Copyright Statement

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

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

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

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

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the Library Thesis Consent Form and Deposit Licence.

Note: Masters Theses

The digital copy of a masters thesis is as submitted for examination and contains no corrections. The print copy, usually available in the University Library, may contain corrections made by hand, which have been requested by the supervisor.
Modelling and Synthesis of Safety-critical Software with IEC 61499

Li Hsien Yoong
June 2010

Supervisor: Partha S Roop
Co-supervisor: Zoran Salcic

A thesis submitted in partial fulfillment of the requirements of Doctor of Philosophy in Engineering
Abstract

Software forms a central part of modern industrial control systems. It is routinely used, nowadays, to control a variety of physical processes through an integration of computing elements with sensors and actuators. The falling cost of digital electronics have also led to an increasing use of multiple computing units to control large distributed systems in a networked environment.

The design of such systems is complex, and requires programming methodologies that seamlessly support the specification of distributed and concurrent software. The methodologies should further be easy to understand, and should provide design artefacts that naturally support reuse. Most existing techniques for developing control software, however, are unable to support these features.

Recognizing this need, the IEC 61499 standard has been proposed as a standard for developing distributed industrial control systems. This standard prescribes a component-oriented approach for developing distributed control software, based on function blocks. The graphical nature and the encapsulation offered by function blocks provide an intuitive way to describe software in a reusable manner. Executable code can further be automatically synthesized from these function blocks. This helps to simplify the task of programming, while ensuring more reliable software.

The standard, however, lacks the semantic rigour necessary for the automated verification and unambiguous execution of function blocks. In particular, the model of concurrency for a network of function blocks running in a centralized or distributed fashion is not clear. Several scheduling techniques have been proposed to overcome this problem. Various run-time environments, each adhering to a particular scheduling policy, have correspondingly been developed to execute function blocks. This has resulted in incompatible behaviours, as well as complications in the formal verification of function block programs.

This thesis addresses these problems by proposing a formal model for distributed IEC 61499 systems based on the globally asynchronous locally synchronous (GALS) paradigm. For a centralized implementation, function block networks are executed synchronously, while distributed implementations are executed as a collection of synchronous islands that communicate with each other asynchronously. The semantics
for synchronous execution is provably correct for any arbitrary composition of function blocks. Moreover, run-time scheduling overhead is eliminated, as all scheduling decisions can be made before a program is run. These semantics further enable formal verification of function blocks using the well-known concept of synchronous observers.

The approach proposed in this thesis also allows communication in distributed systems to be specified in an abstract way, which does not yet imply any particular implementation. This abstraction can be automatically refined to obtain various implementations with different trade-offs. This has been done in a manner that is fully compatible with the IEC 61499’s notion of communication function blocks.

A prototype compiler has been developed to synthesize either centralized or distributed code from function block programs. The code generated by this compiler is markedly superior to that produced by existing techniques in terms of execution speed, as well as code size. These results demonstrate the viability of the ideas presented in this thesis for the development of practical industrial control software.
Acknowledgements

This thesis is a product of contributions from many people. I have benefited greatly from the many stimulating discussions that I had during the course of my research. First among them is my supervisor, Partha Roop, who provided constant challenge and encouragement to me up till the very end of my work. His detailed feedback on many of my ideas has been invaluable.

I am also grateful to Zoran Salcic, who also supervised me in this thesis. He was always able to provide fresh perspectives to my work that enabled me to see the wider picture of things. I must also acknowledge Valeriy Vyatkin, whose early input in my work provided me with many initial ideas. His comments have certainly shaped the direction of my research.

I have also been privileged to have a wonderful set of cubicle mates throughout my work. Cheng Pang provided substantial help in getting me started with function block programming, as well as in the early phase of my compiler design. I have gleaned a lot through my discussions with him on the problems with function block programs.

I thank Gareth Shaw for being the “guinea pig” for my compiler right from the early days of its creation. Frankly, at that stage, I did not have as much faith as him!

My appreciation also goes out to Roopak Sinha. His insights into verification were most helpful. Discussions with Avinash Malik have always been very stimulating and, at times, entertaining. More often than not, my conversations with him that began with research questions, land up digressing into something altogether meaningless.

Finally, I would like to acknowledge Simon Yuan and Sidharta Andalam, who have been wonderful companions during the humdrum of work. Without them, doing a Ph.D. would have been less endurable.
# Contents

Abstract .......................................................... i

Acknowledgements .............................................. iii

Contents .......................................................... iv

List of Figures ..................................................... x

List of Tables ....................................................... xi

1. Introduction .................................................... 1
   1.1 IEC 61499 preliminaries ...................................... 4
   1.2 Ambiguities in function block execution ....................... 7
       1.2.1 Evaluation of transitions within function blocks ........ 8
       1.2.2 Compositions of function blocks in a network ........... 9
   1.3 Formal model for function block systems ..................... 10
   1.4 Software synthesis ........................................... 12
   1.5 Abstract communication patterns .............................. 13
   1.6 Research contributions ....................................... 13
   1.7 Thesis organization .......................................... 15

2. Literature Review ............................................... 17
   2.1 Formalisms for function blocks ............................... 17
   2.2 Run-time execution models for function blocks ............... 19
   2.3 Techniques for dealing with compositions .................... 21
   2.4 Distribution of function block programs ..................... 23
   2.5 Remarks ..................................................... 24

3. Formal Model for IEC 61499 Function Blocks .................. 25
   3.1 The synchronous programming paradigm ....................... 26
   3.2 Synchronous model for function blocks ....................... 28
       3.2.1 The cruise control example ............................ 32
   3.3 Semantics of synchronous function blocks ..................... 36
3.3.1 Formal semantics ........................................ 39
3.3.2 Definitions and proofs .................................. 45
3.4 Discussion .................................................. 49

4. Compilation and Verification of Function Blocks .............. 53
  4.1 Compiling function blocks to Esterel ..................... 54
  4.2 Translating the basic function block .................... 55
    4.2.1 Generating code for state transitions ............... 58
    4.2.2 Generating code for implementing algorithms ....... 60
  4.3 Translating a function block network .................... 62
  4.4 Automatic verification ................................ 64
  4.5 Discussion .............................................. 67

5. Efficient Code Synthesis from Function Blocks ............... 69
  5.1 Effects on scheduling order and communication .......... 72
  5.2 Code generation for function blocks .................... 73
  5.3 Translating basic function blocks ...................... 75
  5.4 Translating composite function blocks .................. 77
    5.4.1 Implementing delayed communication ............... 77
    5.4.2 Implementing instantaneous communication ......... 80
  5.5 Generating the graphical function block simulator ....... 82
  5.6 Discussion .............................................. 83

6. Distribution of Function Block Systems ....................... 85
  6.1 GALS model for distributed systems ..................... 86
    6.1.1 An example of a distributed system .............. 88
  6.2 Communication ........................................... 91
    6.2.1 Connections and channels ......................... 91
    6.2.2 Bounded lossless and lossy channels ............. 93
  6.3 IEC 61499 communication function blocks ................ 97
    6.3.1 Client-server communication function blocks ...... 98
    6.3.2 Publish-subscribe communication function blocks ... 101
  6.4 Implementation .......................................... 103
    6.4.1 Synthesizing communication function blocks ....... 104
  6.5 Discussion .............................................. 107
## CONTENTS

7. **Case Study and Results** ........................................... 109
   7.1 Modelling a baggage handling system .......................... 110
      7.1.1 Simulation and visualization of the BHS .................. 111
      7.1.2 Function block implementation of the BHS .............. 112
   7.2 Experimental results ............................................. 115
   7.3 Discussion .......................................................... 120

8. **Conclusion and Future Work** ..................................... 121
   8.1 Perspectives and future work .................................... 123
      8.1.1 Verification issues .......................................... 123
      8.1.2 Real-time requirements .................................... 124
      8.1.3 Visualization/simulation of control systems ............ 125
      8.1.4 Distribution issues ......................................... 126
   8.2 Final remarks ....................................................... 126

References ............................................................. 127

A. **Templates for the Communication Function Blocks** ........... 137
   A.1 Publisher template ............................................... 137
      A.1.1 Header file .................................................. 137
      A.1.2 Implementation file ........................................ 138
   A.2 Subscriber template .............................................. 139
      A.2.1 Header file .................................................. 139
      A.2.2 Implementation file ........................................ 140

B. **Research Articles Based on this Thesis** ...................... 143
   B.1 Published papers .................................................. 143
   B.2 Papers under review .............................................. 143
   B.3 Papers under preparation ........................................ 144
List of Figures

1.1 A function block network depicting a cruise control system. . . . . . . 5
1.2 Interface of the Throttle function block. . . . . . . . . . . . . . . . . . . . . . 6
1.3 Execution control chart (ECC) for the Throttle function block. . . . . 6
1.4 Ambiguities in state transitions of execution control charts. . . . . 8
1.5 Ambiguities in execution behaviour of a function block network. . . 10

3.1 Example of an Esterel program. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
3.2 State transitions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.3 Ambiguities in execution behaviour of a function block network. . . 31
3.4 Esterel code for the function block network of Figure 1.1. . . . . . . . 33
3.5 Esterel code for the ECC in Figure 1.3. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
3.6 Timing diagram for an execution trace of the program in Figure 3.5. . 35
3.7 Timing diagram showing the effects of “pipelining” on communication. 35
3.8 Simple function block network appended with ECC fragments. . . . 37

4.1 Algorithm to initiate the translation of an ECC into Esterel. . . . . . . 56
4.2 Algorithm to prepare an ECC for structured code generation. . . . . 56
4.3 Algorithm to generate Esterel code for a given ECC. . . . . . . . . . . . . . . . . . . 57
4.4 Illustration on how code is generated for state transitions. . . . . . . 59
4.5 Esterel code for the normal algorithm in the ECC of Figure 1.3. . . . 62
4.6 Esterel code for the function block network of Figure 1.1. . . . . . . . 63
4.7 The cruise control system function block network with an observer. . 65
4.8 Esterel code for an observer. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67

5.1 Illustration of a set of strongly connected function blocks. . . . . . . . . 71
5.2 Illustration of the effects of execution order and communication. . . 73
5.3 Algorithm to generate C code for a basic function block. . . . . . . . . 76
5.4 Code snippet illustrating the generated code for the ECC in Figure 1.3. . 78
5.5 Algorithm to construct a netlist for a function block network. . . . . . 79
5.6 Algorithm to topologically sort function blocks in a network. . . . . . 81
5.7 The graphical simulator for the Throttle function block. . . . . . . . . . 83
6.1 Example of a water level control system. .......................... 89
6.2 Distributed implementation of the water level control system. ... 90
6.3 Illustration of connections and channels. ............................ 92
6.4 Bounded lossless communication. .................................... 94
6.5 Illustration of the liveness property of an MDG. .................... 96
6.6 Client-server communication function blocks. ....................... 98
6.7 Sequence diagram depicting the four scenarios for connection establishment between the client and the server. .................... 99
6.8 Sequence diagram depicting server-initiated and client-initiated disconnections. ......................................................... 100
6.9 Sequence diagram depicting bidirectional data transfer between the client and the server. ............................................. 100
6.10 Publish-subscribe communication function blocks. .................. 102
6.11 Sequence diagram depicting the three scenarios for connection establishment between the publisher and the subscriber. .......... 102
6.12 Sequence diagram depicting publisher-initiated and subscriber-initiated disconnections. ..................................................... 102
6.13 Sequence diagram depicting normal unidirectional data transfer between the publisher and the subscriber. ......................... 103
6.14 Pseudocode for part of the C code generated for the publisher. .................................................................................. 105
6.15 Pseudocode for part of the C code generated for the subscriber. ............................................................ 106
7.1 Visualization of the baggage handling system simulation. ........... 111
7.2 Plant model for a conveyor. ................................................. 112
7.3 Function block network for a single conveyor. .......................... 114
7.4 Controller model for a conveyor. .......................................... 114
7.5 Comparison of benchmark execution times using various code generators. ............................................................... 117
7.6 Normalised benchmark execution times. .................................. 118
7.7 Comparison of benchmark code sizes using various code generators. ............................................................... 118
7.8 Normalised benchmark code sizes. ........................................ 119
List of Tables

3.1 Example of an input/output trace for the program in Figure 3.1 . . . . 26
3.2 Function block to Esterel translation map. . . . . . . . . . . . . . . 29
3.3 Kernel statements for synchronous function blocks. . . . . . . . . . . 37
4.1 Function block to Esterel translation map. . . . . . . . . . . . . . . 54
4.2 Structured Text (ST) to Esterel translation map. . . . . . . . . . . 61
4.3 Verification times for test programs. . . . . . . . . . . . . . . . . . . 66
The development of industrial control software has been an exciting and challenging field in engineering over the past few decades. It has provided a confluence for numerous advancements in technology, which have come together to meet the demands of a rapidly changing market. In particular, the falling cost of digital electronics has led to an increase in the pervasiveness and complexity of embedded systems [1]. Such systems, which perform dedicated computing functions as part of a larger application, now lie at the heart of all industrial control software.

In industrial systems, embedded computers are routinely used to monitor and control a variety of physical processes through a tight integration of computing elements, sensors, and actuators. Recent trends combining computing and networking in low-cost embedded computers have also led to the emergence of an increasing number of distributed control systems. This development has had a large impact on many industrial sectors, as applications once thought to be prohibitively expensive are now conceivable using standard embedded computing and networking solutions. In particular, the possibility of designing large distributed systems at reasonable cost has paved the way for many emerging technologies in the industrial sector. Examples of industrial applications involving distributed systems can now be found in automotive systems, material handling, manufacturing, process control, smart power grids, and in many other areas of industrial automation.

While readily available hardware and bandwidth have fuelled this growth, the development of tools and techniques for designing distributed industrial control software has been less pronounced. The ubiquitous “embedding” of computing units in various safety-critical systems [2] has further compounded this problem by fuelling the need for ever greater reliability in the software that controls them. Meanwhile, market pressures have resulted in a relentless push for ever cheaper and better solutions within shorter design times. Together, these challenges have led to greater
advocacy for a model-based development [3] of industrial control software. Model-based development emphasizes the use of models, or abstractions, in programming that better resemble the familiar concepts of a particular domain, rather than those of computer science. For example, a control systems engineer would be more inclined to think about a program in terms of the order of the system, its stability, and the response time, rather than the “implementation details” such as the algorithmic complexity, scheduling techniques, or deadlines. This abstraction helps to reduce programming errors by allowing more intuitive specifications of systems, and by facilitating the validation thereof.

A lot of embedded software, however, is still developed in an ad hoc manner, using low-level languages, like C. In the industrial control domain, the languages for programmable logic controllers (PLC), such as Ladder Diagrams, Structured Text, and Sequential Function Charts (SFC), continue to dominate [4]. These languages offer very primitive abstractions, which make the description of complex systems error-prone, hard to understand, and with little scope for reuse. Unfortunately, since industrial engineers tend to be experts in specific industrial domains rather than in software engineering, many of the recent advancements in software engineering remain inaccessible to the majority of them. Consequently, there is a need for a high-level abstract modelling technique that is both robust and easy to use for industry practitioners.

Recognizing this, the International Electrotechnical Commission (IEC) has put forward the IEC 61499 standard to facilitate a component-oriented approach for the development of distributed industrial-process measurement and control systems [5]. This standard proposes the use of function blocks as the basic construct for developing reusable modules for control software in a target-independent manner. Each function block is a functional unit of software that encapsulates local data and algorithmic behaviour within a well-defined event-data interface. The operation within a function block is controlled by an event-driven state machine.

Complete systems may be described graphically by connecting existing function blocks with newly created ones using event and data lines to form a network. Networks of function blocks may further be grouped together and distributed on different devices for execution. Typical implementations of IEC 61499 specifications rely on some run-time environment to dispatch events between function blocks in a network, and to schedule the blocks for execution. For this purpose, the function blocks will need to be compiled into appropriate software objects that can be instantiated in the run-time environment.

Code is generated from function block descriptions using software synthesis tools.
Such automatic synthesis from high-level abstractions provides a pathway for more reliable implementations. The simplicity and clarity provided by function block descriptions not only reduce the development effort, but also help to ensure coherency between design requirements and the final implementation.

While software synthesis from graphical models is not new (see for example [6–10]), the IEC 61499 standard offers numerous practical advantages over other model-based methodologies for developing industrial control software. It provides:

1. **Different abstraction levels for working with designs.** Architectural modelling of a system can be done intuitively with block diagrams, while the internal behaviour of each block can be defined using state diagrams. Complex data computations may in turn be specified using textual algorithms. The ability to work in the most natural specification notation for different parts of the design is a key feature of IEC 61499 function blocks.

2. **Support for the reuse of legacy algorithms.** Legacy algorithms written in the PLC languages of IEC 61131 can be encapsulated in IEC 61499 function blocks to enable their reuse in new designs. This ability is of practical importance to industrial engineers who are already accustomed to the former PLC languages.

3. **System-level approach [11] for designing distributed control systems.** Current distributed control systems are often developed by designing the various parts of the system separately, and then putting them together manually using appropriate communication protocols. This makes the design of such systems difficult, as the global view of the system is lacking. The IEC 61499 overcomes this by providing a framework to describe concurrent and distributed systems within a single system-level description. This approach preserves the global view of the system throughout the entire development phase, thereby allowing tools to automate the generation of code for distributed systems.

Despite its many benefits, the IEC 61499 standard, in its current form, is not without shortcomings. Design frameworks for large-scale industrial control software must not only allow concurrency to be easily expressed, but should also have clearly defined semantics of execution. The IEC 61499 standard, however, lacks a formal model of concurrency, and has left the precise behaviour of function block compositions to be implementation-dependent. At present, all IEC 61499 implementations rely on some run-time environment (e.g., [12–14]) to schedule the execution of function blocks in a program. The different scheduling policies used by the various
run-time environments are widely known from literature [15–17] to result in different program behaviours (e.g., in the sequence of events received and emitted). For a standard intended to facilitate reuse and portability, this situation is far from ideal.

The dependence on a run-time environment presents at least one other drawback: implementations of IEC 61499 systems are bulkier and slower than that which could be derived without them. The run-time environment increases memory footprint, while the dynamic scheduling performed by it to emulate function block concurrency adds overheads to the execution time. This may be ameliorated if the execution semantics of function blocks had been formally defined. Formal semantics facilitate the efficient generation of code [2], as concurrency can be compiled, instead of emulated.

The final lacuna in the standard concerns the distribution of function blocks onto different devices. Since the execution semantics of IEC 61499 systems are not formally defined, the model of communication between distributed parts of the system is unclear. Arbitrarily choosing a particular communication model will simplify the process of automatically synthesizing distributed systems. However, the wide variety of industrial applications that may be designed with IEC 61499 would make the choice of a single communication model rather restrictive. The challenge is to allow high-level communication properties to be specified using generic interfaces, which can then be refined by the designer with an implementation of choice. The synthesis tool may then automate the generation of communication code according to the designer’s choice.

In subsequent sections, an overview of the IEC 61499 function blocks will be provided, along with illustrations of some of the deficiencies in the current standard. Following that, the three challenges involving the formal model, efficient code synthesis, and the use of abstract communication patterns for distributed IEC 61499 systems will be briefly described. These challenges are the central issues addressed in this thesis, and will be further elaborated in Chapters 3, 4, 5, and 6.

1.1 IEC 61499 preliminaries

The function block network in Figure 1.1 depicts a cruise control system. When in use, the system will attempt to maintain a constant vehicle speed so long as the accelerator and brake pedals of the vehicle are not depressed. The modelled cruise control system has five buttons, which produce discrete input events to the system whenever they are pressed. The set button is used to save the current speed and to activate the cruise mode once the vehicle has reached the desired speed. This speed
can then be increased or decreased by 5km/h by pressing the quickAccel or quickDecel buttons respectively. This speed will remain unchanged until the brakes (which are also an input) are applied. If this happens, the system goes to the standby mode and can be resumed with the memorized speed by pressing the resume button. At any time, the cruise control system can be deactivated by pressing the off button.

The three function blocks in the network operate concurrently based on the input events that they receive. They perform the following functionalities:

- the CruiseManager computes the state of the system and the desired cruising speed based on the current vehicle’s speed, the buttons pressed, and the input from the brake pedal;
- the SpeedGauge computes the actual speed based on the input from a rotary encoder; and,
- the Throttle regulates the throttle position by computing the difference between the desired cruising speed and the actual speed, taking into account the cruise control’s state and the depression of the accelerator.

The IEC 61499 standard provides three different kinds of function blocks that can be used to describe a program. These are the basic function block, the composite function block, and the service interface function block respectively. Regardless of the type, every function block has a well-defined input/output interface that consists of event and data ports, as illustrated in Figure 1.2. Event ports are drawn on the upper half of the block, while data ports are drawn on the lower half. Event-data associations may be specified at the interface to update internal data with new values from the interface whenever the associated event occurs.
The behaviour of a basic function block is defined using a Moore-type finite state machine (FSM), known as an execution control chart (ECC), similar to the one shown in Figure 1.3 for the Throttle function block. An ECC consists of execution control (EC) states, EC transitions, and EC actions. Each of these elements are highlighted in Figure 1.3 for illustration. By convention, the initial state of an ECC is indicated by a double rectangle.

Function blocks adopt an event-driven model of execution. The transition conditions in an ECC are evaluated whenever the function block receives an input event. The conditions are expressed using an input event and/or a Boolean guard condition. For states with more than one outgoing transition, the transition conditions will be evaluated according to the order in which they are declared in their XML
representation\(^1\) (see page 33 of [5]). Each state may be further associated with zero or more EC actions, which consist of an algorithm to be executed and/or an output event to be emitted. Actions associated with a given state will be executed once upon entry to the state.

Composite function blocks enable the encapsulation of a network of function blocks within another block. Unlike basic function blocks, the behaviour of composite function blocks depends on the composite behaviour of the encapsulated network, rather than on an ECC. Service interface function blocks, on the other hand, are implementation-dependent blocks that provide an interface between the application and the underlying execution platform. These have been omitted from Figure 1.1 for brevity. Networks consisting of these various types of blocks may be further grouped together within a functional unit of software known as a resource. Resources may be allocated to specific devices, and a device itself may have several resources allocated to it. In IEC 61499 parlance, a device is simply a programmable controller. A system is made up of various devices that implement a complete specification.

This hierarchical structure and intuitive graphical notation enable modular description of programs. Unfortunately though, the nature of event propagation and the scheduling order of function blocks in a network are not clearly defined in the standard. In fact, the standard seems to be ambivalent concerning this by stating that,

> “Standards, components and systems complying with this part of IEC 61499 may utilize alternative means for scheduling of execution” (page 21 of [5]).

This makes reasoning about the formal correctness of function block descriptions difficult. Some of these ambiguities are highlighted in the next section.

### 1.2 Ambiguities in function block execution

Many of the discrepancies between existing function block implementations arise due to the standard’s inadequate treatment of two fundamental aspects of concurrent models of computation. They are:

1. **The lack of any notion of time.** Function blocks do not have an explicit notion of time. Hence, the concept of simultaneous events and the lifetime of events within an ECC can be ambiguously interpreted.

\(^1\) Graphical notations in IEC 61499 have corresponding representations in XML form as specified in [18].
2. The lack of any notion of composition. While individual function blocks may be connected to form networks like in Figure 1.1, the standard does not define the composite behaviour of such a network. Nothing is said about the composite state when multiple ECCs are connected, or of the semantics for their communication. Hence, a variety of ad hoc approaches have been suggested in literature [19].

These have resulted in different interpretations and/or implementations, which are exemplified in the following two subsections.

1.2.1 Evaluation of transitions within function blocks

Figure 1.4 shows an ECC fragment, where $E_1$ is an input event, and $C_1$, $C_2$, and $C_3$ are Boolean guard conditions. Transitions in an ECC are only evaluated with the occurrence of an input event, and are done in the order in which they have been declared in their XML representation. Let us assume here that transitions in the figure are evaluated from left to right, and the current state is State1. Two ambiguities arise in this case:

1. First, consider the case where $C_1$ and $C_2$ are both true when $E_1$ occurs. Consequently, the transition to State2 will occur. Whether or not a subsequent transition to State4 will take place is not clear. This is because the lifetime of an event is undefined in the standard. In fact, some run-time implementations, like ISaGRAF [14], have left the decision concerning the lifetime of events entirely at the discretion of the designer [20]. In that case, it becomes the designer’s responsibility to remove an event once it has been used.

2. Next, consider the case where $C_2$ is true, while $C_1$ and $C_3$ are both false when $E_1$ occurs. Then, the transition to State3 will take place. Subsequently, however, two different interpretations are possible from the standard:
1.2. AMBIGUITIES IN FUNCTION BLOCK EXECUTION

(a) Transitions consisting of pure data conditions will only be evaluated once, upon the completion of the action in the state. Thus, the designer must ensure that “eventless” transitions have at least one guard condition that will be true upon the completion of the preceding state (e.g., C1 and/or C3 should be true at the end of State3). Otherwise, the ECC will freeze in the culpable state. This is the approach adopted by FBRT [12], a widely-accepted function block run-time environment.

(b) “Eventless” transitions will be evaluated whenever an input event is fed to the function block. This implies a dependence on input events for the evaluation of data conditions. In this case, should C1 and/or C3 eventually become true, the transition out of State3 will occur when the function block receives an input event.

Obviously, the first interpretation may potentially lead to a deadlock and should, therefore, be avoided.

Besides these, another source of ambiguity in ECCs concerns the treatment of simultaneous events. The standard is not clear as to whether it allows function blocks to react to multiple events simultaneously (see [19] for a discussion on this). Some run-time implementations, such as FBRT [12] and FORTE [13], strictly forbid this, while others, like ISaGRAF [14], permit this. This ambiguity again relates to the notion of time. It is not clear whether the standard views time as being continuous or discrete. If time is continuous, it is problematic to give precise definitions for simultaneity, and may be the main reason why some run-time implementations have chosen not to allow it at all.

The approach adopted in this thesis (see Chapter 3) will explicitly define time to elapse only in discrete instants, making formal arguments for the lifetime of events and for the simultaneity of events possible. On the other hand, the question of when to evaluate “eventless” transitions will be explicitly dealt with without recourse to any run-time environment.

1.2.2 Compositions of function blocks in a network

The standard also lacks a formal definition for the composite behaviour for a network of function blocks. This results in different behaviours for a given program, depending on the scheduling scheme chosen by a particular implementation.

Consider the network in Figure 1.5. Both FB2 and FB3 need to be invoked as a result of the execution of FB1. The decision of when to execute FB2 and FB3,
Figure 1.5: Ambiguities in execution behaviour of a function block network.

However, is not fully clear—FB2 can be executed ahead of FB3 or vice versa, or both can be executed concurrently [21]. This ambiguity is compounded further when there is an event feedback, like that between FB2 and FB1. Without a well-defined model for concurrency, the use of ad hoc policies to schedule such scenarios may inadvertently lead to:

- *race conditions*—FB1 may be triggered again by FB2 before it can complete its current execution; and

- *starvation*—FB3 may be left unattended, while FB1 and FB2 monopolise the computational resources.

These issues relating to the composition of function blocks in a network will be completely resolved in this thesis. Chapter 3 will introduce an intuitive, yet mathematically precise, manner to describe the semantics of concurrent compositions, which can be used to fully define the execution order for a network of function blocks.

### 1.3 Formal model for function block systems

The IEC 61499’s rich notations—consisting of block diagrams, graphical state machines, and conventional textual algorithms—enable system-level designs to be intuitively done. Automated verification and code synthesis are possible side effects of such system-level designs. However, in order for this to be realised, a formal definition of function block semantics is necessary so that everything that manipulates the
function block program, including the designer, may clearly envisage the meaning of the program.

At present, the standard provides only an intuitive description of the expected behaviour for function block execution. This gives rise to various ambiguities, which have been resolved by specific implementation choices adopted by the various runtime environments, as exemplified previously. This has resulted in subtle differences in behaviour [22], which not only defeats the purpose of a standard, but also complicates efforts towards the formal verification of function block programs. In order to address this, the intuitive description for function block execution needs to be formalised, instead of being left to the (usually opaque and arbitrary) policies of a run-time environment.

This thesis proposes the globally asynchronous locally synchronous (GALS) paradigm [23] as a formal model to describe distributed IEC 61499 systems. With the GALS model, a distributed IEC 61499 system can be viewed as a collection of synchronous islands (consisting of synchronous compositions of function blocks), which communicate with each other over an asynchronous network. Resources in IEC 61499 provide a suitable boundary for such synchronous islands. Since resources are the artefacts that get allocated onto different devices for execution, asynchrony arises naturally between the resources as they communicate with each other over the network.

Within a single (but possibly hierarchical) function block network, the synchronous approach [24] is used as the semantic basis for function block execution. This approach views function blocks within a network as running concurrently in lock-step with a logical clock. Communication between concurrent blocks occur instantaneously, as all blocks are fully synchronized with each other. The behaviour of the blocks are deterministic, as the synchronization removes the various inter-leavings between communicating components. This makes it easier to describe and analyse compositions of function blocks, while facilitating the automatic verification of each locally-clocked resource [25].

While the assumptions of the synchronous approach match well with a centralized architecture, it is a poor fit to many distributed architectures. Control systems targeted by IEC 61499, like those for process-control and manufacturing, are often distributed over a communication network. The network usually does not comply with the synchronous model due to variances in computation speed among the nodes, as well as communication latencies in the network. Enforcing a fully synchronous distributed solution, like that of time-triggered architectures [26], is notoriously costly and inefficient [27]. As such, asynchronous models are more natural for distributed
systems, though greater effort is often required to ensure that the overall system will behave as desired.

The proposed GALS model, thus, offers a good compromise between the synchronous and asynchronous approaches. Determinism is preserved locally in each resource, while the global system benefits from the higher parallelism and less constrained communication of asynchronous distributed execution. Purely synchronous solutions, like Esterel [28] and Lustre [29], would have made distributed execution difficult, while purely asynchronous ones, like Occam [30] and Erlang [31], would have forced all parallel compositions to be non-deterministic.

1.4 Software synthesis

A system-level modelling framework should not only be defined formally, but should also have a compilation procedure that produces efficient code. A sophisticated modelling framework that cannot be executed efficiently will not be too useful in practice. This is particularly so for modelling frameworks intended for industrial control software, as such software often runs on embedded controllers with limited processing power and memory capacity.

In resource-constrained embedded applications, it is still common to find low-level C code interspersed with carefully crafted assembly code. Such code continues to be written despite significant advancements in programming languages and compilers due to its compactness and efficiency. Hence, it is crucial that any high-level approach intended to substitute manual writing of control software is adequately supported by synthesis techniques that produce efficient code.

Current IEC 61499 implementations all rely on a run-time environment, such as FBRT [12], FORTE [13], or ISaGRAF [14], for execution. This is undesirable, and may not even be feasible for resource-constrained applications, as run-time environments impose significant speed and size penalties on embedded software. Instead of a run-time environment, the code synthesis technique described in this thesis relies on the synchronous approach [24] to schedule function blocks at compile-time, which results in faster execution. Furthermore, the benchmark programs evaluated in this thesis indicate that existing function block code generators produce noticeably bulkier object code compared to the proposed approach, even without considering the overheads incurred by the run-time environment (see Chapter 7).

As efficient code generation is a significant consideration for adopting high-level modelling frameworks, the relevance of such research cannot be understated.
1.5 Abstract communication patterns

The new design paradigm offered by IEC 61499 is a significant step forward from existing PLC programming techniques, which has hitherto relied on vendor-specific methods to enable communication. IEC 61499 abstracts communication mechanisms through the concept of communication function blocks—a special class of service interface function blocks—which hide the underlying implementation while exposing the communication interface to the application.

The use of communication function blocks instead of vendor-specific methods facilitates reuse and portability by clearly separating computation from communication. This separation allows independent refinement of each, and the freedom to mix and match function blocks for computation with various kinds of communication function blocks. Software synthesis techniques can then be applied to automate the generation of communication code in distributed systems. This would not have been easily accomplished for program specifications that are tightly-coupled to vendor-specific code.

The goal, however, is to provide a way for designers to specify abstract communication patterns between synchronous islands using high-level properties that do not yet imply any particular mode of communication. For example, there should be a way for the designer to characterize a particular communication as being lossy or lossless without already deciding on the implementation, such as the protocol or the buffer size to use. The ability to reason about communication properties at this abstract level provides flexibility to trade-off various metrics, like robustness and efficiency, in the implementation space. Each type of communication pattern can be represented by a separate type of communication function block, which can then be collected to form a library of communication blocks. Appropriate code can then be automatically generated for each communication block and woven together with the code for the synchronous islands to form a complete distributed system.

1.6 Research contributions

In this thesis, the subject of formal modelling and synthesis of safety-critical industrial control software will be investigated in the context of the IEC 61499 standard. The three challenges briefly described in the previous sections will be addressed through the following primary contributions of this thesis:

1. A novel execution model for function block networks based on the synchronous approach [24]. This thesis introduces synchronous semantics for function
blocks that will remove execution ambiguities. The synchronous semantics will give explicit meaning to the various interpretations surrounding some of the clauses in the standard. Consequently, implementation concerns, such as scheduling policies, can be ignored during the specification phase. The synchronous semantics is given as structural operational semantic rules [32], and all programs constructed with this semantics are guaranteed to be reactive and deterministic. Reactivity is the property that ensures a program never enters a deadlocking state, while determinism is the property that ensures a program will always behave in the same manner in a given state for a given input. These properties are essential for safety-critical software and will be formally defined in Chapter 3. The proposed semantics are also an extension to an earlier synchronous semantics [33].

2. A compiler [34] to translate function block networks to the Esterel [28] synchronous language. The feasibility of the synchronous approach for function blocks is demonstrated by a new compiler that maps function block descriptions to a subset of Esterel, a well-known synchronous language. Esterel has well-defined formal semantics with commercial tools for verification and code synthesis. Since current tools for formal verification cannot operate on IEC 61499 descriptions, this approach provides a means to quickly take advantage of existing tools from the synchronous domain to verify, for the first time, both control and data properties of function block programs. Verified Esterel programs can subsequently be compiled for execution, without the need for any run-time environment.

3. A tool-chain for generating efficient C code from function blocks. A new function block code generator that generates C code [35] following the proposed synchronous semantics has also been developed. The implementation of this compiler is markedly different from the one that generates Esterel [34], as this compiler must now ensure synchronous concurrency and communication in the resulting C code on its own. This task of ensuring synchrony was previously delegated to the Esterel compiler. The C code generated by this compiler is significantly more efficient than the Esterel code produced by the former in terms of execution speed and object code size. This tool-chain is further able to automatically generate a graphical simulation environment that will enable the user to observe the outputs produced by a function block for a given input sequence.
4. A model for distributed IEC 61499 systems based on the GALS paradigm. Various communication patterns can be used between distributed nodes in this GALS model with appropriate communication function blocks. The proposed tool-chain includes a library of communication templates, from which actual code for the different types of communication function blocks envisioned can be automatically generated. To the best of our knowledge, this is the only tool for IEC 61499 programs that can generate distributed code from function block descriptions without requiring any middleware or run-time environment for execution.

5. A case study and extensive benchmarks to evaluate the validity of the proposed execution model and developed tools. The developed tools and techniques have been applied to the design of a prototype airport baggage handling system [36]. The code produced by the tools described herein compares very favourably with other existing code generators. In further evaluations on a suite of benchmark programs, experiments continue to demonstrate the superiority of the code produced by these tools compared to existing ones, both in terms of execution speed and code size. This indicates the suitability and advantages of the proposed approach over existing code generators for developing control software with IEC 61499, particularly in resource-constrained environments.

1.7 Thesis organization

This thesis consists of seven remaining chapters. Chapter 2 positions this work in the context of other related work. Previous models used for formal verification of function blocks, as well as less formal ones proposed for execution, are both reviewed. Existing code generators for function blocks will also be reviewed to see how different implementations have gone about realising the various models for execution. Closely related work in the wider synchronous programming context will also be looked at, together with relevant techniques used in implementing distributed systems.

Chapter 3 introduces synchronous programming using the Esterel language. A synchronous model for function blocks is then presented to show how it is able to resolve the various semantic issues related to the execution of function blocks. The formal semantics for synchronous function blocks are then presented to demonstrate that all programs constructed with this semantics will always be reactive and deterministic.
In Chapter 4, the compilation of function blocks to Esterel is described. The principles of translation and the internal details of the compiler are also presented. The chapter will then go on to show how function block programs can be verified using the verification tools for Esterel. The compilation of function block programs to efficient C code is treated next in Chapter 5.

Following that, Chapter 6 describes the GALS model for distributed function block systems. Communication function blocks with different high-level properties are presented, together with an approach to automate their synthesis.

Then, in Chapter 7, a case study and experimental results comparing the tools described in this thesis with other existing tools are presented. This thesis finally concludes with some closing perspectives and avenues for future work in Chapter 8.
The previous chapter has introduced the IEC 61499 function blocks [5], as well as issues surrounding them that will be addressed in this thesis. In this chapter, some of the existing work related to the issues discussed will be reviewed.

A major source of the problems described in the previous chapter stems from the lack of well-defined semantics for function block execution in the current IEC 61499 standard. Numerous efforts have, thus, been undertaken in the past to address this shortcoming. Some of these efforts have proposed formal models in the context of formal verification of function block descriptions. Others, on the other hand, have put forward less formal models of execution that are intended to guide code generation and run-time execution of function block programs. Existing proposals from both these lines of research will be surveyed here.

Related work in the synchronous programming domain for ensuring compositionality will also be reviewed. The globally asynchronous locally synchronous (GALS) paradigm [23] will be employed in this thesis to implement distributed IEC 61499 systems. Therefore, a survey of some relevant work regarding distributed systems within the GALS context will also be given here. This chapter will finally conclude with some perspectives on the state-of-the-art that have motivated the research undertaken for this thesis.

The next section begins this survey by reviewing the various formalisms that have been put forward for function block verification.

### 2.1 Formalisms for function blocks

Earlier attempts at a formal model of IEC 61499 function blocks have advocated the use of *Net Condition/Event Systems (NCES)* [37, 38]. NCES is a formalism based on Petri Nets [39] that was originally intended for modelling discrete event
systems. Similar to Petri Nets, NCES consists of \textit{places} and \textit{transitions} that are used to represent conditions and events, respectively. With NCES, each input event of a function block is modelled as an input event state machine and a storage module. At the output side, an additional module is used to model each output event. Every element in the ECC is also mapped to a separate NCES module, with one for each state, action, and transition. Another additional module is associated to each ECC to indicate whether a transition has cleared or not. These modules each consist of a number of places and transitions. While there exists tools capable of verifying NCES \cite{40}, the large number of places and transitions required to represent even a small function block program presents a significant challenge to the scalability of this approach.

An alternative to this approach models function blocks as a network of \textit{timed automata} \cite{41}. This technique builds an automaton for each input event of a function block. An additional automaton is further associated to each ECC to model a reaction of an ECC. The timed automata obtained from a given function block network can subsequently be verified using the Uppaal \cite{42} model-checking tool. While this approach is simpler than the one in \cite{37}, it still requires the use of many automata to verify a function block program. Other state-transition formalisms, like \textit{interacting automata} \cite{17}, have also been similarly applied to model the various operations of function block execution. Automata obtained through this technique can be analyzed with Supremica \cite{43}, which is a tool for supervisor verification and synthesis.

All the aforementioned approaches, however, are unable to adequately model data computations and require them to be abstracted. The NCES model of \cite{37} can only handle Boolean data conditions in the ECC. Meanwhile, the automata model in \cite{17} is restricted to state transitions that do not involve data conditions, while that in \cite{41} is even more limited by requiring that all data variables are fully abstracted. As such, formal verification based on these models lead invariably to overly conservative results. The verification itself is restricted to control properties, as data properties cannot be verified.

Others, like Dubinin et al. \cite{44}, have proposed a new semantics dedicated to function blocks, independent of other formalisms. The model adopted there allows function block networks to be subsequently verified as closed-loop systems using Prolog \cite{45}. The semantics proposed in \cite{44}, however, assumes a sequential execution of blocks in a network. This assumption is awkward for a standard intended for distributed systems. However, an even greater drawback of a sequential model is that it provides no means for compositional properties of function blocks to be
studied. Without a notion of a composite state for a combination of ECCs in a network, it becomes difficult to express compositional behaviour mathematically. This complicates efforts towards the automated verification of function block programs.

While all these methods mentioned so far can be used to verify certain properties over a given model, it is significant to note that none of them are able to automatically transfer the verified model to actual executable code. A previous work [46] did, however, attempt to map function blocks to Signal [47], a synchronous programming language. The use of synchronous languages to formally model function blocks is advantageous, as it enables powerful compilers to be exploited for code generation of function block programs. In the work in [46], function block elements were mapped to the various constructs in Signal. Function block verification could then also rely on the existing verification tools for Signal. That work, however, seemed to have focus only on the control aspects of function block verification. Moreover, it did not present any automated approach to translate function block programs into Signal.

There have also been previous attempts to introduce the synchronous approach in the industrial control systems domain for the programming languages of the earlier IEC 61131 standard [48]. The work in [49] proposed a mapping for SFCs to Esterel [28]. While some simple translation rules were sketched out in [49], that work did not seem to have any automated means for converting an arbitrary SFC to Esterel. The faithfulness of the Esterel translation to the original SFC description was also not thoroughly treated.

### 2.2 Run-time execution models for function blocks

All existing IEC 61499 solutions currently rely on some run-time environment to dispatch events among function blocks in a network. These run-time environments, however, are each implemented using different execution models. There are currently three main execution models. They differ primarily in the way function blocks are scheduled, and in the manner by which events/data are propagated through a network. Due to this, the choice of run-time environments greatly influences the way code is being generated from function block descriptions.

The first function block run-time environment to appear is the Function Block Run Time (fbrt) [12]. Its companion code generator, called Function Block Development Kit (FBDK), produces Java code from function block descriptions that can be
executed within FBRT. FBRT adopts an event-triggered model for invoking function blocks. Event emissions are implemented as direct method calls to the destination function block(s). This results in a *depth-first event propagation scheme* through the network. While simple, this approach requires a very deep stack to implement long event chains in a network. Moreover, the direct invocation of methods holds up all event-emitting function blocks in the chain until each nested method call returns. If an event cycle appears between a set of function blocks, the first function block in the cycle to be triggered again will not be able to respond, since the initial method call would not have returned yet. This results in unexpected behaviours, as well as unpredictable temporal characteristics [15]. These limitations, coupled with the dependency on a Java Virtual Machine, makes this approach poorly-suited for resource-constrained microcontrollers.

Another event-triggered model that has been proposed makes use of an event dispatcher to pass events between function blocks [50]. A first-in-first-out (FIFO) queue is used to store all emitted events, which are then delivered to the destination blocks by the event dispatcher. This results in a *breadth-first event propagation scheme* through the network. Unlike the previous approach, this technique decouples the execution of the event-emitting function block from the receiving block, thereby making the blocking period of the former independent of the length of the event chain. Implementations adopting this approach are exemplified by the C++ code produced by the 4DIAC-IDE for its run-time, called FORTE [13]. While the compilation to C++ removes the need for a virtual machine, the multithreaded implementation of FORTE still results in slow and bulky code (see Section 7.2 for a quantitative comparison).

The Fuber run-time environment [17, 51] also implements a FIFO input event queue, together with an additional function block instance queue. However, unlike [50], each function block instance maintains its own event queue. Whenever an event is queued in a particular function block, that block is added to the instance queue. A scheduler is then responsible to take the queued instances in FIFO order and to execute them. This, too, has the effect of a breadth-first propagation of events through the network.

A third alternative adopts a cyclic-scan model [52]. This technique maps each function block to a separate program within a traditional PLC scan cycle, and orders them in some sequence. In each scan cycle, the input events of each block are checked, and the corresponding code is executed only when an associated input event has occurred. Output events are issued immediately, and may be tested by other blocks in the same scan cycle if they appear after the current one in the
execution sequence; otherwise, the output events would only be read in the next cycle. Obviously, in this approach, the behaviour of an application directly depends on the execution order within a cycle. Nevertheless, ISaGRAF [14], a commercial IEC 61499 development tool, has adopted a similar cyclic-scan scheme for its run-time environment and the corresponding code it generates from function block descriptions [20].

These various run-time environments will obviously result in different behaviours for a given application. Such discrepancies are largely due to the lack of rigorous specifications in the standard for describing the behaviour of a network of function blocks. Clearly, this current state of affairs is undesirable and needs to be urgently addressed, if IEC 61499 is to enjoy wide industry adoption.

2.3 Techniques for dealing with compositions

In the IEC 61499 research community, the issues concerning function block behaviour in a network, as mentioned in Section 1.2.2, have oftentimes been regarded as a scheduling problem. Many different scheduling techniques [15, 19–21, 50, 53] have been proposed in an attempt to find an acceptable solution on how to execute a network. These, however, have been done in a rather ad hoc manner, resulting in different program behaviours even with just subtle variations, like the size of the event queue used, in the implementation.

In the synchronous programming domain [24], this issue involving the arbitrary connection of function blocks in a network have also been encountered before. There, instead of viewing this as a scheduling problem, the issue has been addressed through a notion of correct compositions [54] of concurrent blocks. In the synchronous paradigm, time elapses in discrete instants, with all computations within an instant conceptually taking place instantaneously. The computation that takes place within an instant is commonly referred to as a reaction. Due to this notion of instantaneous reactions, arbitrary connections of blocks in a block diagram may result in a deadlock condition, if inputs and outputs are connected together in an instantaneous feedback loop. This makes the distinction between inputs and outputs subtle. The natural intuition of outputs being produced as a result of inputs is no longer clear, since the causal relation between them is now blurred. Such programs are said to be non-causal [54].

This issue of causality in concurrent compositions has been addressed in a variety of ways in different programming languages. These are briefly outlined here.
(courtesy of Benveniste et al. [24]).

1. **Acyclic**: A programming language can simply insist that the compositions in block diagrams contain no instantaneous loops. This is usually adequate for many data-flow models, and it guarantees that the composition will behave coherently. This approach has been adopted by the graphical synchronous language SCADE [8], and its textual counterpart Lustre [29].

2. **Microsteps**: A programming language can also ensure that compositions are coherent by defining the reaction that needs to be done as a sequence of elementary microsteps. The elementary components that make up the concurrent composition can then be thought of as implementing such a sequence of microsteps. This gives an operational semantics for execution, but exposes the various internal interleavings of the composition. Hardware description languages, like VHDL [55] and Verilog [56], as well as the reactive programming language Statecharts [6], have adopted this approach.

3. **Unique fixpoint**: Each reaction in a composition may also be viewed as the unique solution of a fixed-point equation. The composition is guaranteed to behave coherently by insisting that each reaction is a deterministic function of the form:

   \[
   \{\text{state, input}\} \rightarrow \{\text{next state, output}\}
   \]

   Compiling a program with these semantics is difficult, however, because it is necessary to ensure that the relations implied by the program always have a unique solution of the form above. Despite these difficulties, the synchronous programming language Esterel [28] has chosen this approach.

One other approach that has also been proposed to resolve causality issues is to simply make it impossible to write non-causal programs through the semantics of the language. The synchronous language SL [33] achieves this by allowing instantaneous reactions to signal presence, but delays all reactions to signal absence to the next instant. Signals can only be determined to be present if they are emitted, while their absence can only be decided at the end of the current instant. Thus, unlike Esterel programs that are compiled according to their communication dependencies, SL programs resolve communication dependencies at run-time. The execution of SL threads containing a test of an unresolved signal is suspended, until the signal is emitted, or until all other parallel threads pause, terminate, or become suspended. In this case, all unresolved signals will be recognised as absent.
2.4 Distribution of function block programs

Distribution of function block programs is a very open area in the IEC 61499 standard. While specific design artefacts, like resources, can be deployed onto different devices to obtain a distributed system, no precise distribution model is prescribed. Few works have also been published on specific models for distributed IEC 61499 systems. Existing literature, like that in [57], has mostly reported on the mechanics of implementation. That work was concerned with the allocation of function blocks on platform-specific execution contexts, such as threads or processes. Models of communication and concurrency for function blocks executing on multiple devices were not studied at all.

Beyond the domain of IEC 61499, numerous works have been done regarding distributed systems that adopt the GALS paradigm [23]. In [58], codesign finite state machines (CFSM) are used to describe and synthesize embedded control software. CFSMs extend “pure FSMs” with variables, and interact with each other via an asynchronous communication mechanism. The model of computation for a network of CFSMs is GALS, with the communication between each CFSM taking place through a single-place buffer for every event/data exchanged. Hence, it is possible for events/data in CFSMs to get overwritten and lost.

The work of Sangiovanni-Vincentelli et al. [59] proposed an abstract representation for communication among processes using a variant of CFSMs, called abstract CFSMs (ACFSM). ACFSMs decouple the behaviour of each CFSM from the communication with other CFSMs. This enables different communication mechanisms to be specified in lieu of the single-place buffers of the original CFSM model. These abstract communication specifications can then be refined to yield concrete implementations. However, that work did not describe how such an attempt can be automated for the variety of physical channels and protocols used in practical distributed systems. On the other hand, a goal of this thesis is to leverage an industrial standard, like the IEC 61499, to automatically synthesize various communication patterns for distributed control system based on the GALS model.

Others, like Caspi et al. [60], have proposed an approach to automatically distribute a centralized synchronous program onto an asynchronous network of processors, and to ensure its subsequent resynchronization. In that work, a synchronous program is first compiled into a single automaton before being distributed. The distribution specification is given according to the physical location of sensors and actuators. The control structure of the automaton is duplicated on every processor that forms part of the distribution. This, however, cannot be directly applied to
distributed IEC 61499 systems, as compiling them into a single automaton is not a viable option. The distribution of the system is, furthermore, already specified by the allocation of resources to various devices.

2.5 Remarks

All the works surveyed in the previous sections give a snapshot of the current state-of-the-art in IEC 61499 implementations. It is worth noting that none of these existing proposals fully address the issues of formal modelling and synthesis of distributed IEC 61499 systems, as mentioned in the previous chapter.

A prominent feature that is lacking is a formal model for function blocks through which formal verification and correct-by-construction code—free from compositionality issues—can be obtained. Hence, this thesis will extend the state-of-the-art by presenting a formal framework for ensuring causal code, as well as prototype a tool to automatically generate verifiable models and executable code from function blocks. Instead of relying on run-time environments, like those described in Section 2.2, the generated code will be able to run directly on its own in a fully predictable manner. This will not only avoid issues of run-time environment compatibility, but will also lend itself to the efficient implementation of function blocks.

Automatic synthesis of various communication patterns in distributed IEC 61499 systems will also be dealt with in the context of the proposed formal model. This is a significant contribution to the state-of-the-art, as minimal work has been done so far in this area for IEC 61499.
This chapter will introduce a formal model for a centralized IEC 61499 function block program based on the synchronous programming paradigm [24]. In order to do so, an overview of the synchronous programming paradigm will first be illustrated using Esterel [28], a well-known synchronous language for programming safety-critical reactive systems. Esterel has formal semantics that provide a precise and elegant approach to deal with the issues of time and compositionality, like those described in Section 1.2.

Following that, this chapter will present a set of postulates that will provide a basis for interpreting IEC 61499 function blocks within a synchronous framework. Using these postulates, this chapter will introduce synchronous function blocks, which are function blocks that adopt synchronous semantics for execution. This will be illustrated through a mapping of the cruise control example of Figure 1.1 to a subset of Esterel. This enables the ambiguities in the standard, as exemplified in Section 1.2, to be addressed using the elegant notions for time and compositionality in the synchronous programming paradigm.

The proposed synchronous semantics guarantee causality under any arbitrary composition of function blocks [34]. This is unlike Esterel, which requires costly analysis at compile-time to ensure the same [61]. The compositional semantics proposed for function blocks are formalised using structural operational semantic rules [32]. Section 3.3 will demonstrate that all programs constructed with this semantics will always be reactive and deterministic [34]. This chapter will finally conclude with a discussion on the significance of this semantics for function block implementations.
<table>
<thead>
<tr>
<th>Tick</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>O (true)</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>O (false), Q</td>
</tr>
<tr>
<td>2</td>
<td>I</td>
<td>O (true), Q (program exits)</td>
</tr>
</tbody>
</table>

Table 3.1: Example of an input/output trace for the program in Figure 3.1.

## 3.1 The synchronous programming paradigm

The synchronous approach [24] is a programming paradigm that is specifically tailored for the development of reactive systems. Reactive systems, originally introduced in [62], are used to refer to systems that need to react continuously to their environment, at a speed determined by the stimuli coming from the environment. The synchronous programming paradigm facilitates the specification of such systems by assuming an idealized program that produces its outputs synchronously with its inputs by executing infinitely fast. Time is treated as a sequence of discrete instants with nothing happening between the completion of the current instant and the start of the next. This is the central tenet of synchronous languages, and is known as the synchrony hypothesis [63]. This idea is prevalent in various fields of engineering, and has been used extensively by control engineers in discrete-time dynamic systems, as well as hardware engineers in digital logic design [24].

The idealized notion of instantaneous computations can be realised so long as the minimum inter-arrival time of input events is greater than the maximum time required to complete the computation and to produce outputs on the implementation target. Unlike general-purpose programs, computing the worst-case execution time for synchronous programs is much simpler, and has been demonstrated to be feasible even for large programs [64, 65], making this approach suitable for real-time systems.

The synchrony hypothesis has been adopted by Esterel [28], where every program reaction occurs with respect to a logical instant of time, known as a tick. A new reaction is triggered at the start of each tick by taking a snapshot of the input signals, performing some computation, and generating the output signals before the start of the next. This can be thought of as an abstraction of a PLC’s scan cycle. Key features of this language are highlighted through the example in Figure 3.1, while an output trace for a particular input sequence to that example is depicted Table 3.1.

The basic programming unit in Esterel is a module. Each module consists of an interface declaration (line 2), followed by a body of executable statements (lines 3–13). Signals in Esterel may consist of a status and/or a value component. Signals
Figure 3.1: Example of an Esterel program.

with only a status component are known as pure signals (e.g., signal \( I \)), while those with a value component are known as valued signals (e.g., signal \( O \), which has status and a Boolean value component).

The ‘;’ and ‘||’ operators denote sequencing and synchronous concurrency of statements, respectively. For instance, the ‘;’ operator on line 11 indicates that the pause statement is to be executed in sequence after the emit statement has been executed. Meanwhile, the ‘||’ operator on line 8 indicates that both the preceding and proceeding statements, consisting of a non-terminating loop each, are to be executed in parallel.

In the first loop, the emit statement (line 5) is used to perform two simultaneous functions: it sets the value of \( O \) as its toggled value in the previous instant, and makes the status of \( O \) present. The pre operator is used to obtain information about a signal in the previous instant. It can be used to obtain either the value (line 5), or the status (line 10) of a given signal in the last instant. When emitted, the signal’s status is made present for the current instant, and becomes absent again in the next. However, the value of the signal persists. Signal emissions are synchronously broadcast and may be tested concurrently (see line 10).

The if statement (line 6) performs an instantaneous presence test for the input \( I \). If \( I \) is absent, execution on this branch will pause for the current instant. The pause statement marks the end of a tick, and serves as a synchronization point for parallel threads in each tick. Otherwise, the trap construct (lines 3–13) will be terminated by the exit \( T \) statement, causing both branches of the parallel statement to terminate in the same instant. Such lock-step progression of concurrent threads
at each tick is known as synchronous parallel execution.

Meanwhile, in the second loop, the await statement in line 10 pauses execution until its delay predicate becomes true. The predicate can be specified by an arbitrary signal expression, like \( \text{pre}(\varnothing) \) in this example. When used with the immediate modifier, the await statement terminates instantaneously if the signal expression is true in the starting instant. Due to the \( \text{pre} \) operator, the emission of \( \varnothing \) (line 11) will always be lagging that of \( \varnothing \) by one tick.

### 3.2 Synchronous model for function blocks

The idea for synchronous function blocks has been inspired by Esterel’s synchronous model. In proposing a synchronous model for function blocks, it is desirable that the proposed semantics do not contradict what has been specified in the standard. The requirements of the standard, however, are not always well-defined, and may be scattered at various places in the text, or even left completely to the discretion of the implementer. Therefore, a set of postulates are required to guide the mapping of function blocks to the proposed synchronous model. These postulates, together with their conformance to relevant portions of the standard, are listed below:

1. **An event has a lifetime of only a single tick, regardless of whether or not the event was actually used in the evaluation of any transition within that tick.**
   
The lifetime of events are not specified in the standard, as mentioned earlier in Section 1.2. Instead, this postulate treats events like transient signals, which are commonly used as transition conditions in automata (see [54] for example).

2. **If more than one transition condition is true, they will be evaluated in the order in which they are declared.**
   
   This is explicitly asserted in subclause 5.2.2.2 of the IEC 61499 standard.

3. **The execution in EC states conceptually occurs instantaneously, with the EC actions executed in the order in which they have been declared.**

   Subclause 5.2.2.2 requires that the sequence of operations from the invocation of a function block to the evaluation of transitions and the execution of actions, be done as an atomic operation. This rule is analogous to having each EC state implemented as a synchronous state in Esterel, whereby algorithmic computations and event emissions are conceptually instantaneous within that state.
### 3.2. SYNCHRONOUS MODEL FOR FUNCTION BLOCKS

<table>
<thead>
<tr>
<th>Function block element</th>
<th>Esterel feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function blocks</td>
<td>Modules</td>
</tr>
<tr>
<td>Events</td>
<td>Pure interface signals</td>
</tr>
<tr>
<td>Data</td>
<td>Value-only interface signals</td>
</tr>
<tr>
<td>Event-data binding</td>
<td>Association of pure and value-only signals</td>
</tr>
<tr>
<td>Internal variables</td>
<td>Value-only local signals</td>
</tr>
<tr>
<td>EC states</td>
<td>Demarcated by pause statements</td>
</tr>
<tr>
<td>Algorithms</td>
<td>Instantaneous modules</td>
</tr>
<tr>
<td>Transition conditions</td>
<td>await statements</td>
</tr>
<tr>
<td>Function block network</td>
<td>Parallel composition of modules</td>
</tr>
</tbody>
</table>

Table 3.2: Function block to Esterel translation map.

Table 3.2 shows the basic translation map from function blocks to Esterel. The actual translation will be detailed in Chapter 4.

Intuitively, each function block in a network will be mapped to an equivalent Esterel module. Within a network, the various modules will be connected in parallel to achieve the same event and data connections as in the original function block network. Due to synchronous concurrency, the function blocks in a network will be executing in lock-step with one another. In each tick, input events will be sampled and computations will be performed as an atomic operation. For basic function blocks, the atomic operation within a tick consists of the evaluation of EC transitions in the current state, and the corresponding computation of the action(s) in the destination state, should a transition be taken.

Let us now revisit the semantic ambiguities in the IEC 61499 standard that were first presented in Section 1.2. Figure 3.2a shows the ECC of Figure 1.4, which was said to have ambiguous transitions due to the lack of a notion of time in the standard. Owing to the postulates, each EC state has been mapped to an equivalent synchronous state in the Esterel code depicted in Figure 3.2b. The life-span of an event is also now strictly defined to persist for the duration of the tick in which it occurs. The Esterel code has no transition ambiguities like those discussed in Section 1.2.1 and, hence, the ECC can now be unambiguously executed without recourse to any particular run-time environment.

Also, since all inputs are read at the start of each tick, and the tick itself is conceptually instantaneous, simultaneous events are possible in this model, as in other cyclic-scan models [20, 52]. This is in contrast to the event-triggered model which assumes that events can only propagate through the network one at a time, with the execution of function blocks ordered according to this single event-induced flow [17, 19, 50]. This assumption is both unnecessary and inadequate for modelling
concurrent systems.

In modelling concurrency, it is often necessary to express the occurrence of simultaneous events in different parts of a system. If simultaneous events are admitted, the approach that relies on the total ordering of events to schedule function blocks will immediately fail. However, even if events are assumed to only occur one at a time, the execution of blocks that emit multiple events in a given EC state will still be problematic if there exists event connection feedbacks in the network. In such cases, the event-triggered approach will either result in the loss of events, or the need for unbounded queues to store events. Both of these are undesirable in safety-critical systems, which are typical of many industrial control applications.

Nevertheless, even with a synchronous model, compositions of function blocks involving event connection feedbacks may still cause problems. Let us consider again the event loop connecting FB1 and FB2 in the function block network of Figure 1.5. This has been reproduced in Figure 3.3 for convenience. The ECC fragments of FB1 and FB2 can be written as,

```plaintext
await immediate EI1; emit EO1; emit EO2; pause;
```

and,

```plaintext
// State1
pause;
await
   case immediate ?C1 do
      // State2
      pause;
      await immediate E1 and ?C2;
      // State4
      pause;
      case immediate E1 and ?C2 do
         // State3
         pause;
         await
            case immediate ?C1 do
               // State5
               pause;
               case immediate ?C3 do
                  // State6
                  pause;
               end await
      end await
end await
```

Figure 3.2: State transitions. (a) ECC that gives rise to ambiguous state transitions. (b) Esterel implementation of the ECC in (a) that resolves ambiguities in the state transitions.
Figure 3.3: Ambiguities in execution behaviour of a function block network.

```plaintext
await immediate EI3; emit EO3; pause;
```

respectively. Since EO1 and EI3 are connected together, they form a common event and can be bound together by a shared signal, say S1, in Esterel. Similarly, EO3 and EI1 are connected and, hence, can be bound together by another shared signal, say S2. Thus, the equivalent Esterel code becomes,

```plaintext
await immediate S2; emit S1; emit EO2; pause;

and,

```plaintext
await immediate S1; emit S2; pause;
```n

respectively. Composing FB1 and FB2 in synchronous parallel will yield the following Esterel code:

```plaintext
await immediate S2; emit S1; emit EO2; pause;
||
await immediate S1; emit S2; pause;
```

Such compositions will deadlock in Esterel, as the synchrony hypothesis will result in an instantaneous cycle between the concurrent threads due to the dependency on S2 in the first thread, and a reverse dependency on S1 in the second thread. Compositions of this nature are considered to be non-causal [61], as signals cannot be propagated from the producer to the consumer in a causative manner without returning again to the original producer.

So, while Esterel allows for very expressive forms of communication between concurrent components through instantaneous signal reactions, complete adherence
to the synchrony hypothesis makes the compilation of Esterel complex. Every Esterel program would first need to be checked that it is causal before code can be generated for it. This checking is known as causality analysis, and involves ensuring that constraints arising from control and data dependencies will never lead to deadlocks. This possibility of writing non-causal (deadlocking) programs makes composition difficult. In fact, Huizing et al. [66] have shown that no semantics can be responsive (obey the synchrony hypothesis), causal and modular (notions related to compositionality) at the same time.

Thus, one way to achieve seamless composition of function blocks within the synchronous paradigm without raising causality problems is to postpone all event/data communication between function blocks to the next tick. Whenever there is a producer-consumer relationship between two or more function blocks, this technique ensures that the producer will emit the event/data in the current instant, while the consumer(s) will only read what is emitted in the next tick of the program. This simple “pipelining” of the producer-consumer blocks guarantees that their parallel composition will always be acyclic. This, in fact, weakens the synchrony requirements for communication between different blocks, and is equivalent to reacting only to the pre of signals in Esterel.

With this constraint, the Esterel code for the previous composition will be the following:

```plaintext
await immediate pre(S2); emit S1; emit EO2; pause;
||
await immediate pre(S1); emit S2; pause;
```

The composition above is now causal, and can be statically scheduled for execution. Due to this weakening of the synchrony hypothesis, it is possible to derive a set of primitive statements for describing synchronous function blocks, which will make it impossible to write non-reactive or non-deterministic programs. This set of primitive statements, follows the idea of kernel statements in [61]. Kernel statements were used to define the core statements of the Esterel language, from which, all other statements could be derived. The formal semantics for such a set of kernel statements for synchronous function blocks are presented in Section 3.3.

3.2.1 The cruise control example

Let us now return to the cruise control example in Figure 1.1. The equivalent Esterel code for that function block network and the ECC in Figure 1.3 are shown
module CruiseControl:
  input cclock, set, off, resume, quickAccel, quickDecel;
  // ... other inputs omitted ...
  output speed, speedSet, throttleChg, regulOff;
  // ... other outputs omitted ...
  signal s0 : value signed<[16]>, s1, s2, s3, s4, s5,
         s6 : value signed<[16]>, s7, s8 in
    run SpeedGauge [s0/speedVal];
    run CruiseManager [s1/regulResume, s2/regulSet,
        s3/regulOff, s4/regulStdby, s5/speedSet,
        s6/cruiseSpeed, s0/speed];
    run Throttle [s7/cruiseOff, s8/cruiseOn, s0/speed,
        s6/cruiseSpeed, s5/speedSet];
  loop
    emit regulResume<=s1; emit ?cruiseSpeed<=?s6;
    emit speedSet<=s5; emit regulSet<=s2;
    emit regulOff<=s3; emit s7<=s3 or s4;
    emit ?speedVal<=?s0; emit regulStdby<=s4;
    emit s8<=s1 or s2; pause;
  end loop;
end module;

In Figures 3.4 and 3.5, respectively. This code has been automatically generated using the technique that is presented in Chapter 4.

In Figure 3.4, the run statements (lines 8, 10, and 14) are used to instantiate and execute the code in other modules. These modules correspond to the three function blocks of Figure 1.1. Due to the ‘||’ operator, these modules will execute concurrently in lock-step with one another. The event and data connections between those function blocks are accomplished through appropriate signal binding between the modules, which are given as arguments in each run statement. Local signals in Esterel are declared (lines 6–7) to perform this binding. The final parallel thread on lines 17–22 consists of a non-terminating loop that manages the associations between the local signals and the interface signals of the modules. Its purpose will be described more fully in Chapter 4.

In Figure 3.5, the states in the ECC of Figure 1.3 have all been mapped to corresponding await statements. The await statements faithfully implement pos-
module Throttle:

input cruiseOff, cruiseOn, speedSet, cclock;
// ... other inputs omitted ...
output throttleChg;
output throttleVal : value signed<[8]> init 0;
var stateVar : unsigned<[16]> := 0 in
// ... some statements omitted ...
loop
  switch stateVar
    case 1 do
      pause;
      await
        case immediate pre(cruiseOff) do stateVar := 0;
        case immediate pre(accelPressed) do stateVar := 2;
        case immediate pre(cruiseOn) do stateVar := 1;
        case immediate pre(speedSet) do stateVar := 1;
        case immediate pre(cclock) do stateVar := 1;
      end await;
    case 2 do
      run Throttle_accel;
      emit throttleChg;
      pause;
      await
        case immediate pre(cruiseOff) do stateVar := 0;
        case immediate pre(accelReleased) do stateVar := 1;
        case immediate pre(cclock) do stateVar := 2;
      end await;
    default do
      run Throttle_normal; emit throttleChg;
      pause;
      await
        case immediate pre(cruiseOn) do stateVar := 1;
        case immediate pre(accelPressed) do stateVar := 0;
        case immediate pre(accelReleased) do stateVar := 0;
        case immediate pre(cclock) do stateVar := 0;
      end await;
    end switch;
  end loop;
// ... some statements omitted ...
end var;
end module

Figure 3.5: A fragment of the Esterel code automatically generated from the ECC in Figure 1.3.
Program starts with stateVar=0 and emission of throttleChg.

Nothing happens until cruiseOn occurs.

accelPressed occurs. Meanwhile, stateVar is set to 1 and await on line 31 ends.

stateVar is set to 2 and await on line 12 ends.

throttleChg is then emitted.

Figure 3.6: Timing diagram for an execution trace of the program in Figure 3.5. Each horizontal marker denotes a tick.

CruiseManager Input Output Throttle Input Output

regulOff

set regulSet

brakePressed regulStdby

set regulSet

cruiseOn

cruiseOff

throttleChg

Figure 3.7: Timing diagram showing the effects of “pipelining” on communication for a particular execution trace of the program in Figure 1.1. Each horizontal marker denotes a tick. The input events in the diagram have all been delayed by one tick. Events regulOff and throttleChg are emitted during initialization. Nothing happens then until the set button is pressed, resulting in the output of regulSet. This is fed to the cruiseOn input in the next tick. When event brakePressed occurs, the regulStdby output is produced, which is again delayed before arriving as the cruiseOff input of the Throttle block. This results in a final throttleChg event.
tulate 2, as the first of their predicates (specified as cases) that evaluates to true will be the one that is executed. The switch statement enclosing most of the code is simply a classical statement used for conditional selection. Depending on the input event that occurs, the await statements will assign a different value to the stateVar variable. This variable is used to encode different states of the original ECC, and the switch statement will select the appropriate state to go to based on its value. The timing diagram in Figure 3.6 illustrates the behaviour of these statements for a particular input sequence to the Throttle module, and shows how the transitions in the original ECC in Figure 1.3 is preserved.

Notice also that the delayed semantics proposed for synchronous function blocks have been faithfully adhered to in this mapping, as all predicates of each await statement contain the pre operator. The effect of this on the communication between the CruiseManager and the Throttle blocks of Figure 1.1 for a particular input sequence is illustrated in Figure 3.7.

3.3 Semantics of synchronous function blocks

So far, the synchronous model for function blocks has been described via a mapping to Esterel. This section, however, will present the formal semantics for synchronous function blocks based on Esterel’s weakened synchrony hypothesis, as was intuitively outlined earlier. Although Esterel’s semantics have already been formally defined in [61], a formal semantic definition for synchronous function blocks is still needful for two main reasons:

- Firstly, while function blocks can be translated to Esterel, Esterel’s semantics are not strictly the same as that of synchronous function blocks. Following Huizing’s [66] definition of semantic properties, synchronous function blocks are modular and causal, but not responsive, since they are unable to react instantaneously to inputs. On the other hand, Esterel is both modular and responsive, but potentially non-causal. Causality in Esterel has to be externally checked by a compiler, since it is not a semantic property of the language. In contrast, the proposed semantics for function blocks can be proven to be reactive and deterministic (and, hence, causal) under any composition. This guarantee simplifies the construction of a compiler, and facilitates the modular compilation of synchronous function blocks, as there is no longer a need to perform causality analysis over the whole program before generating code.

- Secondly, a formal semantics that is independent of Esterel is beneficial for
3.3. SEMANTICS OF SYNCHRONOUS FUNCTION BLOCKS

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nothing</td>
<td>do nothing and terminate instantaneously</td>
</tr>
<tr>
<td>pause</td>
<td>pause execution till the next instant</td>
</tr>
<tr>
<td>( t;u )</td>
<td>run ( t ), and then ( u ) in sequence</td>
</tr>
<tr>
<td>( t \parallel u )</td>
<td>run ( t ) and ( u ) concurrently</td>
</tr>
<tr>
<td>loop ( t ) end</td>
<td>repeat ( t ) forever</td>
</tr>
<tr>
<td>emit ( S )</td>
<td>emit signal ( S )</td>
</tr>
<tr>
<td>present ( S^p ) then ( t ) else ( u ) end</td>
<td>run ( t ) if the status of ( S ) in the previous instant (denoted by ( S^p )) is present; otherwise ( u )</td>
</tr>
<tr>
<td>trap ( T ) in ( t ) end</td>
<td>declare and catch exception ( T ) in ( t )</td>
</tr>
<tr>
<td>exit ( T )</td>
<td>raise exception for ( T )</td>
</tr>
<tr>
<td>( v := f(...) )</td>
<td>compute the value of ( f ) and assign it to ( v )</td>
</tr>
<tr>
<td>if ( c(...) ) then ( t ) else ( u ) end</td>
<td>run ( t ) if Boolean function ( c ) is true; otherwise ( u )</td>
</tr>
</tbody>
</table>

Table 3.3: Kernel statements for synchronous function blocks. \( t \) and \( u \) are used to represent any arbitrary statement.

Figure 3.8: Simple function block network appended with ECC fragments.

practical reasons. For one, it allows correctness arguments for function block code generation to be made irrespective of the target language. For instance, direct C code generation from function block descriptions is possible, without needing to first go through a translation to Esterel. Moreover, an independent semantics allows function blocks to be extended beyond Esterel’s purely synchronous model to, say, a GALS paradigm.

This section will, thus, introduce the set of kernel statements that forms the core of the proposed semantics (following the approach in [61]). The syntax and intuitive semantics of these statements are given in Table 3.3, while the formal semantics are presented in the following subsection.

The use of these kernel statements to describe a function block program will be illustrated using the simple example of Figure 3.8 (reproduced from Figure 1.5).
1. The ECC fragment of FB1 can be mapped to the following kernel statements:

```plaintext
... trap T in
  loop
    present pre(EI1) then
      emit EO1; emit EO2; pause; exit T
    else
      pause
    end
  end loop
end trap
...
```

The ‘;’ operator between the `emit` and `pause` statements implements the atomic execution of operations within an EC state. The `present` statement enclosing them ensures that the EC state will be executed only when triggered by the appropriate event. If the transition condition is not satisfied, the `else` branch and the surrounding `loop` will ensure that the program remains at that particular EC state. Finally, the enclosing `trap` construct, together with its `exit` statement, will break out of the `loop` whenever the transition condition is satisfied. This combination of kernel statements implements EC state transitions in a similar fashion as the `await` statement in Esterel, as was previously illustrated.

2. The ECC fragment of FB2 can be mapped in a similar way as follows:

```plaintext
... trap T in
  loop
    present pre(EI3) then
      emit EO3; pause; exit T
    else
      pause
    end
  end loop
end trap
...
```

3. The three function blocks can then be composed together using the ‘||’ operator in the following manner:

```plaintext
// Code fragment for FB1
...
trap T in
...
This is similar to what was already illustrated via Esterel in Section 3.2. As before, EO1 and EI3, as well as EO3 and EI1, may be suitably renamed to reflect their connections with one another. The remaining kernel statements are simply classical programming statements for assignment and conditional branching of data variables and will not be illustrated here.

### 3.3.1 Formal semantics

The formal semantics for the kernel statements are presented as program transitions using Structural Operational Semantic (SOS) rules \[32\] of the following form:

\[
t, D \xrightarrow{O_k} t', D'
\]

where,

- \( t \) is any arbitrary composition of kernel statements;
- \( D \) is the set of values of data variables before the transition;
• $O$ is the set of signals produced by the transition;

• $k$ is the completion code of the transition;

• $I^p$ is the set of signals registered in the previous instant;

• $t'$ is the residual of $t$ after the transition; and,

• $D'$ is the set of values of data variables after the transition.

The notation above describes a program’s transition from the state $t, D$ to $t', D'$, in response to the set of signals registered in the previous instant, $I^p$. For the initial instant, $I^p$ is defined to be an empty set. This transition will produce the set of signals $O$, and finish with the completion code of $k$. The term $t, D$ is said to have been rewritten into $t', D'$.

In any such transition, the resultant state and output may depend on the way some subterms are rewritten, or on the status of certain signals in $I^p$. Dependencies are expressed through deduction rules of the form

$$
\cdots
\frac{\quad}{t, D} \quad \frac{O, k}{I^p} \quad t', D'
$$

where the predicate above the bar ($\cdots$) must hold in order for the transition below it to happen. When no such dependency exists, the bar is omitted.

Completion codes are used to encode the type of transition that has been performed by a given statement, following the idea used in Esterel [61]. A statement may either perform a finished or unfinished transition. A finished transition implies that the sequence of transitions have terminated for the current instant, and that transitions of any residual term will only take place in the next instant. An unfinished transition, on the other hand, implies that the transitions of any residual term will continue to take place in the current instant. The nothing, pause, and exit T statements are the only ones that perform finished transitions. The nothing statement is encoded with 0, pause with 1, and exit T with an integer $\geq 2$. Meanwhile, statements that perform unfinished transitions will return the completion code $\perp$.

Completion codes provide a simple way to synchronize the execution of parallel threads. For instance, consider the parallel statement $t\parallel u$. If $t$ finishes with the completion code of $k$, and $u$ with $l$, the parallel statement itself will finish with the maximum between $k$ and $l$. If either branch does not finish, neither does the parallel
statement. A completion code synchronizer can, thus, be defined as:

\[
\text{syn}(k, l) = \begin{cases} 
  k & \text{if } k \geq l \\
  l & \text{if } k < l
\end{cases}
\]

This effectively ensures that parallel subterms will only synchronize when performing finished transitions and, hence, branches of a parallel statement will either terminate, pause, or exit some trap synchronously. The rewrite rules for all the kernel statements are presented next adopting the approach used in [33, 61].

**Base statements**

The *nothing* statement does nothing and terminates instantaneously.

\[
\text{nothing}, D \xrightarrow{\emptyset,0} \text{nothing}, D
\]

The *pause* statement pauses control over itself for the current instant and resumes from there in the next.

\[
\text{pause}, D \xrightarrow{\emptyset,1} \text{nothing}, D
\]

Exceptions are declared and lexically scoped by the *trap* T in *p end* statement. Within the body p, the exception is thrown by the *exit* T statement. It provokes immediate termination of the trap T statement, killing all other statements within its scope. The completion code generated by *exit* T is \(d + 2\), where \(d\) is the number of trap declarations that have to be traversed before reaching that of T. For semantic purposes, the depth of the exit statement is explicitly encoded as *exit* T\(_d\).

\[
\text{exit } T_d, D \xrightarrow{\emptyset,d+2} \text{nothing}, D
\]

All other statements executing within a given instant will eventually be rewritten into one of these base statements. In other words, a statement executing in an instant will either terminate, pause, or be preempted (exited from some trap). Transitions of base statements are *finished transitions*, while other transitions are *unfinished transitions*. A program’s *reaction* in an instant will be denoted by,

\[
t, D \xrightarrow{E,k} t', D'
\]
if there exists a transition sequence such that,
\[ t, D \xrightarrow{O_1, k_1} t_1, D_1 \xrightarrow{O_2, k_2} \cdots \xrightarrow{O_n, k_i} t_i, D_i \xrightarrow{\emptyset, k} t', D' \] (3.4)

where, \( k_m = \bot \), \( k \geq 0 \), and \( E = \bigcup_m O_m \ \forall m \in [1, i] \). The last transition in a reaction will never produce any signal since, by definition, it is always one of the base statements.

**Signal emission**

The emit \( S \) statement emits the signal \( S \) for the current instant.
\[ \text{emit } S, D \xrightarrow{\{S\}, \bot} \text{nothing}, D \] (3.5)

**Signal test**

If the signal \( S \) was present in the previous instant, the present statement simply gets rewritten to its then branch (rule 3.6). Otherwise, it is rewritten to its else branch (rule 3.7).

\[ \frac{S^p \in I^p}{\text{present } S^p \text{ then } t \text{ else } u \text{ end}, D \xrightarrow{\emptyset, \bot} t, D} \] (3.6)

\[ \frac{S^p \notin I^p}{\text{present } S^p \text{ then } t \text{ else } u \text{ end}, D \xrightarrow{\emptyset, \bot} u, D} \] (3.7)

**Data assignment**

Variables can be assigned with values from an arbitrary data expression. The values of variables are globally persistent. However, only the values registered in the previous instant can be read. Variable values for the current instant cannot be instantly accessed. In rule 3.8, \( D^p \) is the set of variable values that have been registered in the previous instant. The expression following \( D' \) is used to indicate that the value of \( v \), which is a variable in \( D \), has been assigned with the value of \( f(d^p_1, \ldots, d^p_i) \).

\[ \frac{d^p_1, \ldots, d^p_i \in D^p}{v := f(d^p_1, \ldots, d^p_i), D \xrightarrow{\emptyset, \bot} \text{nothing}, D' = D[v \leftarrow f(d^p_1, \ldots, d^p_i)]]} \] (3.8)

While no semantic restrictions are imposed on simultaneous assignments to a variable, write-write concurrency is prohibited at the syntactic level. For example, both
the statements below are disallowed:

\[
\begin{align*}
&x:=1 \ || \ x:=2 \\
&\text{present S then } x:=1 \ \text{end} \ || \ \text{present S else } x:=2 \ \text{end}
\end{align*}
\]

The second statement illustrates the syntactic aspect of this restriction. While the assignment to \( x \) will never happen simultaneously, such programs are, nevertheless, forbidden to simplify compilation.

**Data test**

Conditional branching can be performed based on Boolean data expressions.

\[
\begin{align*}
\text{if } c(d^p_1, \ldots, d^p_i) = \text{true} & \text{ then } t \ \text{else } u \ \text{end, } D \xrightarrow{\theta, \bot} t, D \\
\text{if } c(d^p_1, \ldots, d^p_i) = \text{false} & \text{ then } t \ \text{else } u \ \text{end, } D \xrightarrow{\theta, \bot} u, D
\end{align*}
\]

**Sequential statement**

Rule 3.11 expresses the fact that the sequence does not finish, if its left branch, \( t \), does not.

\[
\begin{align*}
&\text{if } c(d^p_1, \ldots, d^p_i) = \text{false} & \text{ then } t \ \text{else } u \ \text{end, } D \xrightarrow{\theta, \bot} u, D
\end{align*}
\]

If the left branch pauses, so does the sequence.

\[
\begin{align*}
&\text{if } c(d^p_1, \ldots, d^p_i) = \text{false} & \text{ then } t \ \text{else } u \ \text{end, } D \xrightarrow{\theta, \bot} u, D \xrightarrow{\theta, \bot} t', D
\end{align*}
\]

Moreover, if the left branch raises an exception (by exiting a trap), its right branch will never get executed.

\[
\begin{align*}
&\text{if } c(d^p_1, \ldots, d^p_i) = \text{false} & \text{ then } t \ \text{else } u \ \text{end, } D \xrightarrow{\theta, \bot} u, D
\end{align*}
\]
Otherwise, control will be immediately transferred to the right branch, \( u \), when \( \tau \) finishes.

\[
\begin{align*}
\tau, D & \xrightarrow{0,0}_{IP} \tau', D \\
\tau; u, D & \xrightarrow{0,1}_{IP} u, D
\end{align*}
\]  
(3.14)

**Parallel statement**

The execution of the left and right branches of the parallel statement may be interleaved provided either one of them performs an unfinished transition.

\[
\begin{align*}
\tau, D & \xrightarrow{0,1}_{IP} \tau', D' \\
\tau||u, D & \xrightarrow{0,1}_{IP} \tau'||u, D' \\
u, D & \xrightarrow{0,1}_{IP} u', D' \\
\tau||u, D & \xrightarrow{0,1}_{IP} \tau'||u', D'
\end{align*}
\]  
(3.15) (3.16)

Rule 3.17 uses the completion code synchronizer to specify the synchronized behaviour of the parallel statement. When both \( \tau \) and \( u \) perform finished transitions, the parallel statement synchronizes their execution using their completion codes.

\[
\begin{align*}
\tau, D & \xrightarrow{0,k}_{IP} \tau', D \quad k \geq 0 \\
\tau||u, D & \xrightarrow{0,l}_{IP} \tau'||u', D \quad l \geq 0
\end{align*}
\]  
(3.17)

As already mentioned in the description for data assignment, write-write concurrency on variables is disallowed, while read-write concurrency is semantically forbidden by rule 3.8. This means that \( \tau \) and \( u \) will never operate on the same variables in the same instant.

**Loop**

The loop simply rewrites into a sequence of its body with the loop itself.

\[
\text{loop } \tau \text{ end, } D \xrightarrow{0,1}_{IP} \tau; \text{loop } \tau \text{ end, } D
\]  
(3.18)

As a consequence, loop bodies that finish with the completion code of 0 will result in the undesired rewriting into an infinite sequence of unfinished transitions. Such instantaneous loops will be rejected by insisting that the body’s reaction never
terminates instantaneously, as required by rule 3.19.

\[
\begin{align*}
\tau, D \xrightarrow{E_k}_{I_p} & \tau', D' \quad k > 0 \\
\text{loop } \tau \text{ end, } D \xrightarrow{E_k}_{I_p} & \tau'; \text{loop } \tau \text{ end, } D'
\end{align*}
\]  
(3.19)

**Exception declaration**

Rule 3.20 expresses the fact that the trap statement does not terminate if its body performs an unfinished transition.

\[
\begin{align*}
\tau, D \xrightarrow{O_{\perp}} & \tau', D' \\
\text{trap } T \text{ in } \tau \text{ end, } D \xrightarrow{O_{\perp}} & \text{trap } T \text{ in } \tau' \text{ end, } D'
\end{align*}
\]  
(3.20)

If the trap body pauses, the whole trap statement pauses as well.

\[
\begin{align*}
\tau, D \xrightarrow{\emptyset_{1}} & \tau', D \\
\text{trap } T \text{ in } \tau \text{ end, } D \xrightarrow{\emptyset_{1}} & \text{trap } T \text{ in } \tau' \text{ end, } D
\end{align*}
\]  
(3.21)

If the trap body terminates, or exits the trap, then the trap itself terminates.

\[
\begin{align*}
\tau, D \xrightarrow{\emptyset_{k}} & \tau', D \quad k = 0 \text{ or } k \geq 2 \\
\text{trap } T \text{ in } \tau \text{ end, } D \xrightarrow{\emptyset_{1}} & \text{nothing, } D
\end{align*}
\]  
(3.22)

### 3.3.2 Definitions and proofs

The kernel statements presented in the preceding subsection are sufficient for describing all synchronous function block programs. The semantics of these statements can be used to formally reason about two desirable properties of safety-critical software, known as **reactivity** and **determinism** [61].

**Definition 1.** A program is reactive if, for any statement \( \tau \) and data set \( D \), there exists at least one reaction for the set of signals registered in the previous instant, \( I_p \).

**Theorem 1.** All synchronous function block programs are reactive, that is:

\[
\forall \tau \forall D, \ \exists \tau', D', E, k \geq 0 \text{ such that } \tau, D \xrightarrow{E,k}_{I_p} \tau', D'
\]
Proof. The proof can be shown by a structural induction on \( t \). The base statements are easily verified, since rules 3.1, 3.2, and 3.3 imply that the following reactions are valid:

\[
\begin{align*}
\text{nothing}, \ D & \xrightarrow{\emptyset,0} \text{nothing}, \ D \\
\text{pause}, \ D & \xrightarrow{\emptyset,1} \text{nothing}, \ D \\
\text{exit } T_d, \ D & \xrightarrow{\emptyset, d+2} \text{nothing}, \ D
\end{align*}
\]

Subsequently, only the sequential, parallel, and trap statements are of interest, since all other statements complete as unfinished transitions. For the trap statement, rule 3.21 is the only one that performs a finished transition. However, since the trap body can only pause if it consists of the \text{pause} statement, or a sequential or parallel statement containing \text{pause}, only two cases remain of interest:

1. Consider first, \( t = q;r \). Then, assume the induction hypothesis, that there always exist \( q' \) and \( r' \) such that

\[
q, \ D \xrightarrow{E_q,k_q} q', \ D' \quad k_q \geq 0 \quad (3.23)
\]

and,

\[
r, \ D \xrightarrow{E_r,k_r} r', \ D' \quad k_r \geq 0 \quad (3.24)
\]

Then,

- if \( k_q \geq 2 \), we simply get \( q; r, \ D \xrightarrow{E_q,k_q} q', \ D' \);
- if \( k_q = 1 \), we have, \( q; r, \ D \xrightarrow{E_q,1} q'; r, \ D' \);
- otherwise if \( k_q = 0 \), we have, \( q; r, \ D \xrightarrow{E_q \cup E_r, k_q} r', \ D' \).

Thus, any sequential composition of kernel statements will always be reactive.

2. Next, consider the case of \( t = q \parallel r \). Due to hypotheses (3.23) and (3.24), rule 3.17 will then ensure that \( q \parallel r \) will eventually finish, since both its branches will eventually perform finished transitions. Thus, any parallel composition of kernel statements will also always be reactive.

\[\square\]

Definition 2. A program is deterministic if for any statement \( t \) and data set \( D \), there exists at most one reaction for the set of signals registered in the previous instant, \( I^p \).
This definition requires that every sequence of transitions leading to a reaction, as given by equation 3.4, is unique. Intuitively, for this to hold, the rewrite rules for the kernel statements should have the following two properties:

1. **Rules that perform finished transitions need to be distinct and mutually exclusive with unfinished transitions.** This property expresses the fact that it should not be possible for a statement to perform an unfinished transition if it can perform a finished transition, and vice versa. Moreover, if a statement is going to perform a finished transition, then only one finished transition should be possible.

2. **Rules need to exhibit confluence, meaning that statements that may be rewritten in more than one way should eventually get rewritten to the same thing.** This property implies that statements that are rewritten into something else using different rules should eventually converge by getting rewritten to the same thing.

These two properties are formally expressed in the following two lemmas.

**Lemma 1.** If \( t, D \xrightarrow{O_1, k_1} t_1, D_1 \), then there exists no \( t_2, D_2 \) such that \( t, D \xrightarrow{O_2, k_2} t_2, D_2 \), where \( k_2 \geq 0 \). Conversely, if \( t, D \xrightarrow{O_2, k_2} t_2, D_2 \), where \( k_2 \geq 0 \), then there exists no \( t_1, D_1 \) such that \( t, D \xrightarrow{O_1, k_1} t_1, D_1 \). Moreover, there is only one way to perform a finished transition:

\[
t, D \xrightarrow{O_1, k_1} t_1, D_1 \text{ and } t, D \xrightarrow{O_2, k_2} t_2, D_2 \Rightarrow t_1 = t_2, D_1 = D_2, O_1 = O_2, \text{ and } k_1 = k_2, \text{ where } k_1, k_2 \geq 0.
\]

**Proof.** This is easily verified by observing that each base statement has only one rewrite rule. Therefore, finished transitions are always distinct, and mutually exclusive with unfinished transitions.

**Lemma 2.** The rules are locally confluent. Suppose \( t, D \xrightarrow{O_1, k_1} t_1, D_1 \) and \( t, D \xrightarrow{O_2, k_2} t_2, D_2 \). Then, there exists \( t', D', k \) such that \( t_1, D_1 \xrightarrow{O_1, k_1} \cdots \xrightarrow{\theta, k} t', D' \) and \( t_2, D_2 \xrightarrow{O_2, k_2} \cdots \xrightarrow{\theta, k} t', D' \), where \( k \geq 0 \).

**Proof.** Proof by structural induction on \( t \). Only the rules for the parallel statement are of interest here, since all other rules can be observed to provide only one way for terms to be rewritten in any given scenario. This observation provides the base
case for the inductive proof. Then, consider the case where \( t = q \| r \). Suppose that,

\[
q, D \xrightarrow{O_1,k_1,\dagger}{_IP} q_1, D_{q_1} \quad \text{and} \quad r, D \xrightarrow{Q_1,j_1,\dagger}{_IP} r_1, D_{r_1}
\]

If \( q \) and \( r \) both perform finished transitions, lemma 1 implies that these will be the only transitions that can be taken. Then, from rule 3.17, we obtain

\[
q \| r, D \xrightarrow{\emptyset,\text{syn}(k_1,j_1)}{_IP} q_1 \| r_1, D,
\]

since \( O_1 = Q_1 = \emptyset \) and \( D_{q_1} = D_{r_1} = D \), when \( q \) and \( r \) both perform finished transitions.

Otherwise, consider the case where \( q \) or \( r \) performs an unfinished transition (i.e., \( k_1 = \bot \) or \( l_1 = \bot \)). Assume the induction hypothesis that \( q \) and \( r \) are, respectively, locally confluent. Then,

- for \( q \), if \( q, D \xrightarrow{O_1,\dagger}{_IP} q_1, D_{q_1} \) and \( q, D \xrightarrow{O_2,\dagger}{_IP} q_2, D_{q_2} \), we have:

  \[
q_1, D_{q_1} \xrightarrow{O_1,1,\dagger}{_IP} \cdots \xrightarrow{\emptyset,k_m,\dagger}{_IP} q_m, D_{q_m} \quad \text{and} \quad q_2, D_{q_2} \xrightarrow{O_2,1,\dagger}{_IP} \cdots \xrightarrow{\emptyset,k_m,\dagger}{_IP} q_m, D_{q_m},
\]

  where \( k_m \geq 0 \).

- for \( r \), if \( r, D \xrightarrow{Q_1,\dagger}{_IP} r_1, D_{r_1} \) and \( r, D \xrightarrow{Q_2,\dagger}{_IP} r_2, D_{r_2} \), we have:

  \[
r_1, D_{r_1} \xrightarrow{Q_1,1,\dagger}{_IP} \cdots \xrightarrow{\emptyset,l_n,\dagger}{_IP} r_n, D_{r_n} \quad \text{and} \quad r_2, D_{r_2} \xrightarrow{Q_2,1,\dagger}{_IP} \cdots \xrightarrow{\emptyset,l_n,\dagger}{_IP} r_n, D_{r_n}
\]

  where \( l_n \geq 0 \).

If the transitions for \( q \) and \( r \) are allowed to take place until they finish, rule 3.17 will ensure that their interleavings will converge and, hence, we either get

\[
q \| r, D \xrightarrow{O_1,\dagger}{_IP} q_1 \| r, D_{q_1} \cdots \xrightarrow{\emptyset,\text{syn}(k_m,j_n)}{_IP} q_m \| r_n, D'
\]

or

\[
q \| r, D \xrightarrow{Q_1,\dagger}{_IP} q \| r_1, D_{r_1} \cdots \xrightarrow{\emptyset,\text{syn}(k_m,j_n)}{_IP} q_m \| r_n, D'
\]

where \( D' = (D_{q_1} \setminus D) \cdots \cup (D_{q_m} \setminus D) \cup (D_{r_1} \setminus D) \cdots \cup (D_{r_n} \setminus D) \cup (D_{q_1} \cdots \cap D_{q_m} \cap D_{r_1} \cdots \cap D_{r_n}) \).

\[\square\]

**Theorem 2.** All synchronous function block programs are deterministic, that is:

\[
\forall t, D, \forall E, k, t', D', \forall F, l, t'', D'',
\]

\[
t, D \xrightarrow{E,k,\dagger}{_IP} t', D' \quad \text{and} \quad t, D \xrightarrow{F,l,\dagger}{_IP} t'', D'' \Rightarrow E = F, \ k = l, \ t' = t'', \ \text{and} \ D' = D''.
\]
3.4. DISCUSSION

Proof. The proof can be established by assuming the existence of a non-deterministic program, and then showing that this assumption contradicts the proven lemmas. Suppose we have the situation where

\[ t, D \xrightarrow{O_{i}, \perp} \ldots \xrightarrow{O_{i}, \perp} t_{i}, D_{i} \xrightarrow{\theta_{k}, /} t', D', \]

where \( k \geq 0 \), as well as

\[ t, D \xrightarrow{Q_{j}, \perp} \ldots \xrightarrow{Q_{j}, \perp} t_{j}, D_{j} \xrightarrow{\theta_{l}, /} t'', D'', \]

where \( l \geq 0 \). Then, by lemma 2, we either have

\[ t_{i}, D_{i} \xrightarrow{O_{i+1}, \perp} \ldots \xrightarrow{\theta_{l}, /} t'', D'', \]

or

\[ t_{j}, D_{j} \xrightarrow{Q_{j+1}, \perp} \ldots \xrightarrow{\theta_{k}, /} t', D'. \]

However, since the next transition of \( t_{i}, D_{i} \) and \( t_{j}, D_{j} \) are, respectively, finished transitions, Lemma 1 will be contradicted, unless \( t_{i} = t_{j}, D_{i} = D_{j}, \) and \( O_{i} = Q_{j}, \) and hence, \( t' = t'', D' = D'', k = l, \) and \( (E = \bigcup_{m} O_{m}) = (F = \bigcup_{n} Q_{n}) \forall m \in [1, i] \) and \( \forall n \in [1, j] \).

Owing to the synchronous semantics proposed in this section, the reactivity and determinism of function block programs are now guaranteed by construction. This has been achieved by restricting instantaneous signal reactions in exchange for better compositionality.

3.4 Discussion

This chapter has introduced a synchronous model for function blocks, which can be used to reason about the behaviour of function blocks in a network. In the past, various scheduling policies have been used to emulate the concurrent behaviour of function blocks in a network. One common approach that has been adopted makes use of event emissions to schedule function blocks [17, 19, 50]. The sequence of event generation and their order of propagation through the network provided the basis for the schedules. Typically, the function block that will be triggered by the emitted event would be executed next. While many variations of this exist, they all have been noted to have unpredictable timing characteristics [15].
An alternative approach for scheduling uses a cyclic executive instead, similar to that of a PLC scan cycle [52]. In this model, the concurrent execution is simulated by executing a portion of code for each function block in a network during each scan cycle. Unlike the approach based on event emissions, this approach gives predictable temporal properties, making it the preferred choice for safety-critical systems. The function block execution order within a cycle, however, becomes a crucial implementation detail, since the behaviour of the application directly depends on it.

The synchronous model proposed in this work follows a similar cyclic execution method. However, the proposed approach does not suffer from the behavioural variations arising from different function block execution orders within a cycle, as in [52]. This is because computations in synchronous function blocks occur instantaneously in each tick, with their outputs guaranteed to be deterministic by the synchronous semantics. As such, run-time scheduling policies are not needed, since concurrent components can compute and synchronize with each other instantaneously.

This synchronous model, however, differs from a previous approach that mapped function blocks to the synchronous language SIGNAL [46]. In the proposed model, a single clock tick drives the synchronous program. This clock is assumed to be fast enough to observe all incoming input events and to produce the corresponding outputs before the start of the next tick. This is unlike the former work, which allows different clocks to be used to read different input events. A union of these clocks, called $C_{union}$, is then computed. To ensure that output events are produced before the next input event arrives, $C_{union}$ is overclocked to obtain an event clock whose period is equal to a fraction of the smallest interval between two dates of $C_{union}$. The synchronous model prescribed in this chapter is simpler, while the formal semantics given facilitates compiler construction and direct code generation without needing to go through an intermediate synchronous language.

A key feature of the proposed synchronous semantics is the requirement for delayed communication. This is comparable to the semantics of STATEMATE Statecharts [6], where outputs in the current step are sensed only in the next step. The closest work related to this idea, however, was proposed for the prototype language, SL [33]. There are key differences, though. SL allows instantaneous reactions to signal presence, but delays reactions to signal absence till the next instant. Signals are determined to be present only if they are emitted, while their absence can only be decided at the end of the current instant. Thus, SL programs resolve signal dependencies during run-time. The execution of threads containing a test of an unresolved signal is suspended, until the signal is emitted, or until all other parallel threads pause, terminate, or become suspended as well. In this case, all unresolved
signals will be recognised as absent.

In contrast, all communication in synchronous function blocks always occur with respect to registered signals\(^1\). This avoids the need for slow run-time resolution of signal dependencies, and results in delayed reactions to both signal presence and absence. SL programs may avoid the run-time overhead by compiling to automata, but at the cost of an exponential increase in code size, and compilation complexity. Moreover, automata generation hinders modular compilation, while the technique proposed here makes possible the modular compilation of function blocks in true component-oriented fashion. Besides this, the communication of data was not at all discussed in [33], but full treatment for it is provided in this work.

Modular compilation of synchronous programs has been known, in general, to be difficult [24]. However, by delaying all communications to the next instant, the transitions within a given ECC can no longer be affected by other blocks running in parallel. Therefore, complete knowledge of the network would no longer be required during compilation, as blocks can be arbitrarily scheduled in each instant. Such modularity in compilation is of practical importance to industrial engineers, who would be inclined to treat function blocks simply as opaque reusable components in their designs. This enables synchronous programming methodologies to be encapsulated within function blocks so that they can be easily understood and applied by industrial engineers who may not be familiar with synchronous languages.

Finally, the SOS rules presented in Section 3.3 may be viewed as a slightly more fundamental work that contributes to earlier proposals for weakening the synchrony hypothesis, along the lines of what has been done in [33]. While the work here has been presented in the context of function blocks, the semantics itself may find application in other domains requiring similar features.

\(^1\) The \texttt{pre} of signals in Esterel serve as registered signals.
Earlier on in Chapter 3, the proposal for synchronous function blocks was presented via a mapping of IEC 61499 function blocks to the Esterel programming language. Esterel inspired many of the ideas presented there, as its built-in primitives for expressing concurrency and synchronized communication are well-suited for describing concurrent reactive systems. Such systems are typical of many function block programs.

The mapping of function blocks to Esterel provides numerous benefits. It enables function block programs to take advantage of the available tools for Esterel, like Esterel Studio [67], for the purpose of verification and code synthesis. Since existing tools for formal verification cannot operate directly on function block descriptions, a translation to Esterel provides a means to quickly benefit from the available tools in the synchronous domain. Besides, synchronous languages—including Esterel—are already widely-accepted as the choice technology for specifying, validating, and implementing safety-critical embedded systems [24]. While previous attempts to formally verify function block programs have been confined to control properties (as mentioned in Chapter 2), the translation to Esterel enables both control and data properties to be verified at once. The verified Esterel code can then be automatically synthesized to C for execution, thereby enabling function block programs to run without the need for any run-time environment.

The following section will go on to show how function blocks can be mapped to Esterel.
### 4.1 Compiling function blocks to Esterel

The translation of function blocks to Esterel is done in a syntax-directed manner. Table 4.1, which appeared earlier as Table 3.2, summarizes the essential mappings between function block elements and their Esterel counterparts. Most of the mappings are straightforward—function blocks are mapped to Esterel modules, while their event and data ports become interface signals to the modules.

The key point to note here is the use of the `pause` statement to demarcate states in an ECC. This implements the synchronous model for function block execution, as described in Chapter 3. As a consequence, the computation of algorithms in each state must fit within the duration of a `tick`. For this approach to work, the `tick` in Esterel is assumed to be fast enough to observe every event occurring in each state, as required by the synchrony hypothesis. This assumption is fundamental to every Esterel program, and must be valid for the synchrony hypothesis to hold [24]. Then, instead of evaluating transitions in an ECC with respect to an input event, transitions are now evaluated at each `tick`. If the transition condition does not evaluate to true in a given `tick`, no state transition will take place. This approach precisely bounds the lifespan of an event within the `tick` in which it occurred.

A prototype function block compiler, called FBC, has been developed to do this translation. The code generated by FBC conforms to Esterel’s V7 textual syntax [68]. FBC begins the translation process by taking as input a function block network, created by a function block editor (e.g., FBDK [12]) in the XML format. This format is the standard format prescribed by IEC 61499 [18]. FBC will do a depth-first traversal of the network, recursively entering each composite function block it encounters, to perform a bottom-up compilation of every block in the network. Each network will be instantiated within a top-level module in Esterel. If the input XML file contains networks for different resources, FBC will generate the network for each

<table>
<thead>
<tr>
<th>Function block element</th>
<th>Esterel feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function blocks</td>
<td>Modules</td>
</tr>
<tr>
<td>Events</td>
<td>Pure interface signals</td>
</tr>
<tr>
<td>Data</td>
<td>Value-only interface signals</td>
</tr>
<tr>
<td>Event-data binding</td>
<td>Association of pure and value-only signals</td>
</tr>
<tr>
<td>Internal variables</td>
<td>Value-only local signals</td>
</tr>
<tr>
<td>EC states</td>
<td>Demarcated by <code>pause</code> statements</td>
</tr>
<tr>
<td>Algorithms</td>
<td>Instantaneous modules</td>
</tr>
<tr>
<td>Transition conditions</td>
<td><code>await</code> statements</td>
</tr>
<tr>
<td>Function block network</td>
<td>Parallel composition of modules</td>
</tr>
</tbody>
</table>

Table 4.1: Function block to Esterel translation map.
4.2 Translating the basic function block

The translation steps in this stage proceed as follows:

1. Create a new module for each basic function block type.

2. Declare all input/output events of the function block as pure input/output signals, and input/output data as value-only signals at the module interface.

3. Declare all internal variables in the function block as local value-only signals in the Esterel module.

4. Extract state and transition information from the ECC to generate Esterel code.

5. Create an additional module for each function block algorithm.

The main work in these steps lie in extracting the state and transition information from the ECC. Each node in the ECC will be parsed to create a synchronous state representation, called $SyncState$, consisting of the quadruple \{Actions, Transitions, Children, Parents\}, where:

- **Actions** is the list of algorithms and output events to be issued;
- **Transitions** is the list of transition conditions leading to some successor state;
- **Children** is the list of successor states; and
- **Parents** is the list of predecessor states.

As an example, the ACCEL state in Figure 1.3 will be described by the quadruple \{[accel, throttleChg], [cruiseOff, accelReleased, clock], [NORMAL, CRUISE, ACCEL], [CRUISE, ACCEL]\}. Once each EC state has been converted to a $SyncState$, the resulting object is a directed graph with a starting $SyncState$ corresponding to the initial EC state. Using this directed graph, the ECC can be converted to Esterel using the algorithm in Figures 4.1–4.3.

The TranslateECC procedure initiates the conversion of an ECC to Esterel by invoking two recursive functions: the first function makes the ECC amenable to
procedure TranslateECC
1    generate interface signals for input events, $IE$;
2    generate interface signals for output events, $OE$;
3    generate interface signals for input data, $ID$;
4    generate interface signals for output data, $OD$;
5    generate local signals for input data, $ID'$;
6    generate local signals for output data, $OD'$;
7    foreach $ie \in IE$ associated with $id \in ID$ do
8        generate code to update $id'$ with $id$ whenever $ie$ occurs;
9    end

10  $s :=$ initial state for given ECC;
11  $G :=$ set for storing top-level states;
12  FormatECC($s, G$);
13  if $G.size() > 1$ then
14      foreach $n \in G$ do
15          generate switch-case for $n$; // simulates goto to $n$;
16          GenerateCode($n, G$);
17      end
18  else
19      GenerateCode($s, G$);
20  end
21  foreach $oe \in OE$ associated with $od \in OD$ do
22      generate code to update $od$ with $od'$ whenever $oe$ occurs;
23  end
24 end procedure

Figure 4.1: Algorithm to initiate the translation of an ECC into Esterel.

procedure FormatECC($s, G$)
1    if state, $s$, has been visited then
2        return;
3    end
4    mark $s$ as visited;
5    if $s$ has > 1 parents then
6        if initial state $\notin$ set, $G$ then
7            add initial state to $G$;
8        end
9    end
10    if $s \neq$ initial state then
11        add $s$ to $G$;
12    end
13 end
14 foreach child $c$ of $s$ do
15    FormatECC($c, G$);
16 end
17 end procedure

Figure 4.2: Algorithm to prepare an ECC for structured code generation.
procedure GenerateCode(s, G)
  if state, s, has been visited then
    return;
  end

  mark s as visited;
  foreach action, a, of s do
    if a has algorithm, alg then
      generate call to procedure[alg];
    end
    if s has event output, eo then
      forall data output, dout, associated with eo do
        generate emission of signal[dout];
      end
      generate emission of signal[eo];
    end
  end
  generate new state boundary;
  foreach transition, t, of s do
    if t is always true then
      break;
    end
    if G.size() > 1 and t leads to child, n ∈ G then
      synthesize “goto” statement to n;
      return;
    else
      generate code to await t;
    end
  end
  foreach child, c, of s do
    GenerateCode(c, G);
  end
end procedure

Figure 4.3: Algorithm to generate Esterel code for a given ECC.
Esterel, while the second function generates actual code from it. Data ports in function blocks need to be specially handled. Lines 4–7 of Figure 4.1 create one *interface signal* and one *local signal* for each data port in a function block. The interface signal is used to represent the data value at the interface, while the local signal is used to indicate the value as seen within the function block. This is required to correctly handle the possibility of input data being received at different times from their associated input events. In such cases, the value of the input data as seen by the function block’s internal algorithms would still be that of the previous instant. The input data would only be updated internally with the new value when the event associated with that input data is received. This is done on lines 8–10.

Likewise, a similar situation may also occur at the output interface of the function block. In this case, it is possible that internal algorithms may update the values of output data at different times from the emission of the associated output event. All associated output signals will, thus, need to be updated with their new values, as computed by the internal algorithms (lines 22–24). Hence, FBC always generates two value-only signals—one interface and another local—for each input/output data of the function block.

The main tasks performed by this algorithm are detailed in the following subsections.

### 4.2.1 Generating code for state transitions

FBC attempts to generate structured code using nested `await-case` statements for each node in an ECC. In general, however, the control-flow among states in an ECC is unstructured, as transitions between any arbitrary pair of states are possible. This may require excessive duplication of nodes in order to still produce structured code. On the other hand, Esterel is a highly structured language with no “goto” primitive. Thus, in order to avoid node duplication, “goto” behaviour would need to be simulated using an additional *loop* and *switch* statement.

The `FormatECC` procedure of Figure 4.2 performs a depth-first traversal to determine the set of states, $G$, for which transitions must be implemented using a simulated “goto.” For this, the algorithm simply looks for states with more than one predecessor and adds them to the set, $G$ (see lines 6–13). States in this set will be assigned with distinct indices to encode their “goto” position. These states cannot be reached using nested `await-case` statements without duplicating them in each `case` from which they may be reached. If such states are found, the initial state of the ECC is automatically added to the set $G$ (lines 7–9) to enable states in
nested \texttt{await-case} statements to “go to” them using a state variable.

\begin{figure}[h]
\centering
\begin{subfigure}{0.4\textwidth}
\begin{tikzpicture}
\node (Default) at (0,0) {Default};
\node (CaseA) at (1,1) {CaseA};
\node (CaseB) at (0,-1) {CaseB};
\node (CaseC) at (2,1) {CaseC};
\node (CaseD) at (1,-2) {CaseD};
\node (CaseE) at (2,-1) {CaseE};
\node (CaseF) at (3,-2) {CaseF};
\draw (Default) -- (CaseA);
\draw (Default) -- (CaseB);
\draw (CaseA) -- (CaseC);
\draw (CaseA) -- (CaseD);
\draw (CaseB) -- (CaseA);
\draw (CaseB) -- (CaseD);
\draw (CaseC) -- (CaseD);
\draw (CaseC) -- (CaseF);
\draw (CaseD) -- (CaseE);
\draw (CaseD) -- (CaseF);
\draw (CaseE) -- (CaseF);
\end{tikzpicture}
\caption{(a)}
\end{subfigure}
\begin{subfigure}{0.6\textwidth}
\begin{lstlisting}
loop
switch stateVar
  case 1 do
    pause;
    await
    case immediate pre(E) do
      pause;
      await immediate pre(G);
      stateVar := 1;
      case immediate pre(F) do
        ...
        end await;
    case immediate pre(A) do
      pause;
      await
      case immediate pre(C) do
        ...
        case immediate pre(D) do
          stateVar := 1;
          end await;
        case immediate pre(B) do
          ...
        end await;
    end switch;
end loop;
\end{lstlisting}
\caption{(b)}
\end{subfigure}
\caption{An illustration on how code is generated for state transitions. (a) This is a fragment of an ECC. (b) Since state CaseD has two predecessors, it is added together with the initial state (Default) to the set $G$. This causes code for both states to be generated as distinct cases in a \texttt{switch} statement (lines 3 and 13). States that are not in $G$ are simply nested within their respective \texttt{await} statements.}
\end{figure}

Figure 4.4: An illustration on how code is generated for state transitions. (a) This is a fragment of an ECC. (b) Since state CaseD has two predecessors, it is added together with the initial state (Default) to the set $G$. This causes code for both states to be generated as distinct cases in a \texttt{switch} statement (lines 3 and 13). States that are not in $G$ are simply nested within their respective \texttt{await} statements.

Code will then be generated to test for these indices so that the appropriate state can be entered (lines 14–21 of Figure 4.1). Then, whenever a transition is encountered that leads to a state in $G$, a “goto” to that state will be synthesized by setting the next state variable to the appropriate state index. States with only a single predecessor, however, will always be nested below the \texttt{await} statement of its predecessor without needing to be encoded with a state variable. All these are accomplished during code generation in lines 22–27 of the \texttt{GenerateCode} procedure. Figure 4.4 provides an illustration of this process.

The \texttt{await} statements generated always test for the \texttt{pre} of a given condition. This effectively ensures that the compositional semantics described in Chapter 3 is
adhered to. Moreover, since the evaluation order of transitions from any given state is statically determined by their order of appearance in the function block XML description (recall page 33 of [5]), any sibling transition following an always true transition will never be reachable during execution. Lines 19–21 of the GenerateCode procedure check for such unreachable states in the ECC. This is done using constant folding, a common compiler optimization technique, to check for transition conditions expressed entirely in literals [69]. This simple static evaluation prunes unreachable states in order to produce more compact code.

4.2.2 Generating code for implementing algorithms

Algorithms in function blocks are procedures that enable conventional data computations to be performed within states of an ECC. One possible way to do the same in Esterel is to map each algorithm to a corresponding host procedure in Esterel, as proposed in [34]. Host procedures are defined externally in a host language, like C, and are unknown at the Esterel level. With this method, internal variables that are used within the algorithms need to be passed to the corresponding host procedures as arguments. Furthermore, since output data can be directly modified by algorithms in function blocks but signal values cannot be similarly modified in host procedures, additional variables representing signal values need to be generated and passed to the procedures. The main drawback of this method, however, is the resulting inability to verify data properties as host language computations are completely abstracted by Esterel’s verifier.

Due to this limitation, function block algorithms are more suitably mapped to modules instead. Such modules are instantaneous, as they do not contain any pause statements within them and, thus, differ from the modules translated from function blocks. Instantaneous modules implementing algorithms are differentiated from function block modules here by referring to them as algmodules and fbmodules respectively. Currently, Fbc is able to accept algorithms specified either in Esterel, or in Structured Text (ST), a commonly used PLC language [4]. ST is a conventional procedural language with syntax that resembles the Pascal programming language. Fbc can translate a subset of ST directly into Esterel, and can be extended to accept other languages as well. At the moment, Fbc performs the translation of ST to Esterel using a simple syntax-directed lookup table. This mapping from ST to Esterel is shown in Table 4.2.

With this approach, input/output data and internal variables used in algorithms are passed as valued signals through signal bindings between the fbmodules and
### 4.2. TRANSLATING THE BASIC FUNCTION BLOCK

<table>
<thead>
<tr>
<th>ST keyword and operator</th>
<th>Equivalent Esterel syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF...THEN...ELSE...END_IF</td>
<td>if...then...else...end if</td>
</tr>
<tr>
<td>CASE...OF...END_CASE</td>
<td>switch...case...do...end switch</td>
</tr>
<tr>
<td>VAR...END_VAR</td>
<td>var...end variable</td>
</tr>
<tr>
<td>BOOL</td>
<td>bool</td>
</tr>
<tr>
<td>SINT</td>
<td>signed&lt;[8]&gt;</td>
</tr>
<tr>
<td>INT</td>
<td>signed&lt;[16]&gt;</td>
</tr>
<tr>
<td>DINT</td>
<td>signed&lt;[32]&gt;</td>
</tr>
<tr>
<td>LINT</td>
<td>signed&lt;[64]&gt;</td>
</tr>
<tr>
<td>USINT or BYTE</td>
<td>unsigned&lt;[8]&gt;</td>
</tr>
<tr>
<td>UINT or WORD</td>
<td>unsigned&lt;[16]&gt;</td>
</tr>
<tr>
<td>UDINT or DWORD</td>
<td>unsigned&lt;[32]&gt;</td>
</tr>
<tr>
<td>ULINT or LWORD</td>
<td>unsigned&lt;[64]&gt;</td>
</tr>
<tr>
<td>REAL</td>
<td>float</td>
</tr>
<tr>
<td>LREAL</td>
<td>double</td>
</tr>
<tr>
<td>STRING</td>
<td>string</td>
</tr>
<tr>
<td>:=</td>
<td>:=</td>
</tr>
<tr>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>MOD</td>
<td>mod</td>
</tr>
<tr>
<td>**</td>
<td>**</td>
</tr>
<tr>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>&lt;=</td>
<td>&lt;=</td>
</tr>
<tr>
<td>=&gt;</td>
<td>=&gt;</td>
</tr>
<tr>
<td>NOT</td>
<td>not</td>
</tr>
<tr>
<td>&amp; or AND</td>
<td>and</td>
</tr>
<tr>
<td>XOR</td>
<td>xor</td>
</tr>
<tr>
<td>OR</td>
<td>or</td>
</tr>
<tr>
<td>TRUE</td>
<td>true</td>
</tr>
<tr>
<td>FALSE</td>
<td>false</td>
</tr>
<tr>
<td>(...)</td>
<td>(...)</td>
</tr>
<tr>
<td>[...]</td>
<td>[...]</td>
</tr>
<tr>
<td>(<em>...</em>)</td>
<td>/<em>...</em>/</td>
</tr>
</tbody>
</table>

Table 4.2: Structured Text (ST) to Esterel translation map.
module Throttle_normal:
  input cruiseSpeed_ : value signed<[16]>
  input speed_ : value signed<[16]>
  input throttleOffset_ : value signed<[8]>
  inputoutput throttleVal_ : value signed<[8]>
  inputoutput prevOffset_ : value signed<[8]>
  var cruiseSpeed : signed<[16]> := pre(?cruiseSpeed_),
    speed : signed<[16]> := pre(?speed_),
    throttleOffset : signed<[8]> := pre(?throttleOffset_),
    throttleVal : signed<[8]> := pre(?throttleVal_),
    prevOffset : signed<[8]> := pre(?prevOffset_) in
    throttleVal := throttleOffset;
    prevOffset := 0;
    emit ?throttleVal_ <= throttleVal;
    emit ?prevOffset_ <= prevOffset;
  end var
end module

Figure 4.5: Esterel code for the normal algorithm in the ECC of Figure 1.3.

algmodules. Input data are bound as input signals, while output data and internal variables are bound as inputoutput signals, to allow any modification of their values in the algmodule to be propagated back to the fbmodule. Within the algmodules, FBC automatically maps these signals back to variables so that the actual algorithms need not be modified to account for the different syntax and semantics for the signals and variables.

Figure 4.5 illustrates the algmodule generated for the normal algorithm in the ECC of Figure 1.3. The normal algorithm consists of two assignment statements (lines 12–13). Lines 2–4 are input signals corresponding to data inputs, while lines 5–6 are inputoutput signals corresponding to output data and an internal variable of the Throttle function block, respectively (compare Figure 1.2). Meanwhile, the mapping of these signals to variables and back are accomplished on lines 7–11 and lines 14–15, respectively. Only the variables corresponding to inputoutput signals need to be mapped back, since the values of input data to a function block cannot be modified by an algorithm. This algmodule will get instantiated by the Throttle fbmodule, as shown on line 29 in Figure 3.5.

4.3 Translating a function block network

Once every function block type in a network has been mapped into an equivalent fbmodule, the corresponding fbmodules will be composed in parallel in a top-level
module CruiseControl:
  input cclock, set, off, resume, quickAccel, quickDecel;
  // ... other inputs omitted ...
  output speed, speedSet, throttleChg, regulOff;
  // ... other outputs omitted ...
  signal s0 : value signed<[16]>, s1, s2, s3, s4, s5,
      s6 : value signed<[16]>, s7, s8 in
      run SpeedGauge [signal0_/speedVal];
      ||
      run CruiseManager [s1/regulResume, s2/regulSet,
          s3/regulOff, s4/regulStdby, s5/speedSet,
          s6/cruiseSpeed, s0/speed];
      ||
      run Throttle [s7/cruiseOff, s8/cruiseOn, s0/speed,
           s6/cruiseSpeed, s5/speedSet];
      ||
      loop
        emit regulResume<=s1; emit ?cruiseSpeed<=?s6;
        emit speedSet<=s5; emit regulSet<=s2;
        emit regulOff<=s3; emit s7<=s3 or s4;
        emit ?speedVal<=?s0; emit regulStdby<=s4;
        emit s8<=s1 or s2; pause;
      end loop;
  end signal;
end module

Figure 4.6: A fragment of the Esterel code automatically generated from the function block network in Figure 1.1.

module to instantiate the network. Figure 4.6, which first appeared as Figure 3.4 in Chapter 3, is reproduced here to show how this is done for the cruise control system of Figure 1.1. The event and data connections between function blocks are accomplished through appropriate signal binding at the module interfaces. Local signals in Esterel are declared (lines 6–7) to perform this binding.

The final parallel branch (lines 17–23) manages the associations between the local signals and the interface signals of the fbmodules. They are generated for local signals whenever an output signal is emitted to more than one destination, or whenever an input signal is received from more than one source. This enables each input/output port of a fbmodule to be bound to a separate local signal without resulting in erroneous overlaps in the connections. The enclosing loop simply ensures that the local signals will be initialized with the right values in each tick of the program.

It is worth noting that the translation of function blocks to Esterel described here
and in the previous section closely follows the mapping of function blocks to kernel statements, as given in Section 3.3. The use of the pause statement to demarcate states and the || operator to express synchronous concurrent execution of blocks are semantically equivalent in both cases. Meanwhile, the use of the await statement to retain program control at a given EC state in Esterel can be easily derived through a nesting of the trap, loop, and present/if kernel statements. For example, the await immediate pre(G) statement in Figure 4.4b can be expressed using the kernel statements below:

```plaintext
trap T in
  loop
    present pre(G) then
      exit T
    else
      pause
    end
  end loop
end trap
```

The translation to Esterel can, thus, be thought of as a concrete implementation of the semantics described in Section 3.3.

### 4.4 Automatic verification

The resulting Esterel code for a given function block description can be verified using the synchronous approach. Verification goals that are of interest for function block programs typically fall under the category of safety properties [70]. Safety properties are properties which assert that something bad will never happen—i.e., that the program will never enter an unacceptable state.

Such properties can be easily expressed by means of a synchronous observer [25], which is another program that observes the behaviour of the first one and decides whether or not it is correct. The observer will describe the safety properties, or equivalently, the unwanted traces of the program. The verification then consists of checking that the parallel composition of the program and its observer never causes the observer to complain, meaning that no trace of the program is accepted by the observer. This is equivalent to checking that the product automaton for the composition never enters an erroneous state.

Verification by observers is natural in a synchronous framework, since the product automaton used to compute trace intersection is precisely the parallel composition provided by the framework. As such, safety properties for the original function
4.4. AUTOMATIC VERIFICATION

Figure 4.7: The cruise control system function block network with the observer for property A1. The Activate function block serves as the observer.

Block program can be expressed using function blocks itself without needing to learn a new mathematical dialect (e.g., temporal logic). The observer function blocks can be automatically converted to appropriate synchronous observers using a method similar to what had been described in Section 4.2. This simplicity is practically important for industry adoption as industrial engineers are usually unfamiliar with the theoretical foundations of model-checking.

Typically, when verifying a certain property with an observer, signals of interest will be observed by the observer in each tick to see if the desired property is satisfied up to this instant. If this is not the case, a signal will be emitted by the observer to indicate that a violation has occurred. This process is usually done offline, with the observers removed from the program description once the desired properties have been verified.

An actual verification process will be illustrated using the cruise control example. The model checker in Esterel Studio, called Design Verifier, has been used to do this. Table 4.3 shows the list of safety properties that were checked using Design Verifier for the cruise control system, as well as for some other programs adapted from [71, 72]. These have been appropriately recreated using function blocks. The times taken to verify all the properties are also provided in Table 4.3. An illustration of the observer for property A1 of the cruise control system, as well as the Esterel code generated for the observer, are illustrated in Figures 4.7 and 4.8 respectively.

Properties A1 and A4 were verified successfully, taking 10 and 25 seconds respec-
<table>
<thead>
<tr>
<th>Program</th>
<th>Properties</th>
<th>C/D</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cruise control system</td>
<td>The CRUISE mode cannot be activated if the current vehicle speed is $&lt;50\text{km}/\text{h}$ or $&gt;170\text{km}/\text{h}$ [A1]. Pressing quickAccel or quickDecel will never set the desired speed to $&lt;50\text{km}/\text{h}$ or $&gt;170\text{km}/\text{h}$ [A2]. The CRUISE mode must be deactivated when the brake pedal is pressed [A3]. While in STANDBY mode, pressing resume will return the system to the CRUISE mode provided the brake pedal is not pressed, and the off and set buttons are not pressed either [A4].</td>
<td>D</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temperature controller</td>
<td>The controller will be in the OFF mode if the off button is pressed [B1]. The controller will be in the IDLE mode if it is ON and the temperature is just right [B2]. The heater is on if the controller is ON and the temperature is below the desired value [B3]. The air conditioner is on if the controller is ON and the temperature is above the desired value [B4].</td>
<td>C</td>
<td>&lt;1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D</td>
<td>&lt;1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D</td>
<td>&lt;1</td>
</tr>
<tr>
<td>Water level monitor</td>
<td>The system will always be in the TEST mode when the SelfTest button is pressed constantly for $&gt;500\text{ms}$ [C1]. The system will only operate if it is not in the TEST mode and the water level is within the desired limits [C2]. The system will be in the SHUTDOWN mode if it has not been in that mode for $&gt;200\text{ms}$, and neither the water level is within the hysteresis range nor the TEST mode has been activated [C3].</td>
<td>C</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D</td>
<td>6</td>
</tr>
<tr>
<td>Railroad crossing con-</td>
<td>The gate is down whenever a train is inside the railroad crossing [D1]. The gate can never be lowered and raised simultaneously [D2]. If there is no train inside the railroad crossing, the gate will be up within 5 time units [D3].</td>
<td>C</td>
<td>25</td>
</tr>
<tr>
<td>troller</td>
<td></td>
<td>C</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D</td>
<td>132</td>
</tr>
</tbody>
</table>

Table 4.3: Verification times for test programs—the C/D column indicates whether the verified property is for control (C) or data (D).
4.5. DISCUSSION

![Esterel code for the observer for property A1]

```plaintext
module Activate:
    input regulSet, speedVal : value signed<[16]> : init 0;
    output violated;
    loop
        await immediate pre(regulSet);
        if (pre(?speedVal)<50 or pre(?speedVal)>170) then
            emit violated;
        end if;
        pause;
    end loop;
end module
```

Figure 4.8: Esterel code for the observer for property A1. The violated signal will be emitted (line 7) if the property in A1 is violated.

Property A3, however, was found to be falsifiable. Design Verifier provided a counter-example consisting of an input trace that led to the failure of this property in eight seconds. The flaw was traced to an error in the CruiseManager’s ECC, which allowed the cruise control system to enter into the Cruise mode when the brake pedal and the set button are pressed simultaneously. Once this error was corrected, Design Verifier took 25 seconds to verify that property A3 holds. The verification of property A2, however, is inconclusive as Design Verifier was aborted after attempting to verify it for 20 hours.

For models containing significant amounts of data computation, the verification may sometimes fail to complete due to their much larger state space. Still, the approach proposed here provides a plausible way to verify both data and control properties in function block programs using the synchronous approach. Moreover, when compared to dynamic testing, this technique is advantageous, as it is fully automated and can be used to cover every reachable state of the program.

4.5 Discussion

This chapter has presented a function block compiler, called FBC, that translates function blocks to the synchronous language Esterel. The code generated by FBC conforms to Esterel’s V7 textual syntax, which is accepted by Esterel Studio [67]. Although Esterel Studio also allows programs to be described graphically using Safe State Machines (SSMs) [7] that resemble ECCs, FBC produces the textual form, as its syntax is well-documented, unlike the proprietary format for SSMs. Moreover, the use of SSMs does not simplify the key algorithms of FBC in any way, as the same
state and transition information would need to be extracted from a given ECC.

The automatic translation of function blocks to Esterel has enabled safety properties involving both data and control parts of function block programs to be verified using existing tools for Esterel. The ability to do this is significant, as mature tools for function block verification are virtually non-existent. Moreover, prototypes that do exist, like those based on Net Condition/Events Systems (NCES) [37, 40], are completely unable to verify data properties. They further require the designer to learn a new mathematical dialect, as properties to be verified need to be specified in temporal logic using existing tools [40]. The approach proposed here removes this limitation, and allows safety properties to be specified directly in function blocks instead of some mathematical formalism. This solution makes formal verification easily accessible to engineers that are already familiar with IEC 61499 notations. The limitations of the NCES approach notwithstanding, direct comparisons with it for verification are still, nonetheless, not possible, as the semantics of synchronous function blocks are substantially different to that of the NCES model.

For the first time, code for function block programs can be synthesized directly from the verified specification. This code can run on its own without needing a run-time environment. Most existing verification techniques for function blocks are unable to generate executable code from the verified specification [37, 41, 45]. The only other known work [46] that proposed a mapping of function blocks to a synchronous language did not provide an automated approach to obtain synchronous programs and observers directly from function block specifications.
Efficient Code Synthesis from Function Blocks

The preceding chapters have presented a compositional semantics for function blocks, followed by a translation of function blocks to Esterel that adheres to that semantics. In this chapter, the issue of efficient code synthesis will be explored by directly compiling function blocks to C, while still preserving the proposed semantics. In order to enhance the expressivity of that semantics, a slight variant of it will also be explored in this chapter.

Up to now, the idea of synchronous function blocks has been discussed in the context of the Esterel programming language. Semantic issues regarding the causal composition of function blocks, as well as implementation issues concerning the verification and synthesis of function blocks, have been illustrated via translations to Esterel. In particular, the exposition of the previous chapter demonstrates the feasibility of this approach through a prototype tool that automatically transforms function block descriptions into equivalent Esterel programs.

The methodology described so far, however, is not ideal in at least two ways. Firstly, while the translation to Esterel benefits from the existing tools for the verification of Esterel programs, the use of Esterel as an intermediate specification introduces some inefficiencies in the final function blocks implementation. This is because Esterel’s semantics are more general than the semantics proposed for synchronous function blocks. Esterel allows both instantaneous and delayed communications between concurrent components, but synchronous function blocks are restricted to delayed communications only. This semantic restriction and the structure of synchronous function blocks can actually be exploited during code generation. Thus, the direct translation of function blocks to C could potentially result in more efficient code.
Secondly, the requirement for delayed communication between function blocks (using the `pre` operator in Esterel) to ensure causal compositions limits the expressivity of function block programs. Recall, from Chapter 3, that the requirement for a unit delay in communication was introduced for the following advantages:

1. *It ensures reactivity and determinism by construction.* Since non-causal code is completely avoided with delayed communication, programs are guaranteed to be reactive and deterministic. This is unlike general Esterel programs, which require costly causality analysis to ensure the same [61].

2. *It enables separate compilation of function blocks.* Since causality analysis is not needed, individual function blocks can be modularly compiled and arbitrarily connected in a network at a later stage. This is of practical importance, as industrial engineers would be inclined to treat function blocks as opaque reusable components in their designs.

These advantages, however, come at the expense of additional delays whenever events/data need to be passed between blocks. Due to this, it is not possible to express the propagation of events between a series of function blocks as occurring within a single *tick*. Hence, it is desirable for this requirement for delayed communication to be relaxed whenever possible.

In order to do so, it is helpful to reason about delayed communication in a more formal manner. Each input and output of a function block can be viewed as a *sequence* of values in some domain, \( n \). Then, for a given domain of values, \( V_n \), the sequence associated with it is a function, \( s_n \), defining the value at a particular discrete time instant, \( t \), such that \( s_n(t) = v_n \), where \( t \) is some non-negative integer and \( v_n \in V_n \). By stating that send and receive operations between function blocks must always occur with a unit delay, the assignments from a given output sequence, \( o \), to a given input sequence, \( i \), will effectively take the form of,

\[
i(t) = o(t - 1). \tag{5.1}
\]

Equation 5.1 ensures the “pipelining” of the send and receive operations between function blocks. Furthermore, since all communication between function blocks in a network are delayed to the next *tick*, they can be arbitrarily scheduled, while still ensuring an overall deterministic behaviour. This will be illustrated using the function block network in Figure 5.1a. Each \( EIX \) and \( EOY \) ports in that figure can be viewed as input and output sequences, respectively. If we assume that \( EOY \) is emitted at instant \( k-1 \), then, by using equation 5.1, we obtain the input sequence
assignments at $t=k$, as shown in Figure 5.1b. Notice that this ensures causality as current outputs can only be affected by other outputs produced in the previous instant, but never by those produced in the same instant. Moreover, the behaviour of the network will not be affected even if the order of function block execution is altered, since the input assignments to every block is already known at the start of each instant (lines 5–8 can be arbitrarily scheduled).

A close observation of this example also shows that the requirement for delayed communication can actually be relaxed without resulting in non-causal programs so long as at least one event connection in every set of strongly connected function blocks implements a delayed communication. This notion of strongly connected blocks originates from graph theory. A set of vertices in a directed graph is said to be strongly connected if there exists a path from each vertex to every other vertex. Function block networks can be abstracted as directed graphs, by viewing each function block as a vertex and the event/data lines connecting outputs to inputs as directed paths. If a set of function blocks are not strongly connected, delayed communication is not needed at all, since the flow of events/data would be causal as such. In Figure 5.1a, FB1, FB2, and FB3 form a set of strongly connected function blocks. Thus, at least one pair of event connections among them must implement a delayed communication of the form given in equation 5.1 to guarantee that the program is causal. For function blocks that are not strongly connected (e.g. FB2, FB3, and FB4), input-output connections can take the form of, $i(t) = o(t)$, without raising any causality issues.

Obviously, in this case, the compiler that synthesizes the function block program would need to check for the existence of strongly connected blocks. The execution order would also need to be analysed at compile-time, as function blocks can no
longer be arbitrarily scheduled. However, this can be done by abstracting the net-
work as a directed graph, and scheduling each vertex through a topological sort. 
This technique can be separately applied to each composite function block without 
needing to recurse into constituent blocks, thus, allowing function blocks to be still compiled modularly. Significantly also, the checking for strongly connected blocks is much easier and faster than the causality analysis performed by Esterel compilers. As such, the direct compilation of synchronous function blocks to C can be accomplished more cheaply than a translation via Esterel.

5.1 Effects on scheduling order and communication

To qualitatively compare the effects of delayed and instantaneous communication in the synchronous approach with the other execution models mentioned in Section 2.2, the various behaviours for a given function block network are discussed using Figure 5.2a. Let us assume that the emission of EO1 from FB1 results in corresponding emissions of EO2, EO3, and EO5 from FB2, FB3, and FB5 respectively. The depth-first event propagation scheme [12] will result in the invocation sequence of FB1, FB2, FB3, FB4, FB5, and FB4. FB4 will be invoked twice, first by FB3, and then, by FB5. In contrast, the breadth-first event propagation scheme [13] will schedule function blocks according to the sequence of emitted events, resulting in the execution order of FB1, FB2, FB5, FB3, FB4, and, again, FB4. This time, FB4 will be invoked in response, first, to the emission of FB5, and then, in response to FB3. Meanwhile, with the cyclic-scan scheme [52], the execution sequence in a scan cycle is simply left up to the application designer to decide (e.g., FB1, FB2, FB3, FB5, and FB4).

Obviously, each of these approaches will result in different application behaviours. It is worthwhile noting that these discrepancies arise due to the different function block execution orders. The synchronous approach with delayed communication immunizes against this phenomenon by producing the same behaviour irrespective of the execution order of function blocks. If function blocks are allowed to communicate instantaneously with one another, the order of execution will be determined through a topological sort of the blocks. Figures 5.2b and 5.2c contrast the pseudocodes for implementing delayed and instantaneous communications between function blocks, respectively. Again, as in Figure 5.1b, the order of function block execution in Figure 5.2b can be changed without affecting the overall behaviour of the program.

On the other hand, the scheduling order is more constrained when considering in-
5.2. CODE GENERATION FOR FUNCTION BLOCKS

5.2.1 Code generation for function blocks

The function block compiler, FBC (introduced in Chapter 4), is extended here to produce C code as well, besides Esterel. The Esterel code generated by FBC may

Figure 5.2: Illustration of the effects of execution order and communication: (a) shows a function block network that gives rise to different execution orders for various execution models; (b) and (c) contrast the effect of delayed and instantaneous communications in function block execution within a tick.

stantaneous communication. Figure 5.2c provides a possible execution order for the example in Figure 5.2a. Lines 6 and 7 can be arbitrarily re-ordered with lines 2–5 to still produce a valid topological sort. The prototype compiler developed (described in the following sections) will ensure that the execution order of the generated code will correctly adhere to the event/data dependencies implied in the network.

5.2 Code generation for function blocks

The function block compiler, FBC (introduced in Chapter 4), is extended here to produce C code as well, besides Esterel. The Esterel code generated by FBC may
already be synthesized to C using an Esterel compiler. However, the C code generator for fbc has been developed to obtain more efficient code than that produced by the Esterel compiler. This has been achieved by taking advantage of the semantic simplifications due to the communication model described previously. The technique used to allow instantaneous communication in the absence of strongly connected blocks can also be applied to the generation of Esterel code. This can be achieved simply by removing the pre keyword from the Esterel code whenever delayed communication is not needed. In this way, the behaviour of both the generated Esterel and C codes are ensured to be equivalent through a uniform underlying synchronous semantics. Practically, this equivalence is achieved by creating an intermediate representation for each compiled function block program, which is then used to generate both Esterel and C codes using a structural translation approach.

Like in the case of Esterel, C code synthesis in fbc begins by taking as input a function block program specified in the IEC 61499 XML format [18]. A depth-first traversal of the function block network is performed by recursively entering each composite block encountered. This results in a bottom-up compilation of every block in the network. This process will be repeated for each resource described in the input file.

Every function block type encountered by fbc will be synthesized as a separate C structure, whose members describe the event-data interface, as well as the local data (if any) of a function block. The events of a function block are densely encoded as bit fields in C for efficient memory usage. Each function block type also implements at least two functions, named FBTypeinit and FBTypeun. FBTypeinit serves as the constructor method for the corresponding function block type, since C does not have built-in support for object constructors common in object-oriented programming languages.

For basic function blocks, the FBTypeun function implements the execution code for the corresponding ECC. Algorithms within an ECC are implemented as separate C functions. Fbc allows these algorithms to receive additional parameters as function arguments in C. This simple, but useful, extension is currently unavailable in existing free and commercial IEC 61499 development tools [12–14]. For composite function blocks, the FBTypeun function implements a netlist that describes the interconnection of components within the composite block. The execution of component blocks within the composite function block is, in turn, invoked through their respective FBTypeun functions.

At present, a number of common service interface function blocks as prescribed in [5], such as the communication function blocks (e.g., PUBLISH and SUBSCRIBE)
5.3 Translating basic function blocks

The main task in the translation of the basic function block is similar to that described in Section 4.2. State and transition information need to be extracted from the ECC. This information is then used by the compiler to implement the ECC using a switch statement in C.

The translation relies on the same intermediate representation, consisting of a directed graph of SyncStates, as described in Section 4.2. A depth-first traversal is performed over the graph to label each SyncState with a corresponding state index. These indices are subsequently used for generating the case labels for the switch statement. The algorithm used for generating the code for a given basic function block is provided in Figure 5.3.

As can be seen in lines 5–8 of Figure 5.3, two variables are generated for each data port to represent the data value within the function block and to indicate the value at the interface, as explained in Section 4.2. At the start of the execution of a basic function block, all output events from the previous tick will be cleared (line 9). Then, code is generated to check for input events and to update the internal variables of the associated input data (lines 10–12). The loop in lines 13–31 then visits each state to construct the switch statement. Each state will result in a distinct case in the switch statement. Code will be generated there for each action...
procedure GenerateBFB(fb)

1. $S := \text{set of all states in } fb$;
2. $IE := \text{set of all input events in } fb$;
3. $OE := \text{set of all output events in } fb$;
4. $ID := \text{set of all interface input data in } fb$;
5. $ID' := \text{set of all internal input data in } fb$;
6. $OD := \text{set of all interface output data in } fb$;
7. $OD' := \text{set of all internal output data in } fb$;
8. generate code to clear all output events in $OE$;
9. $\text{foreach } ie \in IE \text{ associated with } id \in ID \text{ do}$
10. generate code to update $id'$ with $id$ whenever $ie$ occurs;
11. end
12. $\text{foreach } s \in S \text{ do}$
13. $\text{generate new case for } s \text{ in switch-statement;}
14. $\text{foreach action, } a, \text{ of } s \text{ do}$
15. $\text{if } a \text{ has algorithm, } alg \text{ then}$
16. $\text{generate call to function[alg];}$
17. $\text{end}$
18. $\text{if } a \text{ has oe } \in OE \text{ then}$
19. $\text{generate code to set oe;}$
20. $\text{end}$
21. $\text{end}$
22. $\text{foreach transition condition, } t, \text{ of } s \text{ do}$
23. $\text{if } t \text{ leads to next state } n \in S \text{ then}$
24. $\text{generate code to test for } t \text{ and assign next state to } n;$
25. $\text{end}$
26. $\text{if } t \text{ is always true then}$
27. $\text{break;}$
28. $\text{end}$
29. $\text{end}$
30. $\text{generate code to clear all input events in } IE;$
31. $\text{foreach oe } \in OE \text{ associated with } od \in OD \text{ do}$
32. $\text{generate code to update } od \text{ with } od' \text{ whenever oe occurs;}$
33. $\text{end}$
34. $\text{end}$
35. $\text{end procedure}$

Figure 5.3: Algorithm to generate C code for a basic function block.
5.4 Translating composite function blocks

Fbc compiles the network of function blocks within a composite block into a netlist. To accomplish this, instances of the component blocks are encapsulated within the composite function block’s structure in C, in addition to the members for its event-data interface. The FBTypeinit function of a composite function block will invoke the respective FBTypeinit functions of its component blocks to correctly initialize the entire network during instantiation.

As described in the beginning of this chapter, Fbc ensures the causality of function block compositions in a network by either delaying all communications between blocks, or by ensuring that event-data connections between blocks do not form a set of strongly connected components. The choice of either alternative may be selected through a compiler switch when invoking Fbc, and will result in different types of code being generated. The technique used for generating code for implementing the delayed communication will be described first, before proceeding to do the same for the instantaneous communication model.

5.4.1 Implementing delayed communication

The algorithm for constructing a netlist from a function block network is given in Figure 5.5. As in Figure 5.3, the event and data associations at a composite function block’s interface must again be accounted for. This is done on lines 2–4 and lines 30–32 for the input and output interfaces respectively. However, the input and output events need not be explicitly cleared as previously done, since the netlist execution would already ensure the correct assignment for each input/output port.
```c
switch (me->_state) {
    case 0: // NORMAL state
        ...
        if (me->_input.event.cruiseOn) {
            me->_state = 1; // CRUISE state
            ...
        } else if (me->_input.event.accelPressed) {
            me->_state = 0;
            ...
        } else if (me->_input.event.accelReleased) {
            me->_state = 0;
            ...
        } else if (me->_input.event.clock) {
            me->_state = 0;
            ...
        } else if (me->_input.event.cruiseOff) {
            me->_state = 0; // NORMAL state
            ...
        } else if (me->_input.event.accelPressed) {
            me->_state = 2; // ACCEL state
            ...
        } else if (me->_input.event.cruiseOn) {
            me->_state = 1;
            ...
        } else if (me->_input.event.speedSet) {
            me->_state = 1;
            ...
        } else if (me->_input.event.clock) {
            me->_state = 1;
            ...
        }
        break;
    case 1: // CRUISE state
        ...
        if (me->_input.event.cruiseOff) {
            me->_state = 0; // NORMAL state
            ...
        } else if (me->_input.event.accelPressed) {
            me->_state = 2; // ACCEL state
            ...
        } else if (me->_input.event.cruiseOn) {
            me->_state = 1;
            ...
        } else if (me->_input.event.speedSet) {
            me->_state = 1;
            ...
        } else if (me->_input.event.clock) {
            me->_state = 1;
            ...
        }
        break;
    ...
}
```

Figure 5.4: Code snippet illustrating the generated code for the ECC in Figure 1.3.
procedure GenerateCFB(fb)
  foreach ie \in IE \text{ associated with } id \in ID \text{ do}
    generate code to update id' with id whenever ie occurs;
  end
  P := set of all input ports in the network of fb;
  foreach p \in P \text{ do}
    I := p.connectionSet();
    if p.type = EVENT then
      foreach i \in I \text{ do}
        generate code to assign p to event, i;
      end
    else if p.type = DATA then
      generate code to assign p to data, i \in I;
    end
  end
  foreach component block, b, in the network of fb \text{ do}
    make call to execute b;
  end
  Q := set of output ports at the interface of fb;
  foreach q \in Q \text{ do}
    O := q.connectionSet();
    if q.type = EVENT then
      foreach o \in O \text{ do}
        generate code to assign q to output event, o;
      end
    else if q.type = DATA then
      generate code to assign q to output data, o \in O;
    end
  end
  foreach oe \in OE \text{ associated with } od \in OD \text{ do}
    generate code to update od with od' whenever oe occurs;
  end
end procedure

Figure 5.5: Algorithm to construct a netlist for a function block network.
Due to the delayed communication semantics, the compilation process is greatly simplified. This is because the execution of component blocks can be arbitrarily scheduled with no behavioural difference in the resulting code. The “pipeline” for the send and receive operations requires a double-buffering scheme to hold inputs/outputs for the previous and current ticks. However, the implementation here avoids double-buffering in each function block; every block has only a single buffer for each of its inputs and outputs. Instead, the output buffer of the sending block is always used by the receiving block(s) as an input buffer from the previous tick. This avoids the need for explicit queues for communication between blocks, thus, keeping the implementation compact. Using this technique, the netlist construction can proceed in a straightforward manner, with code generated,

- first, to update the inputs of each component block (lines 6–15);
- then, to execute each component block (lines 16–18); and,
- finally, to update the outputs at the composite function block’s interface (lines 20–29).

The connectionSet() method on lines 7 and 21 returns the set of connections to a given port. Multiple connections may be made to a single event port, but only one connection can be made to a data port. This distinction, plus the possibility of multiple events occurring simultaneously, requires the code for event connections and data connections to be handled differently. These are handled for the input ports of all the component blocks in the network in lines 8–14, as well as for the output ports at the composite function block’s interface in lines 22–28.

### 5.4.2 Implementing instantaneous communication

To ensure causal compositions of function blocks while allowing them to communicate instantaneously, it is sufficient to ensure that the blocks in a network are not strongly connected with one another. If a set of strongly connected blocks is detected, FBC may be configured using a compiler switch to either flag the cycle to the user, or to arbitrarily break the cycle on its own. Figure 5.6 shows the algorithm for doing this, using a modified topological sort.

Every function block in a given network will be iteratively passed to the EnsureAcyclic function, together with an empty stack, S, as arguments. If a function block, fb, is found to be already on the stack, it means that a set of strongly connected blocks have been detected in the network. If configured to do so, FBC will
5.4. TRANSLATING COMPOSITE FUNCTION BLOCKS

```plaintext
1 function EnsureAcyclic(fb, S)
2    if fb is in S then
3        if configured to break cycle then
4            arbitrarily insert unit delay here;
5            return true;
6        else
7            exit with error: cycle detected;
8    end
9 end
10 if fb has not been visited then
11    add fb to visited set;
12    push fb onto S;
13    foreach predecessor, n, of fb do
14        if EnsureAcyclic(n, S) = true then
15            break;
16    end
17    pop fb from S;
18    insert fb into sorted list, L;
19 end
20 return false;
21 end function
```

Figure 5.6: Algorithm to topologically sort function blocks in a network and to ensure that they are not strongly connected.

arbitrarily insert a unit delay for the communication between fb and its predecessor in order to break the cycle of strongly connected blocks. Otherwise, fbc will simply fail with an error message reporting that a cycle has been detected, and the user can manually insert a unit delay at the appropriate point. This is done on lines 2–9.

Lines 10–20 implement the main sorting algorithm. Each unvisited predecessor of fb is recursively visited in order to produce a sorted list, L. If the recursion returns true, it means that a unit delay has been inserted to break a cycle, and the topological order for fb has been determined. Otherwise, the sorting algorithm will continue the recursion to exhaustively order all predecessors of fb before placing fb into L. Once this is done, a similar algorithm to that in Figure 5.5 is called to generate a netlist, but this time, the netlist construction will be sorted according to the order implied by L.

This same algorithm has also been used to augment the Esterel code generator with the ability to produce code for instantaneous communication, while still ensuring causality. In this case, the predicate of each await statement is simply generated without the pre keyword, except at those points where a unit delay has
been inserted. This ensures that fbc can always be made to generate semantically equivalent Esterel and C code from a given function block program.

While the translation for a function block network has thus far been discussed in the context of the composite function block, a similar procedure also applies for a function block network in a resource. In this case, however, the event-data associations done on lines 2–4 and lines 30–32 of Figure 5.5 are no longer necessary, as they are not applicable to resources. In IEC 61499, a resource sits at the topmost position in the hierarchy of function block networks, and it cannot be further nested within another resource. Nevertheless, from a code generation point of view, it can be treated quite similarly as any other function block network.

5.5 Generating the graphical function block simulator

Fbc is also capable of automatically generating a simulator when compiling a given function block description. This option can be easily selected by passing an additional argument to fbc during the compilation process. The simulator provides a graphical interface for the user to interactively enter different input scenarios into a given function block and to observe its corresponding outputs. This enables the user to modularly test different parts of the system for logical correctness very rapidly throughout the entire design process.

To generate the simulator, fbc links the C code synthesized from the function block description together with the code for the graphical interface. These are then automatically compiled in the background to create a single executable file. The graphical interface is based on the Qt UI framework [74], which enables the function block simulator generated by fbc to run on either Windows, Mac, or Linux machines.

As an example, the graphical simulator generated for the Throttle function block of Figure 1.2 is shown in Figure 5.7. The items within the “Inputs” and “Outputs” columns in Figure 5.7 correspond directly to the inputs and outputs of the Throttle function block. A user may enter an input scenario using the check-boxes and text-boxes in the “Inputs” column. The “Outputs” column, however, is read-only, and will display the corresponding output for a given input whenever the “Tick” button is pressed.
5.6 Discussion

The synchronous execution of function blocks described in this chapter is reminiscent of the cyclic-scan model proposed by Lastra et al. [52], and used in the commercial tool, ISaGRAF [14]. Similar to them, the code generated by FBC also results in a cyclic executive. However, unlike the code from ISaGRAF, the code from FBC can run on its own without needing a run-time environment. Moreover, the synchronous execution model does not suffer from the behavioural variations arising from different function block execution orders within a cycle, as in [20, 52].

In contrast to the event-triggered model used in [12, 13], the execution order of function blocks is statically scheduled by FBC, instead of dynamically scheduled by a run-time environment. Since all scheduling decisions are determined at compile-time, the resulting code from FBC is fully predictable and can run independently. This not only avoids the issue of portability among function block run-time environments, but also lends itself to the efficient implementation of function block programs.

The use of delayed communication between strongly connected function blocks only is reminiscent of the synchronous model considered in [75]. The synchronous model there consists of a set of communicating Mealy machines. Since the synchronous composition of Mealy machines may not always be well-defined due to dependency cycles, every such cycle is assumed to be broken by a “unit delay” element. The synchronous model described in [75] was used as an abstract model to
show how synchronous systems can be deployed on a loosely time-triggered architecture. However, the application of delayed communication to strongly connected function blocks in this chapter was introduced to increase the expressivity of function block descriptions, while still

1. enabling the separate compilation, and,

2. ensuring the reactivity and determinism of function block programs.

With the proposed translation technique to Esterel and C based on the synchronous approach, it is further envisaged that safety properties of function block programs can first be verified using existing verification tools for Esterel [67], like done in the previous chapter. Once the Eseterel code for a given program has been verified, efficient code can then be automatically generated from the original function block description using the C code generator of FBC.
The previous chapters have presented a synchronous model for function blocks that not only defines the behaviour of function block compositions, but also ensures the reactivity and determinism of final implementations. Implementations are also more efficient, as code synthesized from synchronous function blocks can run without the need of any run-time environment, unlike existing techniques (see Chapter 7 for a quantitative comparison between the various approaches). In fact, it is already widely-known that one of the strengths of the synchronous approach lies in its ability to produce very efficient object code for concurrent specifications running on a centralized system [63].

The IEC 61499 standard, however, is intended for the development of distributed control systems [5]. The synchronous execution model is not an ideal match for this requirement, as synchronous solutions are known to be costly and inefficient for distributed implementations [27]. Asynchronous models are more natural for distributed systems, but greater effort is often required to ensure that the overall system behaves as desired. This chapter, thus, extends the synchronous model described so far to a globally asynchronous locally synchronous (GALS) [23] model. Using this, distributed function block systems may be viewed as a collection of synchronous compositions of blocks, which communicate with each other over an asynchronous network. This offers a good compromise between the specification and implementation spaces.

In order to elucidate this GALS model, this chapter will first begin by introducing some preliminary concepts pertaining to GALS systems. Some background information regarding communication function blocks in IEC 61499 will also be presented. These are dedicated function blocks intended for describing communication in a dis-
tributed system. The use of the GALS model within the context of distributed IEC 61499 systems will then be illustrated through an example.

Following that, this chapter will describe the use of abstract communication patterns to describe communications in a GALS system. Two communication patterns are given as examples. These patterns enable different communication semantics to be expressed with various trade-offs in the reliability and efficiency of the implementation. This chapter then shows how the communication function blocks prescribed by the IEC 61499 may be used to encapsulate these abstract communication patterns. A technique to automate the generation of code from these communication function blocks is then presented through an extension of FBC.

6.1 GALS model for distributed systems

Distributed control systems consist of a set of concurrent and interacting components. Concurrency in distributed control systems arises naturally from two sources:

1. *Parallelism in the controlled environment.* Quite often, the environment exhibits several degrees of freedom that need to be controlled in parallel. For example, in the cruise control system in Figure 1.1, the vehicle’s speed, brake pedal, and the various buttons of the system must be simultaneously monitored while regulating the throttle position at the same time.

2. *Distribution of the control system.* Distributed control systems are often times necessitated by the physical distance of sensors and actuators, or by the use of redundant controllers to provide fault-tolerance. The necessity for multiple controllers for implementation makes concurrency inherent to such systems.

In practice, these two kinds of concurrency need to fulfil somewhat different requirements. In the first instance, the parallel control of different parameters in the environment often requires a disciplined synchronization at the program level in order to read inputs and to compute outputs coherently. Programs of this nature closely match the differential equations used in discrete-time dynamical systems, and are most naturally specified through the synchronous programming approach. Due to the disciplined synchronization used, the parallelism in synchronous programs is more efficiently compiled rather than emulated. The parallelism, in this case, serves more as a logical means to compose a program, rather than as a literal specification for executing a program.
In the second instance, the use of multiple controllers leads invariably to distributed computations, which implies *literal* parallel execution on each node. In place of periodic synchronizations, communication between nodes takes place only when necessary. Subprograms on each node are most naturally thought of as running at their own pace, and communicating with each other asynchronously. These observations on concurrency provide a natural motivation to adopt a globally asynchronous locally synchronous (GALS) model for distributed IEC 61499 systems.

With the GALS model, each IEC 61499 resource can be considered as a synchronous island containing a network of function blocks that are composed in synchronous parallel, as described in earlier chapters. The IEC 61499 defines a resource as a functional unit of software, contained in a device, which has independent control of its operation [5]. It thus provides a natural boundary for synchronous islands within a GALS model. The local behaviour of each resource is synchronous (as seen from its own perspective), but the global behaviour is asynchronous (as seen from outside the system), since each resource samples its own inputs, computes its respective reactions, and communicates with each other in an undetermined, but finite amount of time. This GALS model is well-suited for network-based distributed systems, where communication delays cannot be ignored, and for heterogeneous deployments, where the relative speed of each computational node may vary substantially.

The appropriate characteristics of the communication, however, will depend greatly on the type of application being designed. Hence, it is advantageous to decouple the behaviour of each resource from the communication mechanism between them, and to allow for different kinds of communication to be chosen. This separation is also useful, as it will enable both the communication and functional specification of a distributed program to be designed and refined independently from each other.

This approach facilitates the treatment of communication at a high level of abstraction. Ideally, a designer should be able to specify communication patterns in an abstract way that does not yet imply any particular implementation. For example, there should be a way to specify the lossless reception of all information sent as a semantic property of a particular communication. The actual protocols or buffer size to use, and so on, are simply implementation choices that realise the desired semantic property. One example of an abstraction that guarantees lossless communication is Kahn Process Networks (KPN) [76]. It does so by assuming an ideal buffering scheme that is unbounded in size. Such abstractions must obviously be refined to finite buffer sizes, while still ensuring no loss in the concrete implementation.
In the IEC 61499 standard, *communication function blocks* are used for describing various communication patterns. Communication function blocks are actually specific service interface function blocks that prescribe generic interfaces for communication. Different types of communication function blocks may be used to specify various properties for communication, while keeping the underlying mechanisms hidden from the application. Communication function blocks, thus, provide a natural means for specifying abstract communication patterns in IEC 61499.

The standard prescribes two types of communication function blocks, that is, the *client-server* and the *publish-subscribe* function blocks. The client-server pair describes *point-to-point bidirectional* communication, while the publish-subscribe pair describes *one-to-many unidirectional* communication. Both these communication patterns, like KPN, assume the use of unbounded queues for communication. This assumption, however, is somewhat tenuous for control software, as they are often deployed on resource-constrained devices. Unbounded queues, thus, need to be refined into finite ones, which require either some synchronization between the sender and receiver, or adequate buffer space to avoid or bound losses to an acceptable level. This refinement from abstract communication patterns to concrete communication code will be demonstrated through these communication function blocks.

The next subsection will provide an example of a distributed IEC 61499 system that makes use of these communication function blocks.

### 6.1.1 An example of a distributed system

Figure 6.1 depicts a water level control system. A proportional-integral-derivative (PID) controller is used to control the level of water in a tank, which is measured as the process variable (PV) by a level transducer. The set-point (SP) for the PID controller is obtained from a faceplate control, which also displays the PV and the output value (OUT). This output value indicates the position of a valve, which determines the rate of flow of water into the tank. The purpose of this setup is to maintain the water level in the presence of exogenous changes in the rate of drainage by a rotary pump.

Figure 6.2 shows a distributed function block program for the water level control system. The faceplate control, the water level sensor, and the PID controller are modelled as the `FACEPLATE1`, `TANKSENSOR`, and `FB_PID` function blocks respectively. These are implemented in three separate resources, namely, the `CONSOLE`, `TANKMODEL`, and `CONTROL` resources. Each of them has been allocated to a separate device for execution. The `E_RESTART` block in the function block network of
each resource is a special service interface function block that is used normatively in IEC 61499 for starting/stopping resources. The rest of the blocks in Figure 6.2 consists of the client-server and publish-subscribe communication function blocks.

The dotted arrows in the figure indicate the various data exchanges that take place between the resources via the communication function blocks. The water level from the TANKSENSOR block is published to the CONSOLE and CONTROL resources via the PUBLISH function block. This value is received by both the SUBSCRIBE function blocks, and relayed to the PV input port of the FACEPLATE1 and FB_PID blocks, respectively, via the RD output port. The CONSOLE resource, on the other hand, sends the user-specified mode and set-point for the PID to the CONTROL resource via the CLIENT block. This is received by the SERVER block and relayed to the FB_PID block. In turn, the FB_PID block computes the OUT value and sends it back to the CONSOLE resource to be displayed on the faceplate via the CLIENT block. While both the client and server blocks can be used for sending and receiving data, this example has restricted their use to just one and not the other. This has been done to make the illustration of the flow of data clearer.

The example here has shown how communication between distributed resources may be accomplished via the communication function blocks of the IEC 61499. The next section will go on to formalize some of the intuitive notions of communication that have been introduced thus far.
Figure 6.2: Distributed implementation of the water level control system on the CONSOLE, TANKMODEL, and CONTROL resources. Each of these resources is contained in a separate device. The dotted arrows show the communication between the resources via the CLIENT-SERVER and PUBLISH-SUBSCRIBE function blocks.
6.2 Communication

Communication function blocks provide generic interfaces for communication in IEC 61499. This facilitates easier reuse by providing an abstract means to describe communication. However, in the standard, communication function blocks are more of an architectural feature that prescribes the normative syntax, rather than the semantics, for communication. In fact, the standard specifically allows for implementation-dependent customizations of communication function blocks. This is useful to cater to the variety of platforms for which IEC 61499 systems may be implemented on, but it leaves the designer poorly equipped to formally reason about the properties of communication. Hence, this section will formalize some properties of communication, and discuss certain trade-offs that can be made by the designer.

The presentation here will, thus, begin by introducing some preliminary concepts regarding communication.

### 6.2.1 Connections and channels

For distinct components in a system to communicate with each other, there must exist a connection between the output port of one component (called the sender), and the input port of another component (called the receiver). Following [59], system components are modelled here as processes, whose set of behaviours, $F$, is defined by a mapping from a set of input signals, $I$, to a set of output signals, $O$, i.e., $F : I \mapsto O$. The sender and the receiver processes can, thus, be modelled by the functions, $F_s : I_s \mapsto O_s$ and $F_r : I_r \mapsto O_r$, respectively. The set of behaviours of the connected components is then the composition of these functions, $F_r \circ F_s$.

An ideal connection is an identity function, $F_d$, that maps the value of an output port, $o_s \in O_s$, from a particular domain to the input port, $i_r \in I_r$, of the same domain:

$$o_s = F_d(o_s) = i_r$$

An ideal connection imposes an equality of values between the input port of the receiver and the output port of the sender.

Connections between components are implemented using channels, which are taken here to include both the hardware network interface, as well as the software protocols used to realise a connection. A channel can be modelled as a function, $F_c$, that mimics the identity function, $F_d$. In general, however, a channel is only an approximation of the ideal connection, as discrepancies between the values of the connected input and output ports may arise due to communication errors. Figure 6.3
illustrates this difference.

The quality of service (QoS) requirements of a particular communication may include criteria, such as throughput, delay, and the rate of packet losses. As in [59], we use QoS requirements to partition the set of behaviours, $F$, into two classes: the class that satisfies the requirements, and the class that does not. A QoS-equivalence relation (denoted by $\sim$) is defined as follows:

$$\sim = \{(f, f') | f \in F, f' \in F\},$$

where both $f$ and $f'$ satisfy the QoS requirements. In this case, the processes $f$ and $f'$ are said to be QoS-equivalent, i.e., $f \sim f'$.

For a given connection and a set of requirements for the quality of the received signal, valid channels for implementing the connection are those that can satisfy the QoS requirements. For example, if no loss is a QoS requirement for the communication, then a valid channel is one that satisfies the identity function, such that, $F_r \circ F_s \circ F_s \sim F_r \circ F_s$.

In light of this, it is helpful to characterise lossyness as a semantic property of communications. This enables communication patterns to be used for specifying in an abstract way whether a given communication ought to be lossy or lossless. The choice between lossy and lossless communication is dependent on the application. If lossy communication is chosen, the maximum rate of loss acceptable must also be decided, with each of these decisions resulting in trade-offs in the implementation space. In communications, data may be lost due to:

1. the poor quality of the physical channel (e.g., due to noise or interference), or
2. the resource constraints in the implementation (e.g., the receiver consumes data slower than the rate of production but does not have sufficient buffers to store the incoming data).
6.2. COMMUNICATION

The first problem is a transport layer concern that can be overcome by using robust protocols to (probabilistically) ensure the correctness of received data by means of re-transmission or coding techniques for error correction. The second problem, however, can be dealt with by addressing the semantics of communication in the application layer. It is this second aspect that is being treated here in this section.

One common approach to allow senders and receivers of different speeds to communicate with each other is to assume the use of an unbounded queue in the communication semantics [76]. Unbounded queues can match any disparity in the rate of production and consumption of data, as it allows an unlimited amount of data to be buffered. However, actual implementations are accomplished using channels with finite queues. Hence, realistic solutions must either rely on additional protocols to block the sender when the receiver’s queue is full, or to simply accept the possibility of a certain amount of loss.

The following subsection will, thus, propose two abstract communication patterns that can be used to describe both these options. These patterns will be used later to refine the client-server and publish-subscribe communication function blocks to bounded lossless and lossy channels, respectively.

6.2.2 Bounded lossless and lossy channels

In order to derive actual communication channels, the size of the buffer used for communication must be finite. For this discussion, let us consider a buffer of size \(q\), where \(q\) is some positive integer.

To obtain lossless communication with a bounded buffer, the sender needs to refrain from sending data whenever the receiver’s buffer is already full. This may be accomplished in several ways:

1. Have the sender check whether or not the receiver’s buffer is full before sending.
2. Use a synchronization protocol to ensure that the sender will never send at a rate that is faster than what can be consumed and buffered at the receiving end.

The first method is simple, yet costly, as it incurs an additional query to the receiver each time before attempting a send operation. Hence, the second method is adopted here. This protocol can be implemented using a send-acknowledge handshake to ensure that the queue never overflows. For this, the sender needs to keep a local count of the number of free slots remaining in the receiver’s queue. At first, this count is initialized to \(q\), which is the buffer size as mentioned above. Each
time the sender sends a message to the receiver, this count will be decremented. Whenever the receiver reads from its queue, an acknowledgement will be returned to the sender, causing the sender to increment its count. If at any time this count reaches zero (indicating that the receiver’s queue is full), the sender will block until it receives another acknowledgement from the receiver.

This protocol is illustrated in Figure 6.4a. \( R_1 \) and \( R_2 \) are two processes that communicate with each other. \( R_1 \) sends data to \( R_2 \) via queue \( Q_1 \), while \( R_2 \) sends data to \( R_1 \) via queue \( Q_2 \). This approach is actually reminiscent of Kahn Process Networks (KPN) [76], except that it uses bounded instead of unbounded queues. KPN has the desirable property that it will never deadlock, provided that processes do not all simultaneously perform read operations on an empty queue.

Processes communicating with a finite buffer using the suggested protocol will also never deadlock under the same condition. In addition, the processes should also not attempt to write to a full queue all at the same time. Furthermore, to prevent dependency cycles among a set of processes connected in a loop (as in Figure 6.4a) from immediately deadlocking on start-up, one of the queues in the cycle can be defined to contain an initial message. These constraints then lead to the following theorem:

**Theorem 3.** Communication between every pair of communicating processes will never deadlock using the send-acknowledge protocol and a finite queue of size \( q \), where \( q \) is some positive integer, with \( q > 1 \) for all queues containing an initial message, and \( q > 0 \) otherwise.

A proof sketch is provided below using marked directed graphs (MDG), a subclass of Petri Nets [39], as done in [75]. Figure 6.4b shows the MDG [77] corresponding
to the abstracted communication of Figure 6.4a. An MDG is a bipartite graph consisting of two kinds of nodes, called \textit{places} and \textit{transitions}, where arcs are either from a place to a transition or from a transition to a place. Places are denoted by circles, while transitions are denoted by rectangular bars. Arcs are labelled with their weights (positive integers), where a $k$-weighted arc can be interpreted as the set of $k$ parallel arcs. Labels for unity weight are usually omitted. Each place in an MDG has exactly one input (preceding) transition and exactly one output (succeeding) transition. Each place may also be assigned with a positive integer, $k$. If a place is so assigned, that place is said to be \textit{marked with $k$ tokens}. This is visually depicted by marking the place with $k$ dots (tokens). Loops that form around a set of nodes connected by arcs are referred to as \textit{directed circuits}. An example of a directed circuit is outlined in dotted lines in Figure 6.4b.

Formally, an MDG is expressed as a quintuple, i.e., $MDG = (P, T, F, W, M_0)$, where:

- $P$ is the set of places;
- $T$ is the set of transitions;
- $F \subseteq (P \times T) \cup (T \times P)$ is the set of arcs (flow relation);
- $W : F \mapsto \{1, 2, 3, \ldots\}$ is the weight function; and,
- $M_0 : P \mapsto \{0, 1, 2, \ldots\}$ is the initial marking.

To understand the proof sketch below, only the basic rules for transition enabling and firing in MDGs need to be known:

1. A transition, $t$, is said to be enabled if each input place, $p$, of $t$ is marked with at least $w(p, t)$ tokens, where $w(p, t)$ is the weight of the arc from $p$ to $t$.

2. An enabled transition may or may not fire, depending on whether or not the event represented by the transition actually takes place.

3. A firing of an enabled transition, $t$, removes $w(p, t)$ tokens from each input place, $p$, of $t$, and adds $w(t, p_o)$ tokens to each output place, $p_o$, of $t$, where $w(t, p_o)$ is the weight of the arc from $t$ to $p_o$.

\textit{Proof.} In Figure 6.4b, each process, $R_i$, from Figure 6.4a is mapped to an MDG transition, $t_i$. Meanwhile, every queue from $R_i$ to another process, $R_j$, is mapped to two places—a forward place, $p_{i,j}$, with input transition $t_i$ and output transition
Figure 6.5: Illustration of the liveness property of an MDG: (a) shows the initial marking, and (b) shows the resultant marking after $t_1$ fires.

$t_j$, as well as a backward place, $p'_{i,j}$, with input transition $t_j$ and output transition $t_i$. For a queue of size $q$, $p'_{i,j}$ will be initially marked with $q$ tokens, while $p_{i,j}$ will have no tokens. This initial marking for every backward place models the size of the queue and ensures that a process will never write to a full queue. If a queue contains an initial message, $p_{i,j}$ will be marked with one token, while $q - 1$ tokens will be placed in $p'_{i,j}$. The placement of an initial token in the forward place models the fact that dependency cycles among processes connected in a loop can be resolved using an initial message at start-up. Whenever $R_i$ sends a message ($t_i$ fires), a token will be removed from $p'_{i,j}$ and placed into $p_{i,j}$. The queue from $R_i$ to $R_j$ is full if and only if $p'_{i,j}$ has no more tokens, in which case transition $t_i$ cannot fire ($R_i$ can no longer send). On the other hand, each time process $R_j$ reads a message ($t_j$ fires), a token is removed from $p_{i,j}$ and placed into $p'_{i,j}$, which indicates that the queue is non-full again. The send-acknowledge protocol described earlier can be thought of as an implementation of the firing rules of this constructed MDG.

It is a classic result from [77] that an MDG in which each directed circuit has a positive token count is live, meaning that it is possible for every transition in such an MDG to be fired, or to be made fireable eventually through some sequence of firings. This is indeed the case of the MDG obtained as described above, since by construction, every $p'_{i,j}$ would contain a positive count of tokens initially. The token count in the directed circuit formed by $p_{i,j}$ and $p'_{i,j}$ is clearly invariant, as $p'_{i,j} + p_{i,j} = q$ under any firing. This liveness property guarantees that the MDG and, thus, the use of the send-acknowledge protocol, will never result in a deadlock of the communicating processes regardless of the firing sequence taken.

To illustrate this liveness property, let us consider the case where the size, $q$, of $Q_1$ and $Q_2$ in Figure 6.4a is four and three, respectively. Since processes $R_1$ and
R_2 are connected in a loop, we define Q_2 to contain an initial message in order to avoid mutual dependencies from causing a deadlock at start-up. As such, p'_{1,2} is marked with four tokens, p_{1,2} with zero, p'_{2,1} with two, and p_{2,1} with one, as shown in Figure 6.5a. Since each input place of t_1 is marked with at least one token, t_1 is enabled. The firing of t_1 will result in one token being removed from p'_{1,2} and p_{2,1} respectively, and one token being added to p_{1,2} and p'_{2,1} respectively. This is illustrated in Figure 6.5b. The token count in both the directed circuits remains invariant, as p'_{1,2} + p_{1,2} = 4 and p'_{2,1} + p_{2,1} = 3 before and after the firing.

Unlike lossless channels, the implementation of lossy channels is less involving. It can be accomplished by using an overwriting circular buffer. Data will simply be overwritten if a cycle of the circular buffer is completed before any of the data in it can be read. This refinement of unbounded queues to circular buffers results in potentially lossy communication. For applications that can tolerate a certain amount of packet loss, such a refinement may be an acceptable trade-off to the overheads of implementing a synchronization protocol between the sender and the receiver.

In practice, this is quite often the case for communication involving continuous valued data, like those coming from sensors. In such cases, it is probably desirable for the sender to overwrite old data if the consumer is unable to momentarily keep up. Besides, it may not always be possible to synchronize the writing and reading pace of various nodes in a network. For example, in Figure 6.2, the TANKMODEL resource will simply publish the water level indicated by the TANKSENSOR function block, while the sensor will continue to produce periodic samples irrespective of the rate at which those samples are being processed. However, if the maximum burst rate of the sender is known and the average production rate is not greater than the average consumption rate, then it may still be possible to choose a suitable q for the size of the circular buffer that will avoid data from being overwritten.

### 6.3 IEC 61499 communication function blocks

Recall, from Section 6.1, that the IEC 61499 standard prescribes the client-server and publish-subscribe function blocks as two communication patterns that may be used in distributed control systems. These function blocks, however, assume the use of unbounded queues for communication. Therefore, they need to be refined into bounded channels in a concrete implementation, as described previously. The

---

1 Alternatively, Q_1 could have been defined to contain the initial message with the same effect.
Figure 6.6: Client-server communication function blocks. The client function block is configured to send two data elements, $SD_1$ and $SD_2$, and to receive one data element, $RD_1$. The server function block is configured to send one data element, $SD_1$, and to receive two data elements, $RD_1$ and $RD_2$.

following subsections will show how this can be done by refining the client-server and publish-subscribe function blocks to bounded lossless and lossy channels, respectively, as an example.

### 6.3.1 Client-server communication function blocks

The client-server function blocks are prescribed as generic communication function blocks for point-to-point bidirectional communication. They are normatively typed as pairs of $\text{CLIENT/SERVER}_{s,r}$, where $s$ and $r$ are non-negative integers that denote the number of data elements to be sent and received respectively. Figure 6.6 shows an example of a client-server pair, where the client is used to send two data elements to the server and to receive one data element from the server.

To implement the client-server function blocks with a bounded queue, they need to be refined in two ways:

1. **Syntactically**, the type naming convention of $\text{CLIENT/SERVER}_{s,r}$ is augmented with an additional index to become $\text{CLIENT/SERVER}_{s,r,q}$, where $q$ is a positive integer that denotes the length of the bounded queue. This simple extension gives the designer precise control over the amount of buffer space that is to be allocated for communication.

2. **Semantically**, the send-acknowledge protocol, as described in Section 6.2.2, can be adopted to ensure lossless communication. As an example, the $R_1$ and
The sequence diagrams in Figures 6.7 to 6.9 depict the protocol for the client and server blocks. As normative in the IEC 61499 standard [5], the suffix “+” is used in conjunction with an input/output event name to indicate that the value of the Q1/QO input/output is true at the occurrence of the associated event, while the suffix “-” is used to indicate otherwise.

For both the client and server blocks, the INIT input event is used to establish or disconnect a communication depending on the Boolean value of QI. The possible scenarios are depicted in Figures 6.7 and 6.8. The ID input provides the means to specify implementation-dependent parameters of the communication, such as the
Figure 6.8: Sequence diagram depicting server-initiated and client-initiated disconnections.

Figure 6.9: Sequence diagram depicting bidirectional data transfer between the client and the server.
6.3. IEC 61499 COMMUNICATION FUNCTION BLOCKS

Protocol or other network-related information. Data that is to be transmitted is placed on the SD\_n inputs, and will be sent whenever the REQ input event is received.

Meanwhile, the CNF output event, together with the Boolean value of QO, may be used to determine if a send operation has been blocked due to a full queue, or has otherwise completed successfully. If space is still available in the queue, the data will be sent over the network and will be available at the receiving end. The IND output event is emitted to indicate that data is ready to be read. Data that was previously received, but have yet to be retrieved from the queue, will be placed first on the RD\_n output ahead of the newly arrived data. Once the data on RD\_n has been read, the RSP input event can be used to indicate that the receiver has successfully processed the sent data. The sequence diagram for this exchange is depicted in Figure 6.9.

This refinement of the client-server function blocks has several appealing characteristics. A minimum queue length of one (two for queues containing an initial message) is sufficient for this approach to work faithfully. The use of larger queue lengths will not affect the semantics of communication, but will lower communication latencies by reducing the amount of time spent blocking. This reduction in latency is a trade-off that comes at the expense of higher memory usage.

6.3.2 Publish-subscribe communication function blocks

Besides the client-server, the IEC 61499 standard also prescribes the use of publish-subscribe communication function blocks. These are intended for unidirectional one-to-many communication. They are normatively typed as pairs of PUBLISH/subscribe\_d, where d is a non-negative integer that denotes the number of data elements to be published/subscribed. Figure 6.10 shows an example of a publisher and a subscriber, which are used to publish and subscribe to a single data element, respectively. The sequence diagrams depicting the communication between them are shown in Figures 6.11 to 6.13.

The meaning of the input/output ports for the publisher and subscriber blocks are similar as before. However, unlike the client-server communication pattern, where both the client and the server need to be directly aware of the other, publishers and subscribers do not need explicit knowledge about the existence of each other. Only the channel on which the “topic” is to be published, or subscribed from, needs to be known. The number and location of possible subscribers will, in general, be unknown—multiple consumers may subscribe to the same topic, or leave the subscription, without affecting the publisher. This flexibility allows for very dynamic
Figure 6.10: Publish-subscribe communication function blocks. The publisher function block is configured to publish a single data element, SD_1. The subscriber function block is configured to subscribe a single data element, RD_1.

Figure 6.11: Sequence diagram depicting the three scenarios for connection establishment between the publisher and the subscriber.

Figure 6.12: Sequence diagram depicting publisher-initiated and subscriber-initiated disconnections.
network configurations, as well as for redundant publishers and subscribers to be deployed for fault-tolerance purposes [78].

Like before, the type naming convention of publishers and subscribers may be augmented with an additional index to become $\text{PUBLISH/SUBSCRIBE}_d.q$, where $q$ is a positive integer that denotes the size of a bounded queue for communication. This time, the queue is implemented as an overwriting circular buffer, which results in lossy communication, as explained in Section 6.2.2. For the special case of a circular buffer of size one, the subscriber will simply always have the freshest data value.

In addition to avoiding the need of a synchronization protocol, the choice of a lossy implementation for the publish-subscribe communication pattern has one further advantage: it enables multicast links to be used for implementing the one-to-many communication. The need for synchronization between each sender and receiver would have effectively required the one-to-many communication to be implemented as separate point-to-point links. Such an implementation would not have been as efficient as the use of a single multicast link.

### 6.4 Implementation

The client-server and publish-subscribe communication patterns that have been described can be used to automatically produce distributed IEC 61499 systems. For this purpose, the FBC function block compiler has been extended to generate distributed code following the GALS model described in this chapter. The distributed code is a set of stand-alone executables that communicate with each other through
automatically synthesized implementations of the communication function blocks. This allows the distributed code to communicate with each other without the need for any middleware or run-time environment, unlike other function block implementations (e.g., [12] and [13]).

The process of generating distributed code with FBC begins by giving it as input a function block system specified in the IEC 61499 XML format [18]. Recall, from Chapter 1, that a system specifies a list of devices, with each consisting of one or more resources. FBC will generate a separate executable to run on each of these devices. For every resource in a device, FBC will do a depth-first traversal of the function block network in the resource, recursively entering each composite block it encounters, to perform a bottom-up compilation of every block in the network. The manner in which resources are executed will vary depending on whether they are deployed within a single device or on multiple devices:

- For resources executing within a single device, they will be executed in a simple round robin fashion. In each cycle of the round robin schedule, every resource in the device will perform its own computations for one clock tick.

- For resources executing on multiple devices, they will be executed at completely unrelated speeds. The relative execution speeds of the resources will depend on the computational dynamics of each device.

FBC relies on a library of implementation templates to automatically generate code for the communication function blocks described in Section 6.3. The use of a library to store implementation templates provides an easily extendible point in FBC to enable synthesis of other communication models using different transport protocols and media, like the commonly used fieldbuses [79] in distributed industrial systems. At present, FBC includes templates to implement the communication function blocks over Ethernet using POSIX sockets [73]. The following subsection will briefly describe how actual code is generated using these templates.

6.4.1 Synthesizing communication function blocks

During compilation, FBC visits every function block in a given network. For each basic and composite function block it encounters, FBC will create an intermediate representation of that block, which will include the internal details of either an ECC or an encapsulated network, as already described in Chapters 4 and 5.

Communication function blocks encountered during this visit, however, will be treated differently. Instead of creating an intermediate representation for their in-
6.4. IMPLEMENTATION

1 procedure PUBLISH
2 run
3 if input event INIT occurs then
4   read QI and ID;
5   if QI = true then
6     initialize socket with multicast IP, given by ID;
7     QO := true;
8     emit output event INITO;
9   else
10      close socket;
11      QO := false;
12      emit output event INITO;
13 end
14 else if input event REQ occurs then
15   read QI and SD;
16   if QI = true then
17     send SD data element through socket;
18     QO := true;
19     emit output event CNF;
20   else
21      QO := false;
22      emit output event CNF;
23 end
24 end procedure

Figure 6.14: Pseudocode for part of the C code generated for the publisher of Figure 6.10.

ternal details, FBC will check if the given communication function block exists in its library of templates. This involves parsing the function block type name to extract the type of communication function block, as well as the indices used for parameterizing the number of data elements to be sent/received and the size of the buffer. The templates themselves contain specially marked-up holders that can be customized appropriately with the extracted indices to generate concrete implementation code for each communication function block. An example of these templates are given in Appendix A for the publisher and subscriber blocks.

Figures 6.14 and 6.15 show the pseudocode for the publisher and subscriber function blocks respectively, with bold and italics font used to indicate the marked-up holders that have been replaced with the appropriate index. The pseudocode is quite easily understandable and should require little further explanation. It mainly implements the semantics for the various input/output ports of the communication
procedure SUBSCRIBE_J_8run
  clear all output events;
  if input event INIT occurs then
    read QI and ID;
    if QI = true then
      initialize socket with multicast IP, given by ID;
      initialize circular buffer, B, of size 8;
      QO := true;
      emit output event INITO;
    else
      close socket;
      QO := false;
      emit output event INITO;
  else
    if socket contains data then
      read 1 data element from socket and store in tail of B;
    end
    if circular buffer, B, is not empty then
      remove 1 data element from head of B and place on RD output(s);
      QO := true;
      emit output event IND;
      update head and tail of B;
    end
  end
  clear all input events;
end procedure

Figure 6.15: Pseudocode for part of the C code generated for the subscriber of Figure 6.10 with buffer size of 8.

function blocks, as described in Section 6.3.1. As the code generated by FBC is based on POSIX sockets, the ID input, which allows for implementation-specific parameters, is used here (line 6 in Figures 6.14 and 6.15) to specify the multicast address and port number on which the communication will take place.

The socket-based example described here is just one possible implementation for the publisher and subscriber function blocks. As can be seen, the use of communication function blocks provides a generic interface for communication, which allows for a variety of transmission media and protocols to be used in implementation. FBC can simply be extended with more communication function blocks templates to accommodate new requirements as they emerge.
6.5 Discussion

This chapter has presented an approach to model and synthesize distributed IEC 61499 systems as globally asynchronous locally synchronous (GALS) systems. Resources in IEC 61499 serve as synchronous islands that communicate asynchronously with each other through communication function blocks. These blocks can then be refined with specific queue sizes, and automatically synthesized with the desired communication protocols by selecting from a library of implementation templates. This has enabled different trade-offs in robustness, efficiency, and memory usage to be made. All these have been achieved without the need for any middleware or run-time environment, unlike existing IEC 61499 implementations.

Little work has been published, so far, along these lines in the context of IEC 61499. Available literature have mostly reported on the mechanics of implementation, rather than on the modelling and automatic synthesis of distributed systems. Beyond the domain of IEC 61499, however, the work presented in this chapter bears similarities with codesign finite state machines (CFSM) [58] used for describing and synthesizing embedded control software. CFSMs interact with each other via an asynchronous communication mechanism. The model of computation for a network of CFSMs is also GALS, but the communication between each CFSM uses a fixed single-place buffer for every event/data exchanged. Due to this, it is possible for events/data in CFSMs to get overwritten and lost. This is indeed similar to the case where resources communicate through a publish-subscribe pair that is configured with a circular buffer of size one.

The closest idea to the research undertaken here, however, is perhaps found in the work of Sangiovanni-Vincentelli et al. [59]. That work proposed an abstract representation for communication among processes using a variant of CFSMs, called abstract CFSMs (ACFSM). ACFSMs decouple the behaviour of each CFSM from the communication with other CFSMs, and assumes an infinite queue for communication. This enables different communication mechanisms to be specified in lieu of the single-place buffers of the original CFSM model. ACFSMs can then be refined into extended CFSMs (ECFSM), which are implementable. ECFSMs have input queues of finite size, and write operations can be either blocking or non-blocking on a channel-by-channel basis. However, that work did not describe how such an attempt can be automated for the variety of physical channels and protocols used in practical distributed systems. The work in this chapter, on the other hand, has shown how an industrial standard, like the IEC 61499, can be exploited to automatically synthesize various communication mechanisms through the use of an extendible
library of templates.

A more recent example of a GALS design language is SystemJ [80]. SystemJ programs consist of a set of clock-domains, where each clock-domain is a purely synchronous node consisting of one or more processes, running in lock-step with one another. Clock-domains, however, execute asynchronously with respect to each other, and communicate with one another through CSP-style [81] point-to-point unidirectional rendezvous. Clock-domains in SystemJ are, thus, analogous to the resource artefact described in this chapter. However, while resources in IEC 61499 may communicate with each other via a variety of communication function blocks, rendezvous are the only means of communication between clock-domains in SystemJ.
The approach and techniques proposed in this thesis for implementing IEC 61499 function blocks have, so far, been discussed in a qualitative manner. This chapter will present a quantitative analysis of what has been achieved in this research.

For this purpose, a practical industrial control system has been developed as a case study. The techniques described in the earlier chapters have been applied to the design of an airport baggage handling system (BHS) [36]. Though targeted at baggage handling, this exercise demonstrates the feasibility of the proposed approach for designing material handling systems in general. Material handling is a common feature of many industrial settings. Examples range from automotive production lines to meat packaging facilities.

The design framework created in this case study enables the development of an airport BHS using the model-view-controller (MVC) [82] paradigm. This approach enables the simultaneous development and testing of the model of the BHS plant and controller using model-accurate visualizations of the physical bags and the conveyor system. The plant model, controller, and visualization components run as a distributed system over a communication network to simulate the final deployment scenario. This methodology may be easily adapted for various control systems, which are oftentimes configured as closed-loop systems, consisting of both plant and controller as well. Thus, this case study presents a realistic application for evaluating the practicality of the proposed approach.

This chapter will also present experiments conducted over a suite of benchmark programs. These experiments, together with the case study, are used to compare the code produced by \texttt{fbc} with that from other function block code generators. The results obtained demonstrate the superiority of the code produced by \texttt{fbc} with respect to other existing tools, both in terms of execution speed and code size. This, in particular, indicates the suitability of the proposed approach, even for resource-
constrained applications.

The next section will proceed by providing details of the baggage handling system used in the case study.

7.1 Modelling a baggage handling system

Existing approaches for designing a BHS usually use a bottom-up approach. The process typically starts by creating a computer-aided design (CAD) drawing of the system. This CAD drawing is used to plan out the interconnections of the various BHS components, such as the conveyors, the X-ray machines, the divert and merge sections, and so on. This constitutes the physical layout of the plant.

Subsequently, the designers must decide on the appropriate points in the system for placing the physical controllers. Software for these controllers are typically implemented using PLC languages prescribed by the IEC 61131 standard [4]. A physical test system will then be assembled using the appropriate mechanical components and PLCs in order to test the system before the final deployment.

There are two key problems with this bottom-up approach to the design of such complex systems. Firstly, every design created is specific to a given physical set-up of the plant, with little scope for reuse of software components. Secondly, debugging such systems is difficult. Without access to model-accurate visualizations of the system, testing of the control software with the plant can only be done on the physical system.

The problems associated with such a monolithic approach to control software development can be overcome with the component-oriented methodology offered by IEC 61499. The MVC paradigm fits nicely within this component-based approach, resulting in a streamlined methodology for the development of complex control systems. For this to work effectively, the system must first be partitioned into three orthogonal subsystems. These include the model of the plant, the controller, and the visualization for the whole system. The latter helps the designer to simulate and visualize the behaviour of the plant and the controller off-line.

IEC 61499 enables a BHS to be designed in a component-oriented manner by allowing different modules of the plant to be modelled using separate function blocks. Each block has an explicit input-output interface, which connects to other components of the plant. This forms a very high-level model of the physical plant. The functionality of the controller may similarly be described as a network of function blocks. The plant and the controller can then be made to exchange information
7.1. MODELLING A BAGGAGE HANDLING SYSTEM

Figure 7.1: Visualization of the baggage handling system simulation.

using appropriate communication function blocks, like those mentioned in the previous chapter. In order to visualize the behaviour of the resulting control system, the view of the system can be described using another network of function blocks, or with other tools for graphical visualization.

In the following subsection, an overview of the BHS will be presented through a discussion of the visualization design. Then, in a later subsection, the plant and controller models of the BHS will be elaborated.

7.1.1 Simulation and visualization of the BHS

For this purpose, a tool, called MHVIS (material handling visualizer), has been developed to create a visual representation of the BHS, together with a graphical interface for user interaction. Figure 7.1 shows the visualization of the BHS. This system consists of six conveyors, including one merging and one diverting conveyor. Each conveyor has three photoeyes, or infra-red detectors. In this model, there is a single bag source and a single bag sink.

MHVIS generates the visual representation of the BHS by parsing the plant’s function block description to create the layout of the conveyors. This approach enables MHVIS to accurately visualize any baggage handling system, given its function block model. Once the view has been created, MHVIS animates each component based on data from both the plant and the controller.
The generated visualization communicates with the plant and controller function block models using publisher and subscriber function blocks. Based on the outputs from the plant and controller, the visual representation of the BHS can graphically show the position of the bags on the conveyor belts, the conveyor movement, and the status of the diverter all in simulated real-time. This enables simulation of both the plant and the controller at the same time, without needing to deploy the control code on the actual physical system.

### 7.1.2 Function block implementation of the BHS

Each of the six conveyors in the BHS plant has inputs to control the belt motor and a diverter. It has two outputs, one representing the movement of the conveyor belt (a Boolean state change every 50mm travelled), and an array containing detection values of the photoeyes. The controller, in turn, must track the positions of all the bags in the system. In addition, it must control each conveyor to start, stop, and divert appropriately so as to dispatch each bag to its intended destination.
The design of the BHS benefited greatly from the IEC 61499’s hierarchical approach to software development. For instance, the BHS plant model for a conveyor segment was first described at a very high-level, as depicted by the ConveyorPlantModel block in Figure 7.2. That figure also shows the input-output interface of a single conveyor connected to two different networking blocks. At the input side, there is a subscriber block, ConveyorToPlantSub, from which it receives control commands. At the output side, there is a publisher block, PlantToConveyorPub, through which it sends the belt position and photoeye detection data to the controller. Both these blocks are composite function blocks that encapsulate the subscriber and publisher communication function blocks of the previous chapter, respectively.

The ConveyorPlantModel block was subsequently refined to a network of blocks as shown in Figure 7.3. This network captures the actual behaviour of the conveyor by modelling the photoeye and the conveyor belt as the ConveyorModel and ConvBeltModel blocks respectively. The relative displacement of the belt is indicated by the EncCount output of the ConvBeltModel block, which increments by one for every millimetre of movement. This value is fed to the ConveyorModel block, which models the photoeyes attached to the conveyor. On the physical system, the photoeye is used to detect the presence of bags on the conveyor belt. Using a model of bag positions on the conveyor and the current position of the belt, the ConveyorModel can accurately simulate instances when the photoeye would be obstructed by bags on the conveyor.

Once the plant model of an individual conveyor is obtained, the entire plant can be created. Outputs of an upstream conveyor (the BagOut event and data) will be connected to the inputs of a downstream conveyor (either the BagIn or the BagMerge events and data). Unconnected inputs are still read by blocks within the ConveyorPlantModel, but no new bag data will be received.

At the top-most level, the plant has Init and Rate events, which are passed to all the lower-level blocks. The Init event, common to almost all function blocks, is used to initialize the required data, while the Rate event is used to control the simulation speed of the blocks to facilitate debugging and testing.

The BHS controller is similar to the plant, with six ConveyorController blocks used to control each of the six conveyors. Figure 7.4 shows a conveyor controller’s functionality contained within a ConveyorController block and the associated network interfaces to communicate with the plant. Each conveyor controller must create a model of the bags on the conveyor based on the values of the photoeyes (PEDetects) and the belt movement (EncoderState) coming from the plant. The controller also requires additional bag data indicating the desired destination. By correlating the
Figure 7.3: Function block network of the *ConveyorPlantModel* modelling a single conveyor.

Figure 7.4: Controller model for a conveyor.
belt position with the changes in the photoeyes' Boolean values, the controller can
determine the position of the leading and the trailing edges of a bag. The controller
compares the destination of each bag with the destination codes of each diverter to
determine whether or not to divert. If it needs to divert a bag, a signal is sent to the
plant to push the diverter arm into the stream, deflecting the bag to an alternate
stream. The diverter is then returned to its normal position.

As can be seen, the ConveyorController obtains the values of PEDetects and En-
coderState from the PlantToConveyor_Sub block, which in turn, subscribes those
values from the PlantToConveyor_Pub block in Figure 7.2. On the other hand, the
MotorStart, MotorStop, DivertStart, and DivertStop output events of the ConveyorTo-
Plant_Sub block have been derived from the MotorControl and DiverterControl values
published by the ConveyorToPlant_Pub block. Such a design of the plant and the
associated controller constitutes a closed-loop model of the overall system.

The hierarchical nature of function blocks has one further advantage: it can also
be used to hide or preset static values for lower-level function blocks. This feature
is utilised in the controller by creating a generic conveyor controller that is able
to control the merging, diverting, and straight-line conveyors. Relevant parameters
specific to each conveyor controller type, such as the position of photoeyes and the
actual physical conveyor type being controlled, are abstracted within another block.
For example, the ConveyorController_3PEs_Divert block sets some attributes for the
three photoeyes and the diverter internally, exposing only data signals that may
vary between instances.

### 7.2 Experimental results

Once the development of the BHS was completed, experiments were conducted to
evaluate the quality of the code generated by FBC. A suite of other benchmark
programs were also evaluated during these experiments. The programs that were
evaluated range from small examples with less than five function blocks with around
two dozen input/output ports, to large examples involving dozens of function blocks
and hundreds of input/output ports (see Figures 7.5–7.8 for the list of programs).
The Speed Regulator, Drill Station, and Cruise Control 1 programs are synthetic
examples created by the author. The LED Flasher is a sample program from FBDK
[12]; the Temperature Controller and Water Monitor were adapted from [71]; while
the Cruise Control 2 and Railway Crossing have been adapted from [83] and [72]
respectively. The Baggage Handling system in the benchmark is the full model
of the function block example introduced in Section 7.1. This, together with the Distributing Station, are actual industrial examples. The code for the Distributing Station was created to control part of Festo’s Modular Production System [84], while the Baggage Handling program is a model of a segment of an airport baggage handling system from Glidepath [85]. All of these programs are available from [86], except for the Baggage Handling example, which cannot be disclosed due to a confidentiality agreement with Glidepath.

The code from FBC was compared with the code generated by the 4DIAC-IDE [13] and by FBDK [12]. Both of these are freely available and provide a widely-accepted approach for function block execution through their respective run-time environments, FORTE and FBRT. The code synthesized by FBC, 4DIAC-IDE, and FBDDK were evaluated based on their execution speeds and object code sizes. FBC was used to generate two different kinds of C code, each adhering to the instantaneous and delayed communication semantics respectively, as described in Chapter 5. The C code produced by FBC was compiled using gcc (version 4.4.3), while the Esterel code from it was first converted to C using the FastC V7 Esterel compiler [68], before being compiled with gcc. Meanwhile, the code synthesized by the 4DIAC-IDE (version 0.3) and FBDDK (version 20081003) were compiled using g++ (version 4.4.3) and javac (version 1.6) respectively. The javac compiler, as well as the Java Virtual Machine (JVM) used for these experiments, are from Sun’s Java SE Development Kit (JDK) 6. All Java programs were tested using the default settings of this JVM, while all C/C++ code were compiled with level 2 optimization (-O2).

For these experiments, a set of pseudorandom input vectors were generated for each benchmark program. Separate testbenches were created to feed these input vectors to FORTE, FBRT, as well as the C programs generated by the Esterel compiler and by FBC directly. The measured times do not include the time to run the testbench, so that the use of four different testbenches would not influence the results in any way. This was accomplished by running the testbenches separately, and then, subtracting their execution time from the experiment results. These experiments were all carried out on an AMD Turion 64 ML-32 processor with 1GB of RAM running the Linux 2.6 kernel.

The graph in Figure 7.5 compares the execution time taken to compute the reaction to one million input vectors for the benchmark suite following its synthesis by the five code generators. Note that the ordinate of the graph is in logarithmic scale. The figures displayed in the graph have been computed by averaging the execution times after five consecutive runs of each test program. This has been done to negate the effects of caching, as well as to account for possible initial lag
7.2. EXPERIMENTAL RESULTS

Figure 7.5: Comparison of benchmark execution times using various code generators. The vertical axis shows the time taken to compute the reaction to one million input vectors.

due to just-in-time compilation of the Java programs. The items labelled as “FBC-Strl,” “FBC-C,” and “FBC-Cinst” in the legend refer to the results obtained from the Esterel, as well as the “delayed” and “instantaneous” C codes generated by FBC respectively.

To better compare the performance of the code from the various code generators, the execution times for each benchmark, normalised to the time taken to execute the same benchmark using the “instantaneous” C code from FBC (FBC-Cinst), were plotted. Figure 7.6 shows that the “instantaneous” C code runs, on an average, 37 percent and 2.1 times faster than the “delayed” C and Esterel codes from FBC respectively. When compared to the code running on FBRT and FORTE, it is, respectively, 4.2 and 6.7 times faster. If we compare the average speed-up of the “delayed” C code over the Esterel code, which also relies on the same semantics, we find that the C code runs 53 percent faster. This is worth noting, since the Esterel code is eventually compiled to C as well, indicating that there are substantial performance gains in translating function blocks directly to C.

The graph in Figure 7.7 compares the object code size obtained for each of the benchmark programs using the five code generators. For FBC and FORTE, the height of the bar indicates the object code size after compilation with gcc and g++ respectively. The values have been obtained using the GNU size utility. For FBRT, the height of the bar is the size of the corresponding function block class files produced by javac. Again, the code sizes for each benchmark, normalised to the
Figure 7.6: Benchmark execution times normalised to the time taken to execute the “instantaneous” C code from FBC.

Figure 7.7: Comparison of benchmark code sizes using various code generators. The vertical axis shows the object code size. For FORTE and FBRT, the figures exclude the size of the run-time environments.
7.2. EXPERIMENTAL RESULTS

Figure 7.8: Benchmark code sizes normalised to the object code sizes of the “instantaneous” C code from FBC.

code sizes obtained for the benchmarks using the “instantaneous” C code generator of FBC, have been plotted. Figure 7.8 shows that both the C codes from FBC show little variation in terms of object code size. This is to be expected, as the code for basic function blocks are identical, while the code generated for function block networks differ primarily only in the netlist order. The C codes are significantly more compact than the rest, with the average code size for the programs in Esterel, FBRT, and FORTE being 3.6, 6.2, and 8.4 times larger than them respectively.

The relatively slow and bulky code produced by the 4DIAC-IDE in comparison with the rest may be due to the large overhead incurred by FORTE, as a result of its multithreaded implementation. For this experiment, in fact, additional steps have been taken to speed up its execution by recompiling FORTE and the function blocks running in it without any debugging information and maximum optimization (-O3 in g++). Still, as the results in Figure 7.5–7.8 indicate, the overall performance of function block implementations in FORTE is rather poor.

On the other hand, the efficient code produced by FBC, both in terms of execution speed and code size, shows that the proposed approach for function blocks is well-suited even for resource-constrained embedded controllers. Significantly also, the code generated by FBC avoids the overheads of a run-time environment used in other existing techniques, which in practice, would further add to their already much higher memory requirements. For the record, the code sizes of the FORTE and FBRT (excluding JVM) run-time environments stand at 4.6 MB and 564 kB, respectively. These have not been included in the comparisons.
7.3 Discussion

The experimental evaluations presented in this chapter show that the code generated by fbc compares very favourably to those produced by other existing tools. These experiments had been conducted on an AMD Turion running the Linux 2.6 kernel. A standard personal computer (PC) was used for the benchmarking, as it provided the most convenient platform to do so. In practice, however, control software would typically be implemented on a wide variety of platforms, ranging from small 8-bit microcontrollers to large 32-bit microprocessors.

To this end, code from fbc has been successfully tested on an embedded platform, namely the Nios II soft-core processor from Altera [87]. However, getting the run-time environments of FORTE and FBRT to work on the Nios II processor required substantial investments of time. Due to this, benchmark evaluations could not be carried out on the Nios II processor. The experimental results in this chapter, however, provide a good gauge of the efficiency of the code from fbc with respect to other code generators. There is no reason to believe that the conclusions drawn from the results of the experiments on the PC would differ substantially on different platforms.

One notable tool that was not evaluated in the experiments is ISaGRAF [14]. Benchmarking code from ISaGRAF is not so straightforward, as it is a proprietary tool, unlike the rest of the tools used in the experiments. ISaGRAF uses its own proprietary file format for function block descriptions, and cannot accept the open XML format prescribed by IEC 61499. All other tools which were used in the experiments can accept the prescribed XML format.

ISaGRAF also cannot describe ECCs directly. It makes use of sequential function charts (SFCs) from the former IEC 61131 standard to describe ECCs. Substantial amounts of time and effort would have, thus, been required to re-create the entire benchmark suite in ISaGRAF’s proprietary format in order to perform comparisons with them. Moreover, the code from ISaGRAF cannot be easily instrumented, as it links against proprietary libraries. Being a commercial tool, the necessary libraries for a standard PC were not freely available.

As a whole, however, the results presented have demonstrated that the approach and techniques proposed in this thesis are viable for developing control software with IEC 61499.
Conclusion and Future Work

This thesis has revolved around the IEC 61499 standard [5], which has been proposed to facilitate the development of distributed industrial control software. This standard from the International Electrotechnical Commission (IEC) provides a basis for such software to be described in an open manner, independent of any implementation target. The standard itself provides a component-oriented technology based on function blocks that makes it easy to understand and to reuse software components. Design artefacts prescribed by the standard allow complex distributed software to be built in a modular and hierarchical fashion.

These features are a significant advancement from the PLC programming techniques [4] that currently dominate the landscape for industrial control software. The standard itself is, thus, an acknowledgement of the deficiencies in the current state-of-the-art in control software development. The standard has emerged out of a concerted initiative by the industry for a more open and interoperable method to create control software, in contrast to the vendor-centric approach of today.

The IEC 61499, however, has suffered from numerous issues surrounding the execution semantics of function blocks, which lie at the heart of the component-oriented technology in the standard. Central to these problems is the lack of a model of concurrency to describe a network of function blocks. Early initiatives to overcome this deficiency have almost universally treated this as a scheduling problem [15, 16, 19, 21, 22, 50]. This has resulted in a variety of run-time environments and corresponding code generators [12–14, 51] that each implemented a different scheduling mechanism. All these different implementations have resulted in different behaviours for a given function block program. These discrepancies have also complicated efforts towards formal verification, as verification efforts must now first conciliate the expected program behaviour with a particular execution model. Behaviours of distributed function block systems are left even more open in the
standard, and with few formal models proposed for it so far.

The goal of this thesis has, thus, been to formalised the execution model of IEC 61499. The research undertaken has resulted in a GALS model of execution, where the behaviour of function blocks is synchronous within a resource, but asynchronous when viewed globally as a collection of distributed resources. Within this framework, the issues of function block scheduling, verification, code generation, and automatic synthesis of communication patterns for distributed systems have been addressed.

In Chapter 3, a precise execution semantics for function blocks based on the synchronous approach have been presented. These semantics elegantly resolved the issue of incompatible scheduling policies used in various existing run-time environments. At the same time, the semantics are fully compositional, in that it allows function blocks to be arbitrarily composed in synchronous parallel by guaranteeing that all compositions are reactive and deterministic. This has been achieved by delaying communications between function blocks. Due to this, function blocks can be compiled modularly in a truly component-oriented fashion. Since scheduling concerns are fully resolved by the synchronous approach, there is no longer a need for run-time environments to execute function blocks. These semantics have underpinned the tools and methodologies developed for IEC 61499 function blocks throughout this research.

In Chapter 4, the proposed semantics were used to translate function blocks to Esterel [28], a well-known synchronous language for specifying safety-critical reactive systems. This was achieved through a prototype compiler, called fbc, that was developed in the course of this research. This compiler was used to generate Esterel code for a suite of benchmark programs, which were subsequently verified using Esterel Studio’s Design Verifier. The properties verified consisted of both control and data properties. This is significant, as mature verification tools for function blocks are virtually non-existent, while existing prototypes have only been able to verify control properties.

Chapter 5 went on to describe the extensions to the fbc compiler to enable it to generate C code from function block programs. The requirement for delayed communication between function blocks was also relaxed to allow for instantaneous communication, so long as the communication does not take place between a set of strongly connected blocks. This is a substantial extension to fbc, as it must now schedule function blocks following the dependencies implied by the communication, and to ensure synchrony on its own. These did not need to be done previously, as they were performed by the Esterel compiler.

Chapter 6 then described how the proposed synchronous model could be natu-
rally extended to a GALS model to enable locally synchronous resources to communicate asynchronously with each other over a network. That chapter showed how concrete communication code could be automatically synthesized from abstract communication patterns. These abstract communication patterns were shown to be suitably described with IEC 61499’s notion of communication function blocks. Fbc was also further extended to enable it to automatically generate code for different communication patterns via a library of communication function block templates.

The viability of all these techniques was evaluated in Chapter 7 by using Fbc to develop a prototype airport baggage handling system in a case study [36]. Both the Esterel and C code produced by Fbc for this case study, as well as for a suite of benchmark programs, compared very favourably with existing function block code generators in terms of execution speed and code size. In particular, the C code from Fbc was, on average, 6.7 times faster and 17.6 times smaller than the equivalent code running in FORTE [13], a well-known function block run-time environment.

All these achievements augur well for the techniques that have been developed for function blocks in this thesis. The next section will attempt to give some insights to possible future directions that this work can take.

8.1 Perspectives and future work

This thesis has grown out of an attempt to investigate the suitability of IEC 61499 methodologies for the development of safety-critical industrial control software. In the course of this research, much has been achieved, but there are still numerous areas that can be developed for future improvement. This section will highlight some of these areas, which are deemed to be more relevant for industry adoption of IEC 61499.

8.1.1 Verification issues

Verification of function block programs have been demonstrated via a translation to the Esterel synchronous language in Chapter 4. The verification itself is performed by Esterel Studio’s Design Verifier. At present, however, if the verification of a given property fails, the counter-example provided by Design Verifier would have to be manually mapped back to the original function block description. This process can
be automated due to the syntax-directed translation between function blocks and Esterel. The counter-example which consists of a signal trace from the Esterel modules has a one-to-one correspondence to events and function blocks in the original specification, which enables this to be easily done.

Beyond this, one promising extension to function block verification lies in the possibility of using FBC to generate verification models directly as inputs to module checkers [88]. Module-checking differs from model-checking (performed by Design Verifier), as it can verify properties for *open systems*, where the non-determinism in the environment needs to be considered during verification. In model-checking, all transitions in every state of the model are always assumed to be enabled. This, however, may not be true of open systems, as the environment influences the transitions that are enabled. Preliminary work in this direction has begun by adapting the module checker in [89] for function block verification.

Recent abstraction refinement techniques [90] combining control and data abstraction may also be explored to enable programs that are larger than what Design Verifier can handle, to be verified. Challenges in the verification of distributed asynchronous systems will also be a key research area for future investigation. As reliable software becomes increasingly critical in industrial control systems, new technologies for function block verification are bound to find broad application in the industry.

### 8.1.2 Real-time requirements

The IEC 61499 standard has no provision for specifying timing constraints in a function block program. However, if function blocks are to be used in safety-critical control software, support for programming real-time systems would need to be provided. Preliminary work to obtain the worst-case reaction time (WCRT) of function block programs has recently been undertaken [91], based on the synchronous semantics proposed in this thesis. Unlike other techniques that rely on run-time environments, the synchronous approach greatly simplifies the effort required to estimate the execution time of function block programs due to the following:

1. *Program execution paths are well-bounded.* Since the whole synchronous program is confined within a reaction loop and unbounded iterations do not exist within that loop, the task of finding the longest execution path within the program is greatly simplified.

2. *The schedule of execution in each reaction is deterministic.* Since the synchronous approach ensures deterministic concurrency through compile-time
8.1. PERSPECTIVES AND FUTURE WORK

scheduling, execution time ambiguities owing to the dynamic nature of run-time environments are avoided.

These two features enable static analysis of assembly code that extracts timing information to be combined with model-checking to predict the WCRT of function block programs. An accurate estimate of the WCRT provides a formal basis for deciding whether or not a program will meet its real-time requirements.

8.1.3 Visualization/simulation of control systems

Industrial systems are typically described as closed-loop systems, consisting of a plant and a controller. Function blocks facilitate the description of such systems, as the plant and the controller can each be neatly encapsulated within a separate block with clear input/output interfaces for them to communicate with each other. Oftentimes, it is desirable that the resulting control system can be visualized. The visualization of the plant and controller provides an intuitive means for testing, and gives confidence to customers of control systems that the system is behaving correctly before final deployment and commissioning. Therefore, it is desirable for visualizations to be created seamlessly together with the plant and controller descriptions using function blocks.

An initial idea to do this has been trialled for an airport baggage handling system (BHS) \cite{36}. A customised tool was created to visualize a conveyor system for bags based on the plant model of the BHS. This idea can be extended to create a more general toolchain that can be used to integrate a model-view-control design paradigm within the IEC 61499 framework. For this concept to work, the controller for various mechatronic components would need to be encapsulated within appropriate function blocks. Then, by graphically connecting various mechatronic components together, the corresponding function block network for the plant and controller can be automatically derived.

This approach will enable models of a plant and controller to be simultaneously developed and tested using model-accurate visualizations. The ability to maintain the same system description throughout the entire development—from design conception, to architectural modelling, simulation, visualization, and synthesis—helps ensure coherency between user requirements and final implementation. The ability to design complete systems from a library of components in this manner can potentially reduce time-to-market significantly.
8.1.4 Distribution issues

The communication function blocks developed in this research have only targeted POSIX [73] sockets. Distributed industrial systems, however, are likely to use a variety of fieldbuses [79] for communication. The use of fieldbuses enables real-time and time-triggered communications in distributed systems. This potentially makes it possible to derive fully synchronous distributed function block systems that maintain I/O stream equivalence with a centralized implementation on a single resource. The approach proposed in [75] provides a suitable starting point to do this. This will enable distributed function block systems to be verified as a centralized synchronous program before deployment on a distributed platform. It is envisaged that with distributed systems being increasingly used in safety-critical applications, this will be a fruitful avenue for research.

8.2 Final remarks

The GALS paradigm proposed for modelling distributed IEC 61499 systems offers a compelling alternative to existing IEC 61499 implementations that rely on runtime environments for execution. The synchronous model for executing function blocks within a resource, as well as the asynchronous model for distributed resources, provide a natural way to describe logical and physical concurrency in distributed control systems.

For centralized implementations, the synchronous approach is an efficient solution that enables safety properties of function block programs to be easily verified using synchronous observers. For distributed implementations, the ability to automatically obtain concrete communication code from abstract communication patterns is both convenient and elegant.

All these have been demonstrated through a prototype tool that generates verifiable models, as well as centralized or distributed code, from function block descriptions. When evaluated against other function block code generators, the code quality from the prototype tool excels in execution speed and memory footprint. In safety-critical control systems that demand efficient implementations of verified code, the research undertaken in this thesis provides a compelling alternative.
References


The following sections contain the templates used by FBC to generate the publisher and subscriber communication function blocks. The text in blue function as special holders that will be appropriately replaced with the extracted indices from the publisher-subscriber blocks (see Chapter 6) in order to generate concrete implementation code.

A.1 Publisher template

A.1.1 Header file

```c
#ifndef PUBLISH_@index@_H_
define PUBLISH_@index@_H_

#define PUBLISH_@index@_H_
@copyFile#pubsub.h
@copyFile#publisher.c
#include "pubsub.h"
#include <stdbool.h>

typedef union {
    unsigned int events;
    struct {
        unsigned int INIT : 1; // service initialization
        unsigned int REQ : 1; // service request
    } event;
} PUBLISH_@index@IEvents;

typedef union {
    unsigned int events;
    struct {
        unsigned int INITO : 1; // initialization confirm
        unsigned int CNF : 1; // service confirmation
    } event;
} PUBLISH_@index@OEvents;
```


```c
#define MAX_ID_LEN 31
#define MAX_STATUS_LEN 31
typedef struct {
  int _state;
  bool _entered;
  PUBLISH@index@IEvents _input;
  bool QI;
  bool _QI;
  char ID[MAX_ID_LEN + 1];
  char _ID[MAX_ID_LEN + 1];
  ANY SD[indexRange#1@];
  PUBLISH@index@OEvents _output;
  bool QO;
  bool _QO;
  char STATUS[MAX_STATUS_LEN + 1];
  char _STATUS[MAX_STATUS_LEN + 1];
  Publisher publish;
} PUBLISH@index@;

/* PUBLISH@index@ initialization function */
void PUBLISH@index@init(PUBLISH@index@* me) {
  memset(me, 0, sizeof(PUBLISH@index@));
}

/* PUBLISH@index@ execution function */
void PUBLISH@index@run(PUBLISH@index@* me) {
  me->_output.events = 0;
  if (me->_input.event.INIT) {
    me->QI = me->_QI;
    strncpy(me->ID, me->_ID, MAX_ID_LEN);
  }
  if (me->_input.event.REQ) {
    me->QI = me->_QI;
    me->SD[indexRange#0@] = me->_SD_indexRange#1@;
  }
  if (!me->_entered) {
    me->_entered = true;
  }
  else {
```

A.1.2 Implementation file

```c
#include "PUBLISH@index@.h"
#include <string.h>

/* PUBLISH@index@ initialization function */
void PUBLISH@index@init(PUBLISH@index@* me) {
  memset(me, 0, sizeof(PUBLISH@index@));
}

/* PUBLISH@index@ execution function */
void PUBLISH@index@run(PUBLISH@index@* me) {
  me->_output.events = 0;
  if (me->_input.event.INIT) {
    me->QI = me->_QI;
    strncpy(me->ID, me->_ID, MAX_ID_LEN);
  }
  if (me->_input.event.REQ) {
    me->QI = me->_QI;
    me->SD[indexRange#0@] = me->_SD_indexRange#1@;
  }
  if (!me->_entered) {
    me->_entered = true;
  }
  else {
```
A.2. Subscriber template

A.2.1 Header file

```c
#ifndef SUBSCRIBE_@index@@_queue@_H_
#define SUBSCRIBE_@index@@_queue@_H_

#include <pubsub.h>
#include <stdbool.h>

typedef union {
    unsigned int events;
    struct {
        unsigned int INIT : 1; // service initialization
        unsigned int RSP : 1; // service response
    } event;
} SUBSCRIBE_@index@@_queue@IEvents;
```

```c
if (me->_input.event.INIT) {
    if (me->QI) {
        openPublisher(&me->publish, 1, me->ID);
        me->QO = true;
        strncpy(me->STATUS, "OK", MAX_STATUS_LEN);
        me->_output.event.INITO = 1;
    } else {
        closePublisher(&me->publish);
        me->QO = false;
        strncpy(me->STATUS, ",", MAX_STATUS_LEN);
        me->_output.event.INITO = 1;
    }
} else if (me->_input.event.REQ) {
    if (me->QI) {
        sendPublisher(&me->publish, me->SD, @index@);
        me->QO = true;
        strncpy(me->STATUS, "OK", MAX_STATUS_LEN);
        me->_output.event.CNF = 1;
    } else {
        me->QO = false;
        strncpy(me->STATUS, "INHIBITED", MAX_STATUS_LEN);
        me->_output.event.CNF = 1;
    }
} } 
```

```c
me->_input.events = 0;
if (me->_output.event.INITO || me->_output.event.CNF) {
    me->_QO = me->QO;
    strncpy(me->_STATUS, me->STATUS, MAX_STATUS_LEN);
}
```
typedef union {
    unsigned int events;
    struct {
        unsigned int INITO : 1; // initialization confirm
        unsigned int IND : 1; // service indication
    } event;
} SUBSCRIBE_index@_queue@OEvents;

#define MAX_ID_LEN 31
#define MAX_STATUS_LEN 31
typedef struct {
    int _state;
    bool _entered;
    SUBSCRIBE_index@_queue@IEvents _input;
    bool QI;
    bool _QI;
    char ID[MAX_ID_LEN + 1];
    char _ID[MAX_ID_LEN + 1];
    SUBSCRIBE_index@_queue@OEvents _output;
    bool QO;
    bool _QO;
    char STATUS[MAX_STATUS_LEN + 1];
    char _STATUS[MAX_STATUS_LEN + 1];
    ANY RD[@queue@][@index@];
    ANY* _RD_@indexRange#1@
    ANY* _head;
    ANY* _tail;
    int qlen;
    Subscriber subscribe;
} SUBSCRIBE_index@_queue@;

/* SUBSCRIBE_index@_queue@ initialization function */
void SUBSCRIBE_index@_queue@init(SUBSCRIBE_index@_queue@* me) {
    memset(me, 0, sizeof(SUBSCRIBE_index@_queue@));
    me->_head = me->_tail = me->RD[0];
}

/* SUBSCRIBE_index@_queue@ execution function */
void SUBSCRIBE_index@_queue@run(SUBSCRIBE_index@_queue@* me) {
    me->_output.events = 0;
    if (me->_input.event.INIT) {
        
A.2.2 Implementation file

#include "SUBSCRIBE_index@_queue@.h"
#include <string.h>

/* SUBSCRIBE_index@_queue@ initialization function */
void SUBSCRIBE_index@_queue@init(SUBSCRIBE_index@_queue@* me) {
    memset(me, 0, sizeof(SUBSCRIBE_index@_queue@));
    me->_head = me->_tail = me->RD[0];
}

/* SUBSCRIBE_index@_queue@ execution function */
void SUBSCRIBE_index@_queue@run(SUBSCRIBE_index@_queue@* me) {
    me->_output.events = 0;
    if (me->_input.event.INIT) {
        

me->QI = me->_QI;
strncpy(me->ID, me->_ID, MAX_ID_LEN);
}
if (me->_input.event.RSP) {
    me->QI = me->_QI;
}
if (!me->_entered) {
    me->_entered = true;
}
else {
    if (me->_input.event.INIT) {
        if (me->QI) {
            openSubscriber(&me->subscribe, me->ID);
            me->QO = true;
            strncpy(me->STATUS, "OK", MAX_STATUS_LEN);
            me->_output.event.INITO = 1;
        }
        else {
            closeSubscriber(&me->subscribe);
            me->QO = false;
            strncpy(me->STATUS, ",", MAX_STATUS_LEN);
            me->_output.event.INITO = 1;
        }
    }
    else {
        receiveSubscriber(&me->subscribe, &me->_tail, &me->_head, &me->_qlen,
                          @index@, me->RD[@index], @queue@);
        if (me->_qlen) {
            me->QO = true;
            strncpy(me->STATUS, "OK", MAX_STATUS_LEN);
            me->_output.event.IND = 1;
        }
    }
}
The following sections present the list of research articles that are based on the techniques presented in this thesis.

**B.1 Published papers**


**B.2 Papers under review**

B.3 Papers under preparation