Efficient Implementation of IEC 61499 Function Blocks

Li Hsien Yoong, Partha S Roop and Zoran Salcic
Department of Electrical and Computer Engineering
University of Auckland, Auckland

Abstract—Current trends in the manufacturing sector have greatly increased the need for portability and reusability in software systems for industrial control and automation. The IEC 61499 standard has emerged out of this need for a platform-independent way to develop industrial control systems. So far, typical IEC 61499 solutions have relied on some run-time environment that differ in the execution model for each implementation. This has resulted in code that is neither portable, nor efficient in terms of execution speed and memory footprint. Therefore, we present, instead, a synchronous model for function blocks, which removes the need for a run-time environment by making all event scheduling decisions during compile-time. This approach has been used to create a function block compiler, with two different back-end code generators. We present experimental results of a benchmark suite implemented using different code generators for different execution models, and show that our technique yields significant gains in efficiency over existing approaches.

I. INTRODUCTION

IEC 61499 [1] has emerged as the standard for developing industrial automation and manufacturing systems using a component-oriented approach with function blocks. A function block abstracts a functional unit of software by encapsulating local data, state transitions, and algorithms within a well-defined event-data interface. This ability to encapsulate an autonomous unit of software facilitates reuse and the distributed design of applications. Whole systems can be described, independent of any implementation platform, by connecting function blocks together in a network. These blocks communicate with each other following an event-driven model to the traditional cyclic-scan model used in programmable logic controllers (PLC). The work in [7], for example, maps each function block in a network to a separate program, and orders them in an execution list. Then, in each scan cycle, the input events associated with each block are checked, and the corresponding code will be executed only if some input event for it has occurred. Output events are issued immediately, and may be tested by other blocks in the same scan cycle if they appear below the current one in the execution list; otherwise, the output events would only be read in the next cycle. ISaGRAF [8], the only commercial IEC 61499 implementation at the moment, adopts this cyclic-scan model.

Obviously, these various run-time environments will produce different behaviours for a given application. Such discrepancies have been widely reported in literature [3], [6]. This is largely due to the lack of rigorous specifications in the standard for describing the composite behaviour of a network of function blocks. Moreover, even subtle variations in the implementation of the run-time environment may result in very different application behaviour, as shown in [9].

So, instead of relying on any particular run-time implementation, we adopt a synchronous semantics for function blocks [10], [11]. We have developed a function block compiler, called FBC, that can compile an arbitrary function block network into a deterministic finite state machine based on the synchronous approach. All scheduling decisions are determined at compile-time, resulting in fully predictable code that can run without a run-time environment. This not only avoids the issue of portability, but lends itself also to the efficient implementation of function blocks based on a formal model. Efficient code generation from high-level models for resource-constrained embedded controllers is a significant consideration in the industry, and is the main focus of this paper.
II. IEC 61499 FUNCTION BLOCKS

This section introduces the essential elements of IEC 61499 function blocks through the example in Fig. 1. The figure depicts a simplified cruise control system for a car modelled using function blocks, presented initially in [11]. The cruise control system is activated by a lever, consisting of the Accel and Off buttons. Whenever the Accel button is held down, a sequence of AccelHold events will be generated to incrementally accelerate the car. When the button is released, the AccelRelease event will be generated and the speed at that instant is memorized. This desired speed will be forwarded to the cruise controller, which in turn will attempt to maintain this speed by appropriately adjusting the throttle position. A separate subsystem calculates the current speed at every Clock tick and updates the cruise controller. This cruising mode will be deactivated when the Off button on the lever is pressed.

The function blocks in Fig. 1 are known as basic function blocks. Besides basic function blocks, IEC 61499 also defines two other kinds of function blocks, namely:

- **composite function blocks**, which provide a way to encapsulate a network of function blocks within another block; and,
- **service interface function blocks**, which serve as device drivers to bind the function block application to a specific hardware target. These have been omitted in Fig. 1 for brevity.

Moreover, the standard allows a network of function blocks to be grouped within an independent unit of software known as a resource, and a complete system may be described using a collection of resources.

A function block interface is defined by its event and data inputs, as well as its event and data outputs, as illustrated in Fig. 2. Event lines are drawn on the upper part of the block, while data lines are connected to the lower part. Event-data associations may be created at the interface to update the values of input data and to produce new output data together with the associated event(s).

The execution logic of a basic function block is determined by its execution control chart (ECC), as illustrated in Fig. 3 for the CruiseControlLever function block. An ECC consists of EC (execution control) states, EC transitions, and EC actions. These elements are labelled in the ECC of Fig. 3. The initial state of an ECC is represented with a double-box by convention.

The transition conditions between states in an ECC are evaluated whenever the function block receives an input event. These conditions are expressed using an input event and/or a boolean guard condition. Each state can be associated with zero or more actions, which may consist of an algorithm and a possible output event to be issued at the algorithm’s completion. The action(s) associated with a given state will be executed once on entry to the state.

When connected together in a network, the output events from one function block may in turn trigger the execution of other blocks. However, the nature of event flow and the scheduling order of blocks within a network are not clearly defined. In fact, the IEC 61499 standard seems to be open with regards to 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 [1]).

Thus, in the sequel, we will introduce our execution semantics for function blocks based on the synchronous approach. This approach has already proven itself to be a mathematically sound technique for developing safety-critical control applications [12]. The formal semantics afforded by the synchronous approach will be used by our compiler to facilitate the efficient implementation of function blocks.

III. SYNCHRONOUS MODEL FOR FUNCTION BLOCKS

The proposed synchronous model views function blocks within a network as concurrently executing modules of a system, as in [10]. Industrial control software is indeed often conveniently described as a collection of concurrently-running processes. The inherent concurrency in function block models, however, may be realised using various scheduling techniques.
One common approach to do this makes use of a cyclic executive, as described in [7]. In this model, the concurrent execution is simulated by executing a portion of code for each function block within a network in each scan cycle. This approach gives predictable temporal properties, making it the preferred choice for safety-critical systems. The function block execution order within a cycle, however, is a crucial implementation detail, since the performance and the behaviour of the application directly depends on it.

The synchronous approach formalizes this notion of a scan cycle by dividing time into discrete instants, known as a tick. Computations performed within a tick are assumed to be instantaneous, with time progressing only when crossing tick boundaries. This notion 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.

In our model, the tick is mapped directly to the period of a scan cycle. 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. All function blocks in a given network are conceptually viewed as running concurrently in lock-step with one another, performing atomic computations in each tick. For a basic function block, the atomic operation within a tick consists of the evaluation of EC transitions in the current EC state, and the corresponding computation of the action(s) in the destination state, should a transition be taken. The life-span of an event is strictly defined to persist for the duration of the tick in which it occurred. Thus, no more than one EC transition can occur within a tick in each basic function block of a network.

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 [7], [13]. 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 flow [5], [6], [14]. If multiple events are issued in a given EC state, and event connection loops exist in the network, this approach will either result in the purposeful lost of events, or the need for unbounded queues to store events. Both of these are undesirable in safety-critical systems, which are typical of industrial control applications.

However, even with a synchronous model, compositions of function blocks involving event connection loops may still be problematic, as discussed in [10]. Event feedback loops may possibly result in non-causal cycles within the function block network. Such feedback loops are considered non-causal as they blur the distinction between input and output events. In a synchronous model where computations conceptually occur instantaneously, non-causal cycles will manifest themselves as a deadlock of the system, or equivalently, as a starvation of those function blocks that do not form part of the cycle.

So, in order to achieve seamless composition of function blocks within the synchronous paradigm, we postpone all event/data communication between function blocks to the next tick. Whenever there is a sender-receiver relationship between a set of function blocks, our technique ensures that the sender will emit the event/data in the current instant, while the receiver will only read what is emitted in the next tick of the program. This simple “pipelining” of the send and receive operations in each function block guarantees that their parallel composition will always be acyclic [11]. 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 is in constrast to the work in [7], where the application behaviour is very sensitive to the block execution order within a scan cycle. The ability to exchange the overhead of a runtime scheduler for a predictable, yet simple, static scheduling scheme, makes the proposed approach a compelling solution for the deterministic and efficient execution of function blocks.

IV. FUNCTION BLOCK COMPILATION

To evaluate the feasibility of the proposed synchronous model, we have developed a function block compiler, called FBCJ, that can compile an arbitrary function block network into executable code. FBCJ is capable of generating code for function blocks in either C, or Esterel [15], a well-known synchronous language. Esterel has been used in the design and formal verification of safety-critical systems, and can itself be compiled to C [16]. Our translation to Esterel has been described in [10], [11], and we focus here on the translation to C instead. While the translation to Esterel paves the way for the formal verification of function blocks using existing tools for Esterel, the direct translation to C implements function blocks far more efficiently.

In FBCJ, the translation process begins by taking as input a function block network, specified in the IEC 61499 XML format [17]. It performs 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. This process is repeated for each resource described in the input file. At present, a number of common service interface function blocks as prescribed in [1] (such as the Publish and Subscribe communication function blocks), are supported by FBCJ. FBCJ provides a library of standard function blocks by including implementation templates in C that are used to automatically generate code for service interface function blocks.

Every function block type will be compiled into 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 implemented as simple bit variables in C. Each function block type also implements at least two functions, named FBTTypeInit and FBTTypeRun. FBTTypeInit 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 languages.

For basic function blocks, the FBTTypeRun function implements the execution code for the corresponding ECC.
Algorithms within an ECC are implemented as separate C functions. For composite function blocks, the FBTTypeRun 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 FBTTypeRun functions.

The translation details for both the basic and composite function blocks are elaborated further in the following subsections.

A. Translating the Basic Function Block

The main task in the translation of the basic function block lies in the extraction of the state and transition information from the ECC. This information is later used by the compiler to implement the ECC using a switch statement in C. FBC begins this process by parsing each node in the ECC to create a synchronous state representation, called SyncState. Each SyncState consists 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 a successor state;
- **Children** is the list of successor states; and
- **Parents** is the list of predecessor states.

As an example, the Accel state in Fig. 3 will be described by the quadruple \{Accel, Set Desired Speed, Off, Disable, START\}.

Once each EC state has been converted to a SyncState, the resulting object is a tree, rooted at the SyncState corresponding to the initial state of the ECC. A simple depth-first traversal is then performed 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 for generating the code for a given basic function block is shown in Fig. 4.

As can be seen in lines 5–8 of Fig. 4, variables representing the data ports of the basic function block are specially handled. We generate for each data port two separate variables—one to represent the data value as seen within the function block, and another to indicate the value at the interface. Code will be generated there for each action in that state (lines 15–22). The loop in lines 13–28 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 in that state (lines 15–22).

In lines 23–27, each transition condition leading to a next state will be converted to a distinct if statement. For instance, the transition conditions leading out from the START state in Fig. 3 will be translated into the following form:

```c
if (me->_input.event.INIT) {
    me->_state = 1; // INIT state
    ...
}
else if (me->_input.event.AccelHold) {
    me->_state = 2; // AccelHold state
    ...
}
else if (me->_input.event.AccelRelease) {
    me->_state = 3; // Accel state
    ...
}
else if (me->_input.event.Resume) {
    me->_state = 5; // ResumeCC state
    ...
}
```

Once the function block has completed its execution for a given tick, all its input events will be cleared (line 29). If any
procedure GenerateCFB(fb)
P := set of all input ports in the network of fb;
foreach p ∈ P do
    I := p.connectionSet();
    if p.type = EVENT then
        foreach i ∈ I do
            generate code to assign p to event, i;
        end
        else if p.type = DATA then
            generate code to assign p to data, i ∈ I;
        end
    end
foreach component block, b, in the network of fb do
    make call to execute b;
end
Q := set of output ports at the interface of fb;
foreach q ∈ Q do
    O := q.connectionSet();
    if q.type = EVENT then
        foreach o ∈ O 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 ∈ O;
        end
    end
end procedure

Fig. 5. Algorithm to construct a netlist for a function block network.

output event has been emitted within that tick, all associated output data will be updated with their new values, as computed by the internal algorithms (lines 30–32).

B. Translating the Composite Function Block

FBC compiles the network of function blocks within a composite function block into a netlist. To accomplish this, instances of the component blocks are encapsulated within the composite function block structure in C, in addition to the members for the event-data interface. The FBTypeinit function of a composite function block will call the respective FBTypeinit functions of its component blocks to correctly initialize the entire network during instantiation. The algorithm for constructing a netlist from a function block network is given in Fig. 5.

Due to the delayed communication semantics described in Section III, the compilation process is greatly simplified, as the communication and execution of component blocks can be arbitrarily scheduled. The netlist construction proceeds in a straightforward manner, with code generated, first, to update the inputs of each component block (lines 3–12), then, to execute each component block (lines 13–15), and finally, to update the outputs at the composite function block’s interface (lines 17–26).

The connectionSet() method on lines 4 and 18 returns the set of connections to a given port. Multiple connections may be made to a single event port, but only at most 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 5–11, as well as for the output ports at the composite function block’s interface in lines 19–25.

V. EXPERIMENTAL RESULTS

In order to evaluate the quality of the code produced by FBC, we ran experiments to compare the execution speed and code size for a suite of benchmark programs with the corresponding code produced by FBRT [2] and the 4DIAC-IDE [4]. Both FBRT and the 4DIAC-IDE are available for free, and provide a widely-accepted approach for function block execution through their respective run-time environments, FBRT and FORTE.

FBC was used to generate C and Esterel code for each function block application. Each Esterel program was subsequently compiled to C using the V7 Esterel compiler in Esterel Studio [18]. All C programs were then compiled using gcc. Meanwhile, programs running in FBRT and FORTE were compiled using the javac and g++ compilers respectively.

Tables I and II give the list of our benchmark programs and summarize the results obtained. The columns labelled FBC-Strl and FBC-C refer to the results obtained from the Esterel and C code generated by FBC respectively.

For these experiments, a set of pseudorandom input vectors were generated for each benchmark program. Separate testbenches were created to feed these input vectors in 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 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.

Table I shows the average time taken to compute the reaction for one million input vectors. The C code generated by FBC consistently ran much faster than all the others, surpassing, on average, the Esterel programs by 2 times, the FBRT programs by 3 times, and the FORTE programs by as much as 11.8 times. Even the Esterel code from FBC ran consistently faster.

### Table I

<table>
<thead>
<tr>
<th>Programs</th>
<th>FORTE</th>
<th>FBRT</th>
<th>FB-Ctrl</th>
<th>FB-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cruise control</td>
<td>1200</td>
<td>209</td>
<td>274</td>
<td>147</td>
</tr>
<tr>
<td>Drill station</td>
<td>789</td>
<td>321</td>
<td>194</td>
<td>78</td>
</tr>
<tr>
<td>LED flasher</td>
<td>649</td>
<td>247</td>
<td>154</td>
<td>67</td>
</tr>
<tr>
<td>Speed regulator</td>
<td>1648</td>
<td>323</td>
<td>125</td>
<td>73</td>
</tr>
</tbody>
</table>

### Table II

<table>
<thead>
<tr>
<th>Programs</th>
<th>FORTE</th>
<th>FBRT</th>
<th>FB-Ctrl</th>
<th>FB-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cruise control</td>
<td>12.3</td>
<td>13.8</td>
<td>12.0</td>
<td>7.65</td>
</tr>
<tr>
<td>Drill station</td>
<td>70.6</td>
<td>6.54</td>
<td>6.46</td>
<td>3.52</td>
</tr>
<tr>
<td>LED flasher</td>
<td>22.2</td>
<td>3.46</td>
<td>6.80</td>
<td>2.04</td>
</tr>
<tr>
<td>Speed regulator</td>
<td>98.8</td>
<td>8.76</td>
<td>6.28</td>
<td>4.62</td>
</tr>
</tbody>
</table>


than its counterparts in FORTE and FBRT, except for the cruise control example, where FBRT managed to execute 31 percent faster. On average, the Esterel programs, however, significantly outperformed both the execution on FBRT and on FORTE.

Table II shows the object code size obtained for the benchmark programs for each code generation technique. For the code produced by FBC and the 4DIAC-IDE, the figures in the table indicate the object code size after having been compiled with gcc and g++ respectively. The figures under the FBRT column are the sizes of the corresponding function block class files produced by javac. Once again, the C code from FBC is significantly superior to the rest, with the average code size for the Esterel and FBRT programs being 80 percent larger, and the programs for FORTE being 17.6 times larger.

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 already 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++). Moreover, all logging operations, which are turned on by default in FORTE, were commented out. Still, as the results in Table I and II indicate, the overall performance of function block implementations in FORTE is rather poor.

On the other hand, the significant gains obtained from the code produced by FBC, both in execution speed and memory footprint, clearly demonstrate the viability of implementing function block programs using the proposed synchronous model. Unlike other techniques, the minimal memory and computational resources required with our approach make it possible to implement efficient function block systems even in resource-constrained embedded controllers. This approach, furthermore, avoids the overhead of a run-time environment, which would, in practice, further add to the already much higher memory requirements of the other techniques.

VI. Conclusion

This paper has introduced a novel technique to efficiently implement IEC 61499 function blocks. The work here has presented a formal synchronous model for function block execution and a corresponding function block compiler. Experimental results have shown that this compiler can generate very fast and compact code for function block applications, substantially excelling all other currently existing techniques. We believe that this ability to generate efficient code, as well as the ability to avoid the need for a run-time environment, would further pave the way for greater adoption of IEC 61499 function blocks by the industry.

With the proposed synchronous model and our current translation technique to Esterel and C, we envisage that function block systems can first be verified using existing verification tools for Esterel utilising synchronous observers and assertions [19]. Once a given system has been verified, efficient control code can then be automatically generated from the function block model using the C code generator of FBC. In modern manufacturing environments, where efficient implementations of portable and verified code are desirable, our technique offers a compelling solution.

References


