and sends the Tick response packet.

The Tick Response Packet

At the end of every tick, and also once the simulation is started, a tick response packet is transmitted from the application. The tick response packet includes status information and the values of all ports for each block in the system, with the exception of service interface function blocks.

The information in the XML is stored in a hierarchy preserving the nesting of function blocks. The entire status is sent to inform the IDE of the complete details of the
current application state, and to avoid any communication until the next tick. Below is an abbreviated Tick response XML packet, showing the structure of the XML, with composite block instances containing the nested block instances. Basic function blocks include information such as the current state, and the transition taken in the tick that was just completed. If the basic block contains an HCECC, the XML will contain a list of all current states and all transitions taken. Information from composite blocks consists of event connections that are present and all of the data connections associated with each present event.

<Tick>
<Instance Name="root" Type="Conveyor_Model">
<Events>
    <Event Name="BagIn" Value="PRESENT" />
    ...
</Events>
<Vars>
    ...
    <Var Name="BagInAngle" Value="0" />
    <Var Name="BagInID" Value="5" />
    <Var Name="BagInLength" Value="800" />
    <Var Name="BagInWidth" Value="200" />
    ...
</Vars>
<Instances>
    <Instance Name="BeltModel" Type="Conveyor_Belt_Model">
    ...
</Instance>
    <Instance Name="PhotoEyeModel"
        Type="Conveyor_PhotoEyes_Model">
        <Events>
            <Event Name="BagIn" Value="PRESENT" />
            ...
        </Events>
        <ECState Name="BagIn" />
        ...
    </Instance>
</Instances>
<Connections>
    <Connection Source="BeltModel.CNF"
The Stop Packet

Synchronous function block applications do not have an explicit termination, as they run in an infinite loop. The stop request, shown below, instructs the simulation application to terminate, with no response sent to the IDE. TimeMe then reverts to the standard function block view, with editing functionalities re-enabled.

<Stop />

Simulation Environment

When starting a simulation, TimeMe creates the input panel shown in 7.14 for the interface of the block under test. This panel allows the user to set multiple events to be present, and provide values for all data inputs.

Controlling the simulation adopts the standard debugging interface used by Visual Studio, using the panel shown in Figure 7.15, to trigger simulation requests. The Start button is used to start the simulation, and also to request another tick by creating the tick request packet and sending it to the application. The Stop button sends the stop request packet, and makes TimeMe exit simulation mode. The Restart Simulation button stops and re-creates the simulation application in order to restart the debugging session.
Simulation of function blocks extends Visual Studio’s debug interface, adding custom visual representations of the application under test. During simulation, all block editing functions are disabled. The status of each instance can be visualised, including current states and transitions in an HCECC (Figure 7.16) as well as network connections in a block network (Figure 7.17). Each tick response packet that is received updates the graphical view of the executing function blocks.
Figure 7.16: TimeMe Simulation: HCECC in ConveyorModel_HCECC

Figure 7.17: TimeMe Simulation: Network in ConveyorModel

When the tick response XML is received by TimeMe, the XML tree is also displayed in a panel such as Figure 7.18. This tick snapshot provides another way for the user to view the status of the simulate model.

All tick responses act like a stack trace for the execution of the simulated model, and so they are all stored in the Tick Stack shown in Figure 7.19. By keeping all previous data, it is possible to look back at the status of the model in previous ticks in order to debug an error.
Simulation Application

Figure 7.20 shows the structure of the simulation application generated by FBC. A top level XML node is passed to the simulated block, which adds its own XML node. Nested blocks then add themselves to the XML node of their parent block. Each block also has nodes for events and variables as well as block specific nodes for connections in the case of a composite block, and states and transitions for basic blocks. This XML tree remains constant during execution, with updates to attributes of static elements. Dynamic elements, such as connections, states or transitions, which vary in quantity are re-added to the XML tree each tick.

To read and write XML in C, the LGPL library mxml [93] is used, with two extra
functions to assist translating the common IEC 61131 data types to and from strings. Every Tick Request from TimeMe the simulation application parses the XML and sets the values of interface events and data. The tick function for the simulated block is then called, with each block also updating its XML attributes and dynamic elements. Once all blocks have been executed, the XML tree is written to a buffer and sent to TimeMe.

When the Stop Request packet is received, the application just closes the socket and terminates.

7.2.3 TimeMe Summary

Leveraging Visual Studio's well known interface, TimeMe presents an elegant user interface for the design of synchronous function blocks with HCECCs. Tight integration with FBC allows TimeMe to generate C code which can then be compiled for many target platforms. The deterministic semantics and efficient implementation from FBC provides a significant benefit over alternative IEC 61499 development environments. Finally, the inclusion of tick by tick simulation for any function block network enables complete off-line and in-depth testing, similar to several commercial IDEs for other languages.
The bottom-up design of industrial and embedded systems is inadequate for the large and complex systems of today. On its own, the IEC 61499 standard provides a component-oriented language for model-driven engineering, simplifying design with graphical self-documenting interfaces, networks and ECCs. The abstract event-triggered execution semantics also simplifies the reuse components on different platforms. However, the execution semantics are informal and ambiguous. In addition, ECCs are unable to succinctly describe complex behaviours. Yoong et al. [31] presented a significant step towards the use of IEC 61499 in safety-critical applications, by developing a formal synchronous semantics for function blocks. However, the semantics presented challenges due to the use of event-triggered ECCs to specify synchronous behaviour.

The goal of this thesis has therefore been to propose improvements to the IEC 61499 standard in order to develop complex and reliable automation and embedded systems. This has been achieved by addressing the shortcomings of the standard with a new synchronous semantics, and the development of tools for the use of IEC 61499. This chapter summarises the contributions of this thesis and presents suggestions for future research.
8.1 Hierarchical and Concurrent ECCs for IEC 61499

8.1.1 Summary and Contributions

Two aspects of the IEC 61499 standard hinder the development of complex and reliable systems. The ambiguous semantics of the standard are responsible for the variety of incompatible implementations. Further, the use of a run-time to implement the event-triggered semantics of the standard can in some implementations create non-deterministic behaviour due to the dynamical scheduling of block execution and buffering of events. In addition, to describe complex behaviours, ECCs require more states and transitions than other graphical state-machines.

The synchronous semantics developed by Yoong et al. [31] address the first concern, allowing deterministic and deadlock free execution of function block applications. As a result, the synchronous semantics are also able to generate directly executable code, which offers improved performance over run-time approaches. However, the proposed semantics were also hindered by the use of ECCs, because simultaneous events could not be monitored as they are not possible in the standard.

The primary motivation for previous ECC replacements was to address the ambiguous semantics of the standard, and thus, to introduce a formalism for state-machine design. As a result, the formalisms either fail to address the inadequacies of ECCs for exception handling or complex specification, or they do so at the expense of compatibility with the standard.

Chapter 4 presented a new synchronous semantics for IEC 61499, with unique features to address the deficiencies in ECCs. The developed extensions, called hierarchical and concurrent ECCs (HCECCs), adopt techniques from other graphical synchronous languages, introducing state-machine refinement and parallelism. With these additional features, HCECCs improve the description of complex behaviour and allow simultaneous events to be monitored while remaining compatible with the standard.

8.1.2 Future Work

The integration of more features from other Statecharts based languages, such as weak-abort, may be beneficial. Also, a more intuitive mechanism for communication between state-machines, potentially introducing instantaneous broadcast, would make HCECCs more powerful. Such features were not introduced in this work due to the desire to maintain compatibility with other IEC 61499 tools.
8.2 Reverse Engineering IEC 61131-3 into IEC 61499

8.2.1 Summary and Contributions

As IEC 61499 is designed as a replacement for IEC 61131-3, it is possible to describe the same applications in the newer standard. While the new standard offers many benefits, industrial manufacturers are likely to delay adoption as redesigning a system from scratch is time-consuming and it means that previous design effort in IEC 61131-3 is lost.

Related work such as [61, 70, 72] present approaches which re-create the semantics of IEC 61131-3 languages as networks of function blocks. This leads to an application that is harder to maintain than an original IEC 61499 implementation, as it will consist of additional function blocks or states which manage semantic differences between the implementation. Although approaches similar to this would allow arbitrary distribution of the resulting function block network, it is unlikely that distribution would be desirable at the level of a single routine.

In contrast, Dai et al. [73, 81] offers an approach which re-creates an IEC 61131-3 application in IEC 61499 by following a class-oriented instead of component-oriented architecture. After a solely manual translation, the class-based architecture removes the need for global variables, as all of the required functions for a task are encapsulated in a single block. This approach does not take full advantage of ECCs for design, but would allow all IEC 61131-3 code to be reused as algorithms within a basic function block. As a single component is divided into many functional classes, the approach is not suited for model-view-controller design.

Chapter 5 presents a technique for the re-engineering of IEC 61131-3 applications into IEC 61499 that reduces the time-cost by maintaining aspects of the legacy implementation. A tool has been developed which allows the automatic translation of IEC 61131-3 routines (designed in Rockwell’s RSLogix 5000 IDE) into C functions for use in a function block design. An approach is then presented for the development of a function block component architecture, which then utilises the generated C code. Together, this allows for a fast migration from IEC 61131-3 into IEC 61499 to harness the advantages of the new standard. Although it is possible to use LLD and ST in function block algorithms, more platforms can be targeted by translating existing code into C code. The function block architecture which encapsulates these algorithms then enables better code reuse and simpler maintenance with the more graphical function block standard.

The allowance for global variables in IEC 61131-3 is handled in the IEC 61499 implementation by allowing the inclusion of C header files when FBC generates C code. This significantly reduces the changes required for a new implementation, at the cost of a design that is no longer fully component-oriented as per the standard. Distribution is still possible however, but manual intervention is required to introduce communication
function blocks to exchange the data between devices.

8.2.2 Future Work

While code migration is useful, the complete automatic translation of a sub-set of IEC 61131-3 applications into IEC 61499 function blocks provides much greater encouragement to industrial manufacturers. This poses an interesting research question regarding the plausibility of translating a flat application into a component-oriented function block architecture. More recent work from Dai et al. [94] presents an ontological approach that enables automated migration of code between IEC 61131-3 and IEC 61499 implementations. This allows users to correct issues in either the IEC 61131-3 or IEC 61499 specification. However, the technique relies on a known software architecture, requiring changes to the original implementation as well as project specific configuration. Combining code translation from this thesis with the ontological approach from Dai et al. would allow the use of more expressive C code in the IEC 61499 implementation.

A more resilient approach for global variables also needs to be investigated that maintains the IEC 61499 standard’s goal of distributable components. However, communication between blocks adds a buffered instance of a variable, requiring more memory and processing time. What is needed is a notion of internal variables for a composite block or resource, allowing instantaneous access to variables for function blocks within the network which cannot be pre-empted. It may also be possible for execution approaches to silently introduce global variables where appropriate to reduce overheads.

8.3 IEC 61499 Code Generation

8.3.1 Summary and Contributions

To execute an application using the new semantics presented in Chapter 4, a compiler is required to parse the application and generate code. Generation of code that is executable on multiple platforms is very useful for heterogeneous distributed systems. In addition, for industrial manufacturers, it is likely that PLCs will continue to be used due to their proven reliability.

While many other works utilise a run-time for the execution of event-triggered function blocks, the synchronous semantics from Yoong et al. [53] and the new semantics developed in Chapter 4 allow for execution without a run-time. The synchronous compiler for IEC 61499 from Yoong et al. generates C code, which allows execution on any platform that supports compilation from C. Industrial PCs can currently be targeted in variety of ways, including commercial support for Beckhoff [95] and Wago [96] devices when using
8.3 IEC 61499 Code Generation

nxtStudio and ISaGRAF. However, IEC 61499 applications cannot currently be executed on IEC 61131-3 PLCs due to lack of support from compilers.

Chapter 6 presents two compilers, which generate executable code for two types of platforms. The FBC compiler developed by Yoong et al. is modified and extended to generate C code for: the synchronous semantics of HCECCs, algorithms written using Structured Text and IEC 61499 adapters. A new function block compiler called FB2L5X is also introduced, that generates an IEC 61131-3 configuration from an IEC 61499 system for execution on a PLC.

The C code generated by the modified FBC can be compiled and executed on any platform that supports C. Using HCECC models, the extended compiler was evaluated against other ECC based execution approaches using a series of benchmarks. The comparison showed that HCECC models compiled with the modified FBC execute faster and require less memory than equivalent ECC models using other approaches, including the original FBC.

FB2L5X generates LLD for use in Rockwell’s RSLogix IDE [79]. This type of compiler allows manufacturers to employ IEC 61499 with HCECCs for improved design specification and code reuse, whilst continuing to use industry-prove PLCs for execution.

Together, the two IEC 61499 compilers allow a single specification to be executed on any platform that supports C code, as well as on PLCs currently in use by industry. In this way, the functionality of a specification can be tested off-line on a PC, prior to deployment on the hardware chosen for implementation.

8.3.2 Future Work

The FBC compiler is now reasonably mature, and after integration with development tools in Chapter 7, it has been tested with many applications. It has been noted however, that the code generated by FBC could be optimised further by reducing the number of variables instantiated and using C pointers to share data. Such optimisations will reduce the memory requirements, and further optimisations could be investigated in order to improve the average reaction time.

FB2L5X has been not yet been tested to a high degree, and does not yet support all elements in the IEC 61499 standard. Of particular interest is the implementation of service interface function blocks, which will allow a function block application to be compiled with either FBC or FB2L5X depending on the target platform. Topological sort could be added to intelligently remove non-causal cycles and allow instantaneous communication in function block networks. This would improve execution performance, similar to its effect on FBC generated code. Once FB2L5X fully supports the standard, generation of LLD for other PLC manufacturers could also be investigated.
8.4 IEC 61499 Development Tools

8.4.1 Summary and Contributions

As a new standard, there are only a few tools that fully support IEC 61499, and only two commercial development environments. Open source or free development environments such as 4DIAC [40] and FBDK [39] are not of commercial quality, lacking usability features. Of the commercial tools, nxtStudio [1] has the greatest compatibility with the standard and provides the developer with a very user-friendly interface. nxtStudio also includes a unique feature, which seamlessly integrates a visualisation or human-machine interface to function block applications.

Chapter 7 presents the integration of the updated FBC compiler with nxtStudio and the newly developed TimeMe IDE [91]. In nxtStudio, FBC allows a more efficient synchronous implementation of function block applications than is possible using the included nxtForte [45] run-time. The TimeMe IDE is tightly coupled with the synchronous semantics for IEC 61499 developed in this thesis in order to create an IDE for the development of reliable specifications using IEC 61499. The integration of the updated FBC into TimeMe, with support for the development of HCECCs, improves the user experience and automates the use of the compiler. As a dedicated IDE for FBC, a user-controlled tick-by-tick simulation was also developed, similar to commercial tools such as SCADE [13].

8.4.2 Future Work

A drawback of the use of FBC with nxtStudio is that users will design ECCs assuming an event-triggered execution, however, the compiler uses a synchronous semantics. This discrepancy may result in events being missed, which is difficult to debug unless the user is aware of the semantics. Because it was not possible to modify aspects of nxtStudio directly, HCECCs were not introduced. Thus, even if the user is aware of the semantics, accurate designs are not possible in the case of simultaneous events which can only be handled by using HCECCs. Should it be possible, a plugin for nxtStudio that allows the use of HCECCs would allow simpler designs and allow synchronous behaviour to be accurately described.

The TimeMe IDE, on the other-hand, requires further development to improve the user interface and make it as usable as commercial IDEs. In addition, because TimeMe is based on a formal synchronous semantics for IEC 61499, formal analysis tools such as timing analysis and verification tools can also be integrated in the future. However, more research is required to achieve these features successfully.

FBC may also be useful replacement for the run-time used by ISaGRAF IDE which uses a proprietary cyclic run-time for execution of function block applications. This
cyclic implementation loses compatibility with the event-triggered standard, and requires
the user to define the execution order of blocks in a network Integration of FBC into
ISaGRAF will provide the benefits of the synchronous function block semantics, allowing
an implementation that is compatible with the standard and remove the requirement of
user-defined block execution order.

8.5 Concluding Remarks

As industrial control systems increase in size and complexity, manufacturers are burdened
by the existing use of IEC 61131-3 and centralised controllers. IEC 61499 includes nu-
umerous features for future industrial systems, offering significant benefits with regard to
code reuse and distribution. However, on its own, the standard lacks a formal execution
semantics and is not as capable as other languages for the specification of complex
behaviour.

This thesis has presented a number of developments that improve the suitability and
capability of IEC 61499 for reliable and complex software development. The semantics
of HCECCs uniquely provide developers with hierarchy and concurrency within basic
function blocks. This enables more succinct designs and allows the description of sim-
ultaneous events within the synchronous execution approach. Directly executable code
can then be generated for a number of platforms using the improved FBC compiler and
the new FB2L5X compiler. When using HCECCs, the code generated by the improved
FBC compiler is also shown to perform faster and with a smaller memory footprint than
alternative approaches.

Reverse engineering of an IEC 61131-3 configuration into IEC 61499 is demonstrated
using an example baggage handling system. The semi-automated approach retains exist-
ing development work and encourages the use of the function block standard by simpli-
fying adoption. Finally, integration of the FBC compiler into two IDEs gives developers
a graphical interface for the design of reliable specifications using IEC 61499. Together,
these contributions improve the quality of IEC 61499 applications and offer efficient im-
plementation on multiple target platforms.
References


REFERENCES


REFERENCES


[57] International Electrotechnical Commission, Committee Draft for Vote: IEC 61499-

[58] Object Management Group,, “OMG Unified Modelling Language (OMG UML) -


[60] D. Drusinsky and D. Harel, “Using statecharts for hardware description and syn-


[64] N. Hagge, “Integrating cnet and iec 61499 function blocks,” in Emerging Technologies
509.


visual flowchart control programs using net condition/event systems formalism in

Computer Languages, Volume 27, Number 1, 2001.


