Reengineering of IEC 61131 into IEC 61499
Function Blocks

Gareth D. Shaw
Department of Electrical and Computer Engineering
University of Auckland
Email: gsha041@aucklanduni.ac.nz

Partha S. Roop
Department of Electrical and Computer Engineering
University of Auckland
Email: p.roop@aucklanduni.ac.nz

Zoran Salcic
Department of Electrical and Computer Engineering
University of Auckland
Email: z.salcic@aucklanduni.ac.nz

Abstract—In order for industrial automation manufacturers to remain competitive, they must utilise modern design approaches. Existing approaches using IEC 61131 languages are not object oriented and are difficult to reconfigure for other applications. IEC 61499 is an open standard aimed at replacing IEC 61131, adding modern design features and hardware abstraction. Therefore there is a need to migrate existing code bases using IEC 61131 technologies into the newer IEC 61499 language for future development.

IEC 61499 is appealing to developers because of the simplified specification approach and benefits related to the language's abstraction. The basic design unit or function block provides a graphical method for control flow design, and uses algorithms written in any programming language. This object oriented approach enables easy block reuse and simple reconfigurability.

This paper presents a semi-automated process for the reengineering of Ladder Logic routines, from Rockwell, into a function block system. The process automatically translates all data types, variable declarations and ladder logic routines into their C equivalents. The function block architecture for the desired system components must be developed, but into which the generated code is encapsulated and used. The approach enables manufacturers using IEC 61131 to migrate their code base into a modern framework, with no loss of previous design efforts and minimal once-off overhead. IEC 61499 then allows for a more stream lined development process with reusable components.

I. INTRODUCTION

The existing design approach for industrial control systems utilises the IEC 61131 standard. The standard is comprised of a set of languages for industrial control. However, the languages: Structured Text (ST), Ladder Logic Diagrams (LLD) and Function Block Diagrams (FBD) have proven to be insufficient for flexible development [1]. The standard is further hampered by proprietary implementations of the languages, restricting developers to a single manufacturer. In order to produce flexible and re-configurable systems, a new technology and approach is required.

The newer IEC 61499 standard [2] was developed by enhancing the features of IEC 61131 function blocks. More object oriented features have been added to the basic functional unit or function block (FB) and an event-driven semantics has been implemented. FBs contain a Moore-machine for control flow handling, from which algorithms are called. Because of its basis in IEC 61131, algorithms within a basic function block may be written in any language including LLD and ST.

A basic function block may be composed into a network of blocks and encapsulated in a higher level composite block, thus abstracting away lower level functionality. Design using IEC 61499 function blocks (FBs) allows implementation of a model-view-controller design pattern, using hardware independent FBs. This both simplifies design and adds offline testing facilities not present with traditional approaches. The system can then be executed on a number of platforms, and ultimately, the controller may be synthesised into IEC 61131 compliant code to be executed on existing controllers. A library of easily reusable FBs can be developed and graphically added into future projects, thus reducing development and testing time.

The desire of manufacturers to move to a new approach is inevitable, and to IEC 61499 specifically is most likely, due to its relation to its predecessor. For industry, the IEC 61499 standard presents numerous benefits with respect to specification, as well as an open standard for platform independent design. A tool chain to aid manufacturers to migrate their existing code base into this new language will be an important milestone. In [3], the authors discuss reasons why people might chose to reengineer existing code: a reduction in maintenance costs, additional functionalities, and new hardware. The migration approach presented, was entirely manual and included analysing existing source code to manually create IEC 61131 code. While it is possible to integrate existing IEC 61131 code into FBs, code modification into the PLCOpen [4] semantics for IEC 61131 languages would be required. Additionally, this approach still retains the disadvantages of existing coding practices, and allows only minimal advantages of newer technologies to be leveraged. Therefore, this paper has chosen to automatically generate code from the existing IEC 61131 code base, in a more usable format, and implement it into the IEC 61499 framework. In [5], Thramboulidis notes that development frameworks will both simplify development and thus encourage adoption of the 61499 standard. This paper presents the precursor to framework usage, allowing for migration of previous development work.

In this paper, reverse-engineering is defined as: the adoption of existing code - in an old language, and creating a new implementation of the same system - in a new language. Re-engineering on the other hand, refers to modification of the architecture or structure of some code, i.e. using the same
code, but in a slightly different way.

This paper therefore presents a tool for the reverse-engineering of IEC 61131 code into C, which can then be re-engineered into IEC 61499 function blocks. While future work may present methods of reverse-engineering the architecture of 61331 code into 61499 FBs, in this paper, the C code is manually re-engineered into an IEC 61499 FB network, based on system analysis and documentation.

This paper is organised as follows. Section II discusses various approaches to reengineering of software. Section III outlines the benefits of the IEC 61499 standard. Then in Section IV, the approach taken in this work is explained and Sub-Section IV-B presents our translation algorithm with simple examples. Section V demonstrates the usefulness of the presented approach with a case study. Finally, section VI gives the conclusion of this work.

II. RELATED APPROACHES

Previous approaches to code reengineering can be divided into two sets: approaches using a high-level specification, and approaches just using low-level code. Many approaches use an entirely manual approach, but are still useful as a comparison of methodologies.

In [6], one of the presented approaches titled Straightforward ladder to FB transformation produces a network of blocks for a ladder logic routine, as opposed to a textual code. The only advantage of this is that it allows easy distribution of sections of ladder logic routines. However the approach relies on a large number of simple blocks to represent otherwise simple code, which reduces the maintainability. The approach proposed by this paper instead assumes a single routine will be executed in a single location, but allows independent routines to be distributed as required.

Christian et al. in [7] present another method for manual migration from IEC 61131 function blocks into IEC 61499 system descriptions. This work demonstrates the capability of IEC 61499 and presents a set of translation rules specifically for IEC 61131 function blocks. The work in this paper follows a similar set of rules, namely the consistency of routine and function block interfaces, but does not focus on 61131 function blocks. Some of the rules could similarly be used for translating IEC 61131 SFCs into ECCs, although this was not developed as SFCs were not used in the presented case study.

Using a higher-level specification, [8] introduces an approach for migrating from PLC to NetMaster controllers using IEC 61499. Their approach used PLCs programmed using signal interpreted petri-nets (SIPN). Using this source language, translation to an ECC for sequential tasks and multiple blocks for concurrent aspects was much simpler. The work presented in this paper is aimed towards data-heavy industrial control systems, such as baggage handling systems, which use the IEC 61131 set of languages for both control and data handling. The authors also note that a cyclic execution approach may have performed better because of the use of IEC 61499 service interface blocks for reading and writing data. However, cyclic execution was not an option for their case because of indeterministic TCP/IP networking. Variabilities in packet arrival times required the cycle time to be too long for the application. In this work distribution of the resulting FB system has not yet been considered, and due to minimal communication between conveyor components, cyclic execution of the included test case will not be an issue.

This paper instead proposes an automated compilation program in order to import higher level code into the new IEC 61499 standard. The generated C code can alternatively be utilised in a number of frameworks or approaches, allowing the benefits that each entails.

The manual development of the FB system can be implemented using a number of approaches, a discussion of which can be found in [1]. This paper partially utilises the approach presented in [9], namely the Model-View-Controller (MVC) design pattern.

III. BENEFITS OF IEC 61499

The IEC 61499 function block standard is intended as a new framework for distributed control, operating using an event triggered state machine, allowing any code for data handling algorithms. IEC 61131 code for example can be implemented within a function block algorithm, however, higher level languages such as C, C++ and Java — used in [10], [11] and [12] respectively — are preferred. The graphical execution control chart (ECC) in basic function blocks, aids control flow design. This removes the necessity to manually implement more complicated software state machines, and acts to hide data management into just an algorithm call within a state.

IEC 61499 provides specification methods for all layers of distributed control system design. Each function block specification is independent of hardware and operating systems. This allows a manufacturer to create a reusable library of function blocks, reducing lead-time in all future deployment projects. The object oriented nature allows for more flexibility and reconfigurability in control behaviour.

The adoption of IEC 61499 by industry is hindered by many factors. In [9] Christensen notes that "several aspects of this standard are unfamiliar to most practitioners of control systems engineering, especially the ideas of distributed applications, event driven execution control and service interface function blocks...". It is hoped that the presented migration approach to FBs will also serve to aid comprehension of the new framework due to direct mappings from the previous implementation.

The authors of [13] explain four factors which contribute to the slow adoption of IEC 61499 by the major control system equipment vendors. The four factors are several unproven aspects of function block implementations: maintainability, predictability of execution, scalability and extensibility. The work presented in this paper address these concerns by selecting a superior execution approach and using good coding practices within FBs. Other work since [13]'s publication have also partially addressed these concerns, such as the large scale examples presented in [14]. The large scale example used in
this paper’s case study also demonstrates FBs usability and effectiveness.

The standard ensures systems described using one of the IEC 61131 languages are also describable using event triggered function blocks. However, the usage of global variables in IEC 61131 is inadequately addressed in IEC 61499. Section V includes some discussion of global variable equivalence in the function block framework.

A. IEC 61499 Execution

FBs are executed by generating compilable code from each FB’s textual XML format. This paper uses the tools and semantics presented in [10] to create executable code from function block systems. The chosen semantics provide a synchronous approach to FB execution, similar to PLC execution and the generated C code out performs other execution approaches.

IV. IEC 61131 TO IEC 61499

A general tool for reverse engineering of PLC code is hindered by the proprietary implementations of the different manufacturers. Instead only a generalised process is possible, with manufacturer specific adjustments to the intermediate steps. As such, this work focuses on PLC code from Rockwell [15]. The presented approach focuses just on translation of code instead of architecture. Future work may automate the architectural phase of conversion, but this paper requires the manual creation of an FB structure. This approach was chosen because in the case study, the underlying structure of the IEC 61131 program changed substantially when re-implemented in FBs. The proposed reengineering process is comprised of the following steps:

- PLC code is saved in a proprietary textual format.
- An Abstract Syntax Tree (AST) is created using a manufacturer dependent lexical parser.
- The lexical tree is traversed and the program outputs global variables, data structures and functions in C according to the manufacturer’s PLC semantics.
- C functions are implemented as FB algorithms, though some suitable functions are re-implemented as ECCs.
- Commonly grouped FB components may be encapsulated in a higher level composite block.

The remainder of this chapter explains each of the reengineering phases. While this work only uses Rockwell’s LLD [16] and equivalent textual format, the same approach may be used for all PLC manufacturers.

A. PLC Parsing

For this work, the free compiler JavaCC [17] was used, although there are several alternative tools that can be used to produce identical syntax trees.

The ladder logic shown in Figure 1, and taken from Rockwell’s RSLogix IDE is exported into a textual format, shown in Fig. 2, which provides a sequential ordering of instructions. Each instruction can then be mapped directly into equivalent C statements, based on documentation in [16].

Compiler rules are supplied to JavaCC for all aspects of information in the textual L5K file. This file is then parsed into abstract syntax tree of nodes containing an ordered array of each identifier.

B. Code generation

From the abstract syntax tree obtained via parsing, an application developed specifically for Rockwell’s semantics generates C code. This program called L5K2C includes a mapping of all textual instructions, into equivalent C statements and mappings of PLC primitive data types into C primitives. More complicated instructions, such as the Branch instruction are handled in special functions in order to generate both semantically equivalent and human readable code.

L5K2C walks through the tree and outputs different information in relevant files. For example data types in the L5K file are all stored in a single sequence, these are then saved in a single C header file as structs. When a routine is found within a PLC program, it is stored as a separate file and the routine is renamed to include the program name as a prefix. Earlier versions of L5K2C struggled with variables with the same name as a routine, therefore the decision was made to rename all non-global routines and variables in the same way. The L5K file can contain global variables as well as variables scoped to a single program. Therefore global variables and program variables are stored in different files, and routines always include the global variables, but only variables scoped to the routines program name.

When generating equivalent C code, most LLD instructions have an equivalent C statement. However, the semantics of the Branch pseudo-instruction complicate compilation due to strange behaviour patterns. Referring again to Fig. 1, the semantics of the relatively simple looking LLD require that the MOV instruction must be executed if B is true, regardless of A, C and D. Finally, the output O must only be emitted if A||B||C||D is true.
In C code, this type of behaviour is implemented using a switch statement, as shown in Fig. 3. The case statements of a switch statement, are used to encapsulate the behaviour of each rung. An additional variable is used to determine whether or not at least one rung was successful. Only if successful, instructions after the branch are then executed.

```c
bool branchcompleted_1005 = false;

switch( )
{
    case 1:
        if( A )
            branchcompleted_1005 = true;
    case 2:
        if( X )
            branchcompleted_1005 = true;
    case 3:
        if( Y )
            branchcompleted_1005 = true;
    default:
        if( branchcompleted_1005 )
            break;
        else
            goto ENC_JONP_59;
}

0 = true;
ENC_JONG_85;
```

Fig. 3. Equivalent Behaviour in C

### C. Function Block Development

Generating C code from a PLC specification results in many C functions and header files which declare global or scoped variables and data types. In order to create a functioning model, this code could be directly encapsulated in a test bench application which migrates PLC I/O mappings into software variables and replacing hardware dependent statements. However, in order to harness newer design techniques and tools, it is beneficial to spend time to completely re-engineer this code into a modern specification such as IEC 61499.

The architecture of the function block network into which C functions are embedded, is preferably developed from a specification document, using ladder logic only to aid state machine creation. A system-level understanding of the code allows for accurate block interfaces and compositions, into which the reverse-engineering IEC 61131 code can be introduced.

A basic function block should only encapsulate code derived from a single PLC program. In this way the block can safely include variables only available in the program’s scope. Some C functions and their original LLD routines may resemble state machine behaviour. Such functions should be manually recreated as ECCs, calling other sub-routines as required. In the absence of a state-machine function for a component, a subroutine is likely to contain state-like behaviour, even if the code just contains an active and inactive modes.

Due to the cyclic nature of the original PLC specification, some algorithms may be called from states with always true loop-back conditions. To enable graphical reproduction of the more complex state-based ladder logic routines, the use of HCECCs [19] is required, allowing hierarchy and concurrent operations within a basic function block. This is especially useful in cases where a component has more than one routine with a state-machine, and for tasks that are executed each tick. In these cases, the two state-machines or condition monitoring states can be designed to be concurrent in the state machine editor. The use of HCECCs enables a switch in the FBC [10] compiler to generate appropriate code.

Finally, composite blocks are used to encapsulate related behaviour. Composite block also enable the engineer to set parameters for a block, and then hide the variable at higher levels of abstraction. This enables a simpler system-level design.

Inputs and outputs should be encapsulated in service interface blocks (SIFB). By using an SIFB the block can be easily replaced depending on the required destination hardware. This also allows variables to be communicated to other programs via networking or variable sharing on a PC or compatible platform. The usage of global variables between blocks’ algorithms may not be easily separable. As a result, distribution of the function blocks to different controllers may by limited by global variable sharing.

### V. CASE STUDY

Testing of this method was performed by reengineering a complex baggage handling system controller, and closing the loop with a separately developed plant model.

Analysis of a specification document as well as the ladder logic code and comments, lead to the specification of a conveyor controller illustrated in Figure 4, which is encapsulated within a composite function block called ConveyorController-ParamatisedWithNetwork. The network created is not an ideal FB implementation, because of the use of global variables. However this allowed for no loss of code development, while the FB architecture replaces only a few high level routines. From this initial implementation, global variables maybe phased out, introducing more communication between blocks, which may then be simplified by the use of IEC 61499 sockets. High inter-block data dependencies are un-implementable using the standard function block separation of data. This usage of global variables goes against the IEC 61499 standard, and thus is undesirable. Subsection V-B discusses some possible approaches for global data handling.

From the specification documents, a conveyor section contains a number of entry and exit devices, as well as photoeye or infra-red detectors. In the FB network shown, the variable number of entry, exit and update devices are encapsulated in container blocks, which only initialise the required number of device controllers. The higher-level conveyor processes, including maintaining the record of all bags in the system and controlling the conveyor belt motor are encapsulated in the ProcessConveyorStopStart and ConveyorController_HCECC blocks.

After compiling the LLD into C code, a few routines were re-implemented as an ECC within an FB; including the PEControllers within UpdateDevContainer. HCECCs [19] were used to simplify the specification of the main controller...
block ConveyorController_HCECC, which has several concurrent tasks. An HCECC block was also used in the plant system, where it is used to simultaneously monitor several different events. All FB ECCs are completely manually developed due to the complexity and variable nature used to define state-based control in IEC 61131. This manual process is quite simple as it just requires analysing a single LLD routine, reimplementing the state names used and associating the correct actions to each state. Although, implementing some of the conditional LLD as an ECC transition sometimes requires the use of FB internal variables and extra algorithms. Code was added to the end of the state’s algorithm, so that exiting transitions could just be used check the values of variables.

The generated C functions are copy-pasted into the function block definitions. The CompilerInfo tag in the FB’s XML is used to add custom include directives for including global variables and variables scoped to the current program. Referring again to Figure 4, the absent data connections between blocks signifies the currently hidden usage of global variables.

At the top most system specification level, a function block system is used to define the FB networks for the plant and controller subsystems. The controller subsystem is comprised of a simple network of composite blocks. The composite blocks at this level of abstraction are developed to preset many of the variables used in ConveyorControllerParamatised_WithNetwork, but for different functionalities such as merging, diverting and standard conveyors. The subsystems communicate using UDP sockets, which other programs, including a visualisation, are also able to monitor. Upon translating this system into C, using FBC [10], and then compilation with gcc, a single executable file is created. Running this file runs both of the subsystems in parallel. The following subsection details the code size and performance results.

A. Results

After completely reengineering the controller, the closed loop FB system resulted in 5.74 MB of C code. In comparison, the original L5K file was 7.28 MB, which only includes the open loop control code. The C code benefits from the use of FB networks for high level specification, instead of long and complicated LLD routines to implement the same thing.

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

The PLC implementation of the original LLD code must execute the fast task every 25ms, while the slow task is executed in any spare time the PLC has. The 25ms restriction, is the maximum rate at which some of the inputs may change, and all changes must be monitored to ensure accurate tracking. For PC execution, the benchmark was tested on a 2.5 Ghz dual core, 4GB RAM laptop. The system was able to execute 1 million cycles in 4 minutes and 47 seconds. That is an average of 287 $\mu$s per cycle of both plant and controller. It should be noted that comparison of the FB system on a PC is not demonstrative of the implementation performance of the FB network compiled back to a PLC. Future work will develop a compiler to translate FB code into IEC 61131 code for PLC execution. This translation will provide a more accurate speed comparison. It is important to note, however, that FB development is significantly faster than PLC development.

B. Discussion

This method allows the reuse of existing code in a modern industrial framework, with minimal time costs. The original
code’s proprietary syntax is reimplemented in standard C. The C code is then much easier to maintain, and IEC 61499 function blocks can be used to encapsulate functions into reusable components. Systems can be created easily from these components, and high level testing can be performed off-line using a developed visualisation.

Complete MVC modelling using a visualisation was hugely advantageous to debugging. Prior to usage of a visualisation, command line print statements and gdb were very restrictive for debugging high-level functionality. Visualisation of the entire system allows representation of the complete system state, and makes determining the cause of errors much easier.

On a technical note, global variables within the reengineered system need to be removed or implemented in a function block friendly way. Global variables contradict the envisaged component oriented nature of the FB standard. However, in many data-heavy and performance critical applications, data sharing using the IEC 61499 standard adds an unacceptable level of overhead. In this case study, it was also noted that global variables are especially useful for constant declarations, an aspect not considered in the standard. An option that may be considered is to extend the IEC 61499 standard to allow composite blocks to have internal variables, along with a different access semantics for these variables for embedded function blocks.

The synchronous execution of FBs remains similar to PLC execution, which aids creation of the FB architecture. Using C for execution, allows us to link arbitrary declarations into the executable, enabling global variable access. Given the synchronous semantics of both PLC and FB, when using code tested in 61131, there will not be any execution concerns.

At present ECCs are manually created, as the less restrictive LLD language allows for a variety of methods of state machine specification, not easily automatically translated. Given a standard specification approach, the translation tool could be extended to create ECCs from appropriate code.

While this work is concerned with modernising control software, it is still important to manage classical PLC hardware. PLCs will likely be integrated with newer control hardware as new standards solidify their position, and older technology is replaced. Integration of PLC design into new frameworks allows for configuration of both new and legacy controllers, thus translation of IEC 61499 function blocks into IEC 61131 semantics for execution would be desirable, and is proposed as some future work.

VI. CONCLUSION

The reengineering approach presented in this paper enables complete migration of PLC code into IEC 61499. Manual function block creation is a minor inconvenience as mapping of routines to blocks is simple, as is determining a block’s interface. Using this approach, a developer is then able to continue development using IEC 61499 and C code, which is much simpler, and without any loss of previous efforts.

Ideally the manual architecture creation from specification documents would be automated by analysing aspects of the code, though depending on the PLC code this may not be feasible.

Automatic translation of state-machine LLD code into an ECC would also be beneficial; optimising this would, however, be difficult. Especially with the chosen synchronous semantics, a single ECC transition takes one cycle. Therefore, if the transition chain between states with actions was too long, the performance would be severely dampened.

Future work may focus on removing or reducing the manual FB and ECC creation steps in the presented process. Implementation of the created FB system on existing IEC 61131 PLCs is also very desirable to manufacturers. Allowing them to maintain their current controller, before committing completely to distributed technologies.

REFERENCES


