Determining the Worst-Case Reaction Time of IEC 61499 Function Blocks

Matthew M Y Kuo*, Li Hsien Yoong†, Sidharta Andalam‡ and Partha S Roop§
Department of Electrical and Computer Engineering, University of Auckland
Email: {mkuo005*,lyoo002‡,sand080‡}@aucklanduni.ac.nz, p.roop@auckland.ac.nz§

Abstract—The IEC 61499 is an international standard for describing industrial process-control systems. Such systems typically consist of embedded computers that interact closely with physical processes within a feedback loop. In order to correctly control these physical processes, computations in response to inputs need to be done in a timely manner. A program’s worst-case reaction time (WCRT) to inputs is usually used to ensure that timing constraints are met. Unfortunately, the standard has no provisions for specifying real-time constraints. Moreover, typical implementations of IEC 61499 are tightly coupled to their run-time environments—each with possibly different semantics and temporal properties—which makes it difficult to automate the estimation of their WCRTs. We propose to adopt a synchronous model for IEC 61499 programs. This allows the programs to be executed without the need of a run-time environment. Consequently, we are able to use a novel model-checking technique to estimate the WCRT of IEC 61499 programs. Experimental results on a suite of programs show that this approach provides conservative estimates that are, on average, less than 10 percent off from the actual WCRT.

I. INTRODUCTION

Software for hard real-time embedded systems must be rigorously proven to meet all its timing constraints. This is particularly true of embedded software used in automation and control systems, which oftentimes need to control physical processes with strict timing properties. The ability to satisfy these timing constraints is typically decided by determining the worst-case execution time (WCET) of the control software. For automation and control systems, software development is typically done using domain-specific languages. The IEC 61499 standard is one such example of these. It prescribes the use of function blocks to facilitate a component-oriented approach to software development for industrial process-control systems [1]. Control software can be intuitively developed by connecting function blocks together within a network. The standard, however, has no provisions for specifying real-time constraints. Moreover, current IEC 61499 implementations are tightly coupled to their run-time environments (e.g. FBRT [2] and FORTE [3]) for execution. These different run-time environments may have possibly different semantics and temporal properties, which makes it difficult to automate the process of ensuring that timing constraints are satisfied.

Recent research, however, has proposed to execute function blocks using a synchronous approach [4]. This technique removes the need for a run-time environment, as function blocks can be statically scheduled and compiled into a stand-alone program. Function block programs that are implemented using the synchronous approach bears many similarities with classical PLC programs. Like PLC programs, each function block in a network will be scheduled within a loop, where computations and outputs are generated synchronously, as a reaction to inputs in each cycle. This approach greatly simplifies the effort required for ensuring that the timing constraints of a program are satisfied in two ways:

1) Program execution paths are well-bounded. Since the whole synchronous program is confined within the 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. In fact, the general undecidable problem of finding a program’s WCET [5] is reduced to finding the worst-case reaction time (WCRT).

2) The schedule of execution in each reaction is deterministic. Since the synchronous approach ensures deterministic concurrency through compile-time scheduling, execution time ambiguities owing to the dynamic nature of run-time environments are avoided.

These two features allow us to use a combination of low-level static analysis to extract timing information of the executable code and model-checking to predict the WCRT of IEC 61499 programs. An accurate estimate of the WCRT provides a formal basis for deciding whether or not a program will meet its timing requirements.

Metzner [6] highlighted that model-checking based timing analysis is competitive to earlier approaches based on integer linear programming (ILP). Kim et al. [7] have used the CBMC model checker (which we also used here) to determine the WCET of sequential C programs by deriving loop bounds automatically. We do not need to deal with loop bounds here, but are concerned instead with deriving the WCRT of synchronous programs that are inherently concurrent. Other recent efforts towards the timing analysis of synchronous programs are exemplified by the ILP based formulation of [8] and the algebraic formulation of [9]. Unlike these approaches, Roop et al. [10] have used model checking to obtain tighter estimates than earlier approaches by taking state-dependencies of multithreaded programs into account. The technique described here takes both the state and data dependencies into account to provide even tighter estimates of the WCRT of a program.

In the function block domain, the work in [11] is perhaps the most relevant. It attempts to determine the timing requirements
for executing a function block network by modelling the network as a Net Condition/Event System. A model checker is then used to statically analyse the execution time of a particular path in the network, taking into account the time required to execute portions of the modelled run-time environment. This work, however, assumes that the execution time for each block in the network is already known. This value could be easily obtained for the given example, as it was made up of simple function blocks. Obtaining the execution time for function blocks involving complex algorithms, however, will not be so trivial and was not discussed at all in this work.

To our knowledge, the tools and technique that we are describing in this paper are the first to be able to guarantee the WCRT of IEC 61499 programs. Before proceeding to describe these in detail, we provide first some preliminary information on IEC 61499 function blocks.

II. FUNCTION BLOCK PRELIMINARIES

A function block system is made up of interconnected blocks. The most fundamental block is the basic function block, exemplified in Fig. 1a. This basic function block implements an addition operation by summing its input data (‘IN1’, ‘IN2’) whenever it receives an input request (‘REQ’). Event-data associations may be created at the function block interface. For example, the ‘OUT’ data port of the adder block will only be updated when the ‘CNF’ event happens.

The execution of a basic function block is described by an Execution Control Chart (ECC). This chart is similar to a finite state machine, where states are represented by boxes, and transitions are marked using arrows. The initial state of an ECC is indicated by a double rectangle. Each state in an ECC may be linked to one or more algorithm(s) and/or output event(s). Fig. 1b shows the ECC of the adder function block. On initialization, this ECC transitions to the ‘CALC’ state from the initial ‘START’ state, and executes the ‘REQ’ algorithm with the initial input values. The output event ‘CNF’ and the resultant ‘OUT’ are then emitted. The block then waits for further ‘REQ’ events.

There are two other types of function blocks, namely the service interface and the composite function blocks. A service interface function block acts as a wrapper to abstract the underlying hardware. This is similar to device drivers and is typically provided by vendors. A composite block is used to encapsulate function block networks within another block.

Fig. 2 shows an example of connecting function blocks together to form a more complex system. The system is composed of an adder block, a minus block, and a multiply block in order to perform the following calculation:

$$(X + Y) \times (X - Y) = X^2 - Y^2.$$  

The execution semantics we have adopted is that of the synchronous approach proposed in [4]. According to this semantics, function block programs evolve in a sequence of logical clock ticks. In each tick, every function block in a network reads its inputs, evaluates a transition, as well as computes an algorithm and emits the corresponding outputs.

III. TIMING ANALYSIS

Timing analysis of function blocks starts with the compilation of function blocks to the C language. The compiler, FBC [13], has been updated to annotate code with additional tags that will facilitate timing analysis. This code is then compiled using GCC to obtain assembly code with time tags. This assembly code is then given to a timing analyzer to calculate the time (clock cycles) required to execute each block of the tagged code. The annotated C code is then updated with the calculated time values to produce a time-annotated C code. The model checker, CMBC [14], is used to incrementally predict the WCRT of the function block system until the true WCRT is verified. Fig. 3 shows a flowchart of the steps and tools involved in obtaining the timing information of function blocks. In this section, the X2Y2 function block system in Fig. 2 will be used as a running example to illustrate how to obtain WCRT for function block systems.

A. Compilation to C

For timing analysis, function block programs have to be first compiled to a target implementation. Using FBC, function blocks may be compiled directly to C code that follows the synchronous semantics described in [4]. Compiling function blocks directly to stand-alone C code is more light-weight and more suited for real-time applications when compared to other implementations [15].
int main(void) {
    // initialization reaction
    X2Y2Reg _sysInterface;
    memset(_sysInterface, 0, sizeof(X2Y2Reg));
    X2Y2 _var;
    X2Y2init(&_var);
    X2Y2run(&_var);
    // reaction loop for subsequent reactions for (;;) {
        // Code to read input data and event.
        X2Y2run(&_var); // execute reactive method
        // Code to update output data and event.
    }
    return 0;
}

Fig. 4. The main function for the X2Y2 system execution.

FBC implements function blocks using a data structure and a number of methods for execution. The data structure stores the current state of the ECC, input/output events, and the input/output data for each instance of that function block type. For composite blocks, the data structure also includes the data structures of the encapsulated blocks. The link between function blocks are implemented using shared memory for local system communication. Custom link algorithms can be implemented by the user for distributed communication such as TCP/IP. At least two methods are implemented to describe the execution of each function block, the initialization method and the reaction method. The initialization method initializes the data structure of each function block instance. For composite function blocks, the initialization method, in addition, recursively calls the initialization methods of every encapsulated block. The reaction method implements a reaction of the function block. It updates the input/output variables according to the event-data associations and executes the ECC. For composite function blocks, the reaction method transfers events and data between each block, and calls the reactive methods of the blocks it encapsulates. Additional methods are implemented for each algorithm used by the ECC of a function block. These algorithms do not require any translation because they are written in C. Composite function blocks, however, do not have additional methods because they do not possess an ECC.

Execution of a function block system is implemented using an infinite loop in the ‘main’ method, where each iteration is a reaction (a tick) of the system. Fig. 4 shows the main method for executing the X2Y2 system. At the start, the system interface (‘_sysInterface’) and system instance data structure (‘_var’) are declared and initialized via the initialization method (‘X2Y2init’). This method invokes the initialization method for all encapsulated function blocks in the system (‘adder’, ‘minus’ and ‘multiply’ block). Then, it executes the reaction method (‘X2Y2run’), which is responsible for updating the inputs and outputs, as well as running the function blocks in the system. After initialization, control is passed to the reaction loop, where subsequent reactions of the system will take place. Each iteration of this main loop defines a reaction of the function block system.

The WCRT of the function block system is the longest time taken for a single reaction (tick) of the system. This is the longest time needed for the initialization reaction, or the longest of the subsequent reactions of the system (the longest iteration of the reaction loop).

B. Code tagging

The generated C code has to be tagged in order for timing analysis to be done; this is similar to Kim’s approach in [7]. Time tags are added to the C code in order to maintain a mapping with the resultant assembly code. This is required because timing information can only be elicited from the assembly code. Matching the assembly code to each line of C code is unnecessary for our purpose. Instead, we look for blocks of code between control points. These control points are statements like loops, if-else statements, switch statements, and method calls.

Tags are added at both the start and at the end of each segment of code between each control point. Tags at both the start and the end of each block of code helps to identify assembly code segments at places where one may not normally be expecting assembly code to exist, for instance, between braces. However, this results in some redundancies where there may be no code between the tags.

The C code is tagged using preprocessor statements, which serve to annotate the C code differently for normal execution, model-checking purposes, and when extracting timing information from assembly. The tags to annotate the assembly are volatile assembly comments with unique numerical identifiers.
The C annotations for model-checking are also tagged with corresponding identifiers, which are used to increment a time variable. This time variable accumulates the total time taken to execute a particular path of code, and will be checked by the model-checker. An example of a tag is shown in Fig. 5.

To extract timing information of function blocks systems, each function block is compiled to assembly code with the preprocessor option ‘COMPILE = 1’. This annotates segments of assembly code with unique time tags. To obtain timing information, each opcode is mapped to its WCET in terms of clock cycles. This value is architecture-specific and can be obtained from corresponding datasheets. For our implementation, the MicroBlaze [16] processor was used.

To acquire the time taken for each block of assembly code, the cost beneath each tag is summed up (also above for the first tag at the start of each method). These values are stored with the corresponding time tag in a header file as a constant, and will be used together with the annotated C code later for WCRT analysis. These constants can be easily matched with the annotated C code as they have the same identifier names. A tool has been implemented to automate the parsing and matching of assembly. An example of tagging and obtaining the timing costs is shown in Fig. 6. In the figure, (1) shows the untagged C code for the adder block’s reactive function generated by FBC; (2) shows the tagged reactive function, where ‘#Time tag’ is a shorthand for the time tag format given in Fig. 5; (3) shows the tagged assembly code generated using GCC; and (4) shows the reactive function with the correct timing costs for each block of C code after preprocessing.

This example illustrates how the timing information for the event-data association code from the adder block’s reactive function is acquired. The timing values for the tags are stored as ‘#define time_5 19’ and ‘#define time_6 8’ in a header file. To make use of this timing information, the header file is included in the C files generated by FBC. Once the C preprocessor is executed, the C code will be annotated with the correct timing costs for each segment of code.

C. Model-checking

Before running the model-checker to obtain the WCRT, a few adjustments to the generated C code have to be made. Firstly, a global time variable needs to be inserted into the function block code. This variable is monitored by the model checker, and is incremented with the appropriate timing costs as the model checker traverses through the code.

To model-check the system, a start function is required as the model checker’s entry point. The first candidate is the main function. However, the execution of each reaction is implemented by an infinite loop (reactive systems do not terminate unless specifically requested to), which will cause the model checker to run indefinitely. To avoid using the main function as an entry function for the model checker, a function that emulates the main function without the infinite loop needs to be created. A method to obtain the longest path without iterating through the system is also needed. To
int modelcheckerEntryPoint (X2Y2Reg v1,X2Y2 s1) {
    time = 0;
    //initialization reaction
    #Time tag 162
    X2Y2 s2 //function block structure
    X2Y2Init(s2); //function block initialization
    //Run reactive function(s2) with initial values
    X2_Y2run(s2);
    assert (time <= -1); // check WCRT
    time = 0;
    //reaction with all possible inputs and states
    //for(p1) { not in entry function
    #Time tag 163
    //Read input;
    [Read input];
    //Pass input to function blocks(v1);
    X2Y2run(s1);
    //Emit output;
    #Time tag 164
    assert (time >= -1); // check WCRT
    //}
    #Time tag 165
}

Fig. 7. The model checking entry point of the X2Y2 function block system.

To achieve this, we initialize the system for every possible state, and then run it for one reaction. This is effectively the same as iterating the infinite loop through each possible state. The CBMC model checker supports this by placing the structure to store the states of function blocks as the entry function argument. Arguments of the entry function are automatically tested for every possibility by CBMC.

To emulate all possible inputs from the environment, the input vector C struct is also included as an argument to the entry function, so that CBMC may test for every possible input automatically. Fig. 7 shows an example of the emulated entry function for the X2Y2 system generated by FBC. The difference between the main function and this emulation function can be noted by comparing it to Fig. 4. It can be seen that the initialization reaction and the subsequent reaction uses different variables. This is so that the inputs and states generated by CBMC are not reset by the initialization algorithm.

Finally, assertions have to be added as WCRT check points. These are added to check if the time variable in any given path exceeds the predicted WCRT. They are to be added after each reaction of the system followed by a reset to the time variable. These are highlighted in bold in Fig. 7.

Once the modifications are made, the model checker is set to verify the assertions in the function block code starting from the entry function. The WCRT is determined in an incremental fashion by proving the predicted WCRT to be incorrect until it is no longer so. The assertion to test the predicted WCRT, (assert(time ≤ predictedWCRT)), is initially set to be less or equal to −1. Minus one is used because it is an impossible value for time, and will ensure that CBMC generates a counter-example to be used as the initial predicted WCRT. The model checker tries to prove that this assertion is incorrect by incrementing the time variable according to the tag’s timing costs, as it traverses down each path. The model checker checks through each different state combination of the system with different combinations of inputs, running the system for one reaction, until the combinations are exhausted or terminates early if the assertion is proven to fail. If the assertion is proven to fail, a counter-example is given where the total time exceeds the predicted value. This counter-example will be used as the new prediction for the WCRT, and the assertion is modified respectively. The model checker is then executed again to verify this new predicted WCRT. This is repeated until the assertion is proven to no longer fail. This final predicted value is the WCRT of the function block system.

An example to calculate the WCRT for the X2Y2 system using a model checker is shown in Table I. Initially, the predicted WCRT is −1 (assert(time ≤ −1)). The model checker verifies that this prediction is incorrect and gives the counter-example of 728. This value becomes the new predicted WCRT and the assertion is modified accordingly (assert(time ≤ 728)). After five iterations, the model checker converges on a WCRT of 1007 clock cycles.

This technique can be used to find the WCRT for individual function blocks or for an entire function block system. The only difference is in the entry function for the model checker. For individual function blocks, only one reactive function is called, whereas for an entire system, every function block’s reactive function will be called.

<table>
<thead>
<tr>
<th>Iterations</th>
<th>Predicted WCRT</th>
<th>Counter Example by Model Checker</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>−1</td>
<td>728</td>
</tr>
<tr>
<td>2</td>
<td>728</td>
<td>791</td>
</tr>
<tr>
<td>3</td>
<td>791</td>
<td>835</td>
</tr>
<tr>
<td>4</td>
<td>835</td>
<td>1007</td>
</tr>
<tr>
<td>5</td>
<td>1007 (WCRT)</td>
<td>n/a</td>
</tr>
</tbody>
</table>

<p>| TABLE I |</p>
<table>
<thead>
<tr>
<th>Iterations</th>
<th>Predicted WCRT</th>
<th>Counter Example by Model Checker</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>−1</td>
<td>728</td>
</tr>
<tr>
<td>2</td>
<td>728</td>
<td>791</td>
</tr>
<tr>
<td>3</td>
<td>791</td>
<td>835</td>
</tr>
<tr>
<td>4</td>
<td>835</td>
<td>1007</td>
</tr>
<tr>
<td>5</td>
<td>1007 (WCRT)</td>
<td>n/a</td>
</tr>
</tbody>
</table>

IV. RESULTS

To test the precision of the WCRT obtained using this approach, experimental simulations of function block examples were carried out and timed. Function block systems were set to execute the worst-case path uncovered by CBMC’s counter-example, and their WCRTs were recorded. The timed results are compared with the statically obtained values to observe the percentage of over-prediction.

The processor used for simulation is the MicroBlaze processor, with area optimization on and caches off to limit speculative behaviour. The static analysis simulation times are for an Intel i7 920 system.

Table II compares the simulated results and the statically computed results for various function block systems. An average overestimate of 9.73% can be observed. This overestimation is mainly due to branch instructions and switch statements. The number of clock cycles needed for branch instructions range from one cycle if the branch is not taken, to three clock cycles if the branch is taken [16]. However, because branches cannot be statically determined, we always assume the worst-case of three clock cycles. Another source
of over-prediction lies in the ECC code implemented in C. ECCs are implemented using switch statements. In assembly, switch statements are implemented using a jump table. Since this table cannot be annotated by assembly comments in the C code, the execution time has to be assumed to be that of the complete table. Thus, ECCs with worst-case paths close to the top of the jump table will exhibit a larger over-prediction. One simple way to improve the accuracy of WCRT estimates would be to replace switch statements with if-else-if blocks in order to avoid the generation of a jump table in assembly.

It can be seen that there is no correlation between the lines of code and the time taken for WCRT analysis. The time taken for WCRT analysis depends on the size of the state space of the C code. It can be seen that the water monitor example, with fewer lines of code than the cruise control example, requires a longer simulation time. This is because the water monitor example contains many integer variables, which produce a large state space and, hence, takes a longer time to simulate.

V. CONCLUSION

This paper presents an approach for static timing analysis of function blocks by relying on a synchronous execution semantics. The proposed approach performs model checking over time tagged C code, where the cost of the tags are obtained by deriving the number of clock cycles between two tags. These costs are obtained by examining the low-level assembly code of a given processor. The proposed approach is the first known technique to obtain very accurate worst-case reaction time (WCRT) estimate for a function block program. WCRT is important for real-time function block systems because it is the longest time required for the system to react to environmental inputs. Function block systems with verified WCRTs that are less than the rate at which the environment produces inputs are guaranteed to execute on time no matter what state the system is in.

In the future, we will look at implementing function blocks on predictable architectures [10] to further improve our estimation of WCRT.

ACKNOWLEDGMENT

The authors would like to thank Simon Yuan from the University of Auckland for helping to set up and run simulations on the MicroBlaze processor.

REFERENCES