Verifying IEC 61499 Function Blocks Using Esterel

Li Hsien Yoong and Partha S. Roop

Abstract—IEC 61499 is an international standard that prescribes the use of function blocks for designing industrial-process control systems. Function blocks enable control software to be developed using an intuitive standard’s-based graphical framework. The standard, however, lacks the semantic rigour necessary for automated verification of function block programs. Several approaches to fill this lacuna have been proposed, but these have so far focused on the verification of control properties by abstracting data from the program. This letter builds on a recent proposal to translate function blocks to Esterel in order to use the verification tools for Esterel to verify both control and data properties of function block programs. The key extensions to this translation are described herein, and have been implemented in a prototype tool. The viability of this approach is illustrated through several examples using this prototype. This demonstrates how a language with rigorous semantics and associated tools, like Esterel, can be advantageously combined with graphical notations familiar to industrial engineers to produce reliable control software.

Index Terms—Esterel, function blocks, IEC 61499, verification.

I. INTRODUCTION

The IEC 61499 [1] is a standard prescribed by the International Electrotechnical Comission for developing industrial-process control systems using function blocks. The standard facilitates a model-driven approach to development, as software components can be easily encapsulated within block diagrams with well-defined interfaces. These block diagrams can be seamlessly mixed with graphical state machines as well as conventional textual algorithms, to provide rich notations for describing programs. This enables system-level designs to be intuitively done using IEC 61499 through which, automated verification and code synthesis may be obtained. Typically, function block programs would be compiled for execution on a particular run-time environment.

The standard does not specify formal semantics, but provides only an intuitive description of the expected behaviour for function block execution. This gives rise to various ambiguities, which have been widely reported in literature [2]. These ambiguities have, hitherto, been resolved by specific implementation choices adopted by the various run-time environments, resulting in subtle differences in behaviour.

This current state of affairs complicates efforts towards the formal verification of IEC 61499 programs. In order to ameliorate this, the intuitive notions for function block execution need to be formalised. To this end, we rely on the synchronous semantics for function blocks proposed in [3]. These semantics are used to facilitate the translation of function blocks to the Esterel [4] synchronous language.

Esterel has well-defined formal semantics with industrial-grade 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. Previous attempts to verify IEC 61499 programs have also had to, first, formalise the semantics for function blocks. Vyatkin et al. [5] proposed a mapping between function blocks and Net Condition/Event Systems (NCES). This approach requires a very large number of NCES modules to represent even a small function block program, making the overheads of using this approach very high. Meanwhile, Stanica et al. [6] proposed to model function blocks as a network of timed automata. While this approach is simpler than the one in [5], it still requires the use of many automata to verify a function block program. Significantly also, both these approaches cannot model data computations, and require variables and their processing to be abstracted. This leads to overly conservative verification results.

The work in [7] is perhaps the most similar to ours. There, function block elements are mapped to the synchronous language, SIGNAL. Function block verification then relies on existing verification tools for SIGNAL. However, that work seemed to have focus only on the control aspects of function block verification. Moreover, that work did not present an automated approach to translate function block programs into SIGNAL. In contrast, we describe a prototype tool for translating complete function block programs to Esterel, and demonstrate its viability through a set of verified examples.

Our primary contributions extend [3] with:

1) a new function block compiler for verifying control and data properties in Esterel; and,
2) a new method to ensure causality in the generated code without forbidding instantaneous communication.

II. BACKGROUND

We begin with a brief overview of IEC 61499 function blocks and the Esterel language. This overview will be structured around the cruise control example, shown in Fig. 1.

A. The IEC 61499 function blocks

The modelled cruise control system has five control buttons. The set button is used to activate the cruise mode once the vehicle has reached the desired speed. This speed can then be automatically increased or decreased by 5km/h by pressing the quickAccel or quickDecel buttons respectively. If the brakes are applied, the system goes to standby and can be resumed.
with the desired speed by pressing the resume button. At any time, the cruise control system can be deactivated by pressing the off button.

Each function block has a well-defined input/output interface, with event and data lines drawn on the upper and lower parts of the block respectively. The function blocks in Fig. 1 are referred to as basic function blocks. The behaviour of a basic function block is defined by a Moore-type state machine, known as an execution control chart (ECC). Fig. 2 shows the ECC for the Throttle function block. An ECC consists of EC (execution control) states, EC transitions, and EC actions. The initial state of an ECC is represented by a double rectangle.

Function blocks adopt an event-driven model of execution. An ECC is evaluated whenever input events are received. If a transition condition evaluates to true, the ECC will transition to the next state. Actions, consisting of an algorithm and/or an output event, may be associated with each state. They will be executed upon entry to that state. A network of function blocks, like that of Fig. 1, can be composed within a composite function block. Unlike a basic function block, the behaviour of a composite function block depends on the composite behaviour of its component blocks instead of an ECC.

While these intuitive graphical notations enable systems to be described easily, arguing about their formal correctness is not easy. It is in this area that a description in a formal language, like Esterel, would be most advantageous. The reader is referred to [3] for a formal treatment of the application of synchronous semantics to function blocks.

B. The Esterel language

Fig. 3 shows the equivalent Esterel code for the ECC in Fig. 2, generated using the technique presented in Section III. The basic programming unit in Esterel is a module. Input/output signals are declared at the start of the module (lines 2–4) and may consist of a status and/or a value. Signals with only a status component are known as pure signals (e.g., input cruiseOff), while those with a value component are known as valued signals (e.g., output throttleVal).

Esterel programs execute following a clock-driven model, where time is divided into discrete instants, known as ticks. Execution within a tick is conceptually instantaneous, with time elapsing only when crossing tick boundaries. The pause statement (e.g. line 9) marks the end of a tick and pauses execution until the next instant. In each tick, input signals are sampled at the start and output signals are emitted based on the computations performed in that instant.

The await statement (e.g. lines 10–14) suspends execution until the first of its delay predicates (specified as cases) evaluates to true. The run statement (e.g. line 16) is used to instantiate and execute the code in another module. Signal emissions (e.g. line 17) are synchronously broadcast and may be tested in concurrently running modules.

Finally, the loop and switch statements enclosing most of the code are simply classical statements implementing infinite iteration and conditional selection respectively.

III. TRANSLATION OF FUNCTION BLOCKS TO ESTEREL

The mapping between function block elements and their Esterel counterparts is summarized in Table I. The goal is to map the informal execution rules in the IEC 61499 standard [1] to Esterel’s synchronous model. For instance, the standard requires that the sequence of operations from the invocation of a function block due to an input event, to the evaluation of transitions and the execution of actions, be done as an atomic operation. This can be implemented by mapping each EC state to a synchronous state in Esterel, which is demarcated using atomic operation. As exemplified in Fig. 3, the actions are emitted at each tick.
A prototype tool, called FBC, has been developed to do this translation. FBC recursively enters each composite function block it encounters, to perform a bottom-up compilation of every block in the network. The main tasks in the translation process are detailed next.

A. 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.

A depth-first traversal is used 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. Such 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 root state is automatically added to the set \( G \) to enable states in nested await-case statements to “go to” them using a state variable. States with only a single predecessor will always be nested below the await statement of its predecessor. This is illustrated in Fig. 4.

The generated code conforms to the V7 syntax used in Esterel Studio. Although Esterel Studio also allows programs to be described graphically using Safe State Machines (SSMs) 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.

B. Generating code for implementing actions

A key distinction of this work compared to that in [3] lies in the way data computations are handled. In the former work, internal variables and algorithms in function blocks were mapped to local variables and host procedures in Esterel. Host procedures are defined externally in a host language, like C, and are unknown at the Esterel level. Thus, these variables, together with the values of input/output signals, had to be explicitly passed to the host procedures in order for data computations to be performed. 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 had to be generated and passed to the procedures as well. The main drawback of this approach, however, is the resulting inability to verify data properties as host language computations are completely abstracted by Esterel’s verifier.

Due to this limitation, the current approach translates algorithms 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. The algorithms themselves may be specified either in Esterel, or in Structured Text (ST), a commonly used PLC language. FBC is able to translate a subset of ST directly to Esterel.

Input/output data and internal variables used in algorithms are passed as valued signals through signal bindings between the fbmodules and algmodules. Input data are bound as input signals, while output data and internal variables are bound as input/output signals, to allow any modification of their values in the algmodule to be propagated back to the fbmodule.

C. Ensuring causal synchronous parallel compositions

The resulting fbmodules from a function block network will be composed in parallel. The synchronous parallel composition is an intuitive, yet mathematically precise, approach for composing function blocks in a network. However, such compositions may not always be causal, especially when event paths are connected in a feedback loop to form strongly connected components among a set of function blocks. A program is said to be non-causal when its control/data dependencies cannot be correctly satisfied, resulting in the program’s deadlock.
An effective way to achieve causal composition is to restrict instantaneous communication between blocks, as proposed in [3]. This allows function blocks to be compiled modularly, as the program is guaranteed to be causal by construction. However, this results in the lost of expressivity, as instantaneous communication between blocks is no longer possible.

In contrast to [3], we allow instantaneous communication between function blocks, but still require that strongly connected blocks be “broken” by at least one delayed communication to ensure causal compositions. Fbc ensures this by doing a topological sort of all the blocks in the network. 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. Thus, the resulting Esterel code from fbc will always be causal.

IV. AUTOMATIC VERIFICATION

The Esterel program from a given function block description can be verified using synchronous observers [8]. The verification consists of checking that the product automaton from the parallel composition of the program-under-test and its observer never enters an erroneous state. As such, safety properties for the original function block program can now be expressed using function blocks itself without needing to learn a new mathematical dialect (e.g., temporal logic).

Table II shows the list of safety properties that were checked using Esterel Studio’s Design Verifier for the cruise control system, as well as for some other programs. Properties A1 and A4 were verified successfully, taking 10 and 25 seconds respectively. Property A3, however, was found to be falsifiable. Design Verifier provided a counter-example for this 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 function block programs.

V. CONCLUSION

This letter has described a tool for translating function blocks to Esterel in order to verify safety properties of function block programs 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 are completely unable to verify data properties. At present, work is on-going to develop a way to express counter-examples directly in terms of function block syntax. Recent abstraction refinement techniques [9] combining control and data abstraction will also be explored to enable programs that are larger than what Design Verifier can handle, to be verified. As reliable software becomes increasingly critical in industrial control systems, the prospects of the approach proposed here is promising.

References