A Synchronous Approach for IEC 61499 Function Block Implementation

Li Hsien Yoong, Partha S Roop, Valeriy Vyatkin and Zoran Salcic

***Abstract*— IEC 61499 has been endorsed as the standard for modelling and implementing distributed industrial-process measurement and control systems. The standard prescribes the use of function blocks for designing systems in a component-oriented approach. The execution model of a basic function block and the manner for event/data connections between blocks are described therein. Unfortunately, the standard does not provide exhaustive specifications for function block execution. Consequently, multiple standard-compliant implementations exhibiting different behaviours are possible. This not only defeats the purpose of having a standard, but makes verification of function block systems difficult. To overcome this, we propose synchronous semantics for function blocks, and show its feasibility by translating function blocks into a subset of Esterel, a well-known synchronous language. The proposed semantics avoids causal cycles common in Esterel, and is proven to be reactive and deterministic under any composition. Moreover, verification techniques developed for synchronous systems can now be applied to function blocks.**

***Index Terms*— Compilation, Esterel, function blocks, IEC 61499, synchronous semantics.**

I. Introduction

IEC 61499 [1] is an international standard that defines a component-oriented approach, based on *function blocks*, for modelling and implementing distributed industrial-process measurement and control systems. A function block abstracts a functional unit of software by encapsulating local data, state transitions, and algorithmic behaviour within a well-defined event-data interface. Fully executable systems can be described through a network of function blocks at a high level of abstraction, independent of the implementation platform. The standard, thus, paves the way for sophisticated software methodologies to be applied in the development of industrial control systems, which has hitherto, been done using low-level techniques for programmable logic controllers (PLC).

In fact, the IEC 61499 standard has emerged in response to the technological limitations encountered in the currently dominating standard, IEC 61131 [2]. IEC 61131 prescribes the use of various languages, such as Sequential Function Charts (SFC), Structured Text, and Ladder Diagrams, for programming a centralized PLC that adopts a sequential *cyclic-scan* model. In this model, a scan cycle typically involves reading all inputs, performing some computation based on the buffered inputs, and updating all outputs at the end of the cycle. This model of computation, as described in [3], is severely inadequate to meet the current industry demands for distributed, flexible automation systems. Moreover, existing tools for PLC design typically only offer simulation and code generation capability, but do not provide any means for analysis based on formal models.

To meet these challenges, IEC 61499 has defined a new *event-driven* model for function blocks intended for distributed execution, while incorporating advanced software engineering principles for component-based design to facilitate reuse. In contrast to the cyclic-scan model that executes a sequential portion of code in each cycle, the event-driven model relies on the occurrence of asynchronous events to trigger program execution. This model makes it more natural to describe control software that may need to react to multiple events concurrently. At the same time, IEC 61499 is also practically appealing as it allows the programming languages prescribed in the former standard to be encapsulated within the new function blocks to support legacy algorithms.

At present, function block implementations typically make use of a run-time environment to dispatch events among the blocks in a network to mimic the execution model given in the standard. The run-time environment provides the means to schedule function blocks for execution in response to events. Executable code is produced by compiling function blocks into appropriate objects that can be instantiated in the run-time environment.

However, the IEC 61499 standard does not provide formal semantics for the execution of function blocks. Instead, the standard contains a verbose description for function block execution, which has resulted in multiple interpretations. Consequently, function block designs may potentially behave differently when executed on the various existing run-time environments, such as FBRT [4], RTSJ-AXE [5], FORTE [6], Fuber [7] and ISaGRAF [8]. Since no rigorous semantics is available, many semantic ambiguities (see Section II for examples) and behavioural differences in various implementations have been reported in literature [9]–[11]. This not only hampers portability (defeating the purpose of a standard), but also complicates any attempts towards the automated formal verification of function block programs.

To fill this lacuna, we have developed formal semantics for function blocks based on the synchronous paradigm [12]. **This semantics view function block instances within a network as concurrently executing modules of a system.** The formal semantics proposed here will avoid the problem of multiple interpretations plaguing the current standard. More significantly, it will open the pathway for powerful compilers and verification tools already developed for synchronous systems, like Esterel Studio [13], to be used by function block designers.

The authors are with the Department of Electrical and Computer Engineering at the University of Auckland, New Zealand.

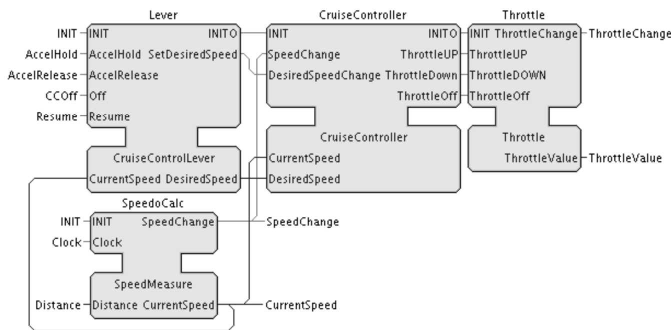

Fig. 1. A simplified function block model of a cruise control system.

We show the feasibility of this approach by describing a new compiler, FBtoStrl, that will map function block descriptions to a subset of Esterel [14], a well-known synchronous language. We give the semantic foundation for our model using operational semantic rules [15], and prove that programs constructed with this semantics are guaranteed to be *reactive* and *deterministic* [16].[1] As a consequence, the approach taken in this work yields the following benefits:

- It avoids the need for costly causality analysis [16] required in the general compilation of Esterel.
- Esterel programs can be compiled directly for execution, removing the need for run-time environments, and hence, avoiding issues of portability altogether.
- The executable code generated with the synchronous approach is 2 to 3 times faster compared to existing function block run-time environments, as the overhead for the run-time is removed.

The remainder of this paper is structured as follows. Section II introduces the nature of function block execution through an example. Then, in Section III, we give an intuitive outline for the synchronous function block model that we are proposing. Section IV is devoted to the internal details of the new compiler. Section V presents our proposed semantics, and demonstrates that all programs constructed with this semantics will be reactive and deterministic. Following that, Section VI gives the results of our work and benchmarks it against existing function block implementations. Section VII compares our work with previous attempts to introduce execution models for function blocks. Finally, the paper concludes with an outline for future extensions to this work.

II. IEC 61499 Function Blocks

Fig. 1 shows a function block model of a simplified cruise control system in a car. It consists of four function blocks, each modelling a component of the system.

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.

[1] Intuitively, we refer to reactivity as the property that ensures that a program never enters a deadlocking state, and determinism as the property which ensures that a program will always behave in the same manner in a given state for a given input. These notions are formally defined in Section V-B.

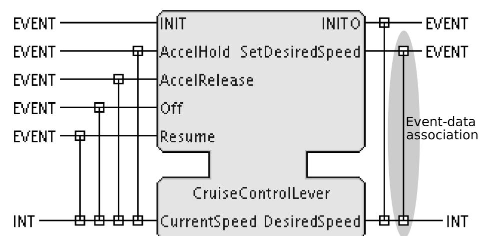

Fig. 2. The interface of the Lever function block.

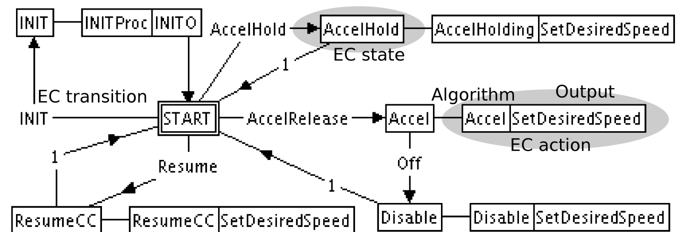

Fig. 3. The execution control chart for the CruiseControlLever function block.

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

- *composite function blocks*, which are simply a syntactic feature 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.

Every function block has an interface, which consists of a set of event and data inputs, and a set of event and data outputs, on both of its sides as illustrated through the CruiseControlLever function block 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 variables 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 and 4 for the CruiseControlLever and CruiseController function blocks respectively. 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.

An ECC basically describes a Moore-type finite state

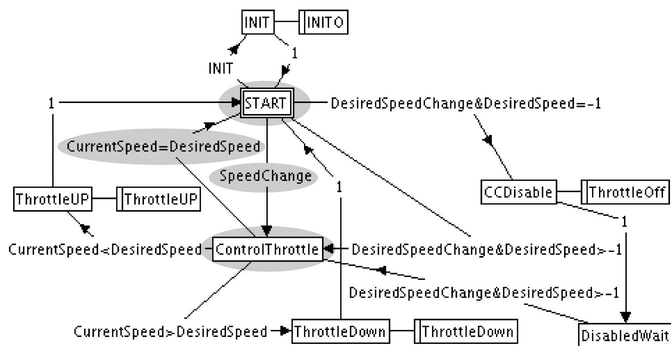

Fig. 4. The execution control chart for the CruiseController function block.

machine. Transition conditions between states are evaluated whenever the function block receives an input event. Such 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 to a given state will be executed once upon entry to the state.

Whenever an ECC is triggered by an input event, a sequence of transitions may take place within the ECC in response to that event. We refer to such a sequence of transitions as a *run* following the definition in [17]. As an example, consider the ECC in Fig. 4, and assume the current state is START. If the SpeedChange event occurs and the CurrentSpeed equals the DesiredSpeed, a run involving the sequence of transitions from START to ControlThrottle, and back, will constitute a run. The elements involved in this run have been shaded for illustration.

While the ECC is reminiscent of Statecharts [18], function blocks do have significant advantages over Statecharts. Function blocks provide a graphical approach to model a physical or logical entity within a single component with a well-defined input/output interface to the environment. This allows explicit point-to-point connections to be made between different components in a distributed system that captures the actual event-data flow between the components. This is in contrast to Statecharts' abstract model of broadcast communication, which does not map well to real-life distributed systems. Moreover, the component-oriented approach of function blocks makes the modelling of plant-controller behaviour in control systems much easier compared to doing so in Statecharts.

Despite these benefits, designers have not yet been able to exploit the full potential of function blocks due to the incomplete execution semantics given in the standard. The two main problems relate to the following:

1) *Lack of any notion of time.* Function blocks do not have any explicit notion of time. Hence, the lifetime of an event within an ECC is not clear.
2) *Lack of any notion of composition.* While individual function blocks may be connected within a network like Fig. 1, the standard does not define the composite behaviour of such a network. The standard does not specify the product state when multiple ECCs are connected in this manner. Hence, a variety of ad hoc approaches have been suggested in literature [11].

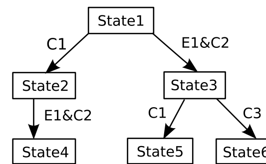

Fig. 5. Ambiguities in state transitions of execution control charts

These complications are illustrated in the following subsections.

A. State transitions within function blocks

Figure 5 shows an ECC fragment of a function block, where `E1` is an input event, and `C1`, `C2` and `C3` are boolean guard conditions. Transitions in an ECC are only evaluated with the occurrence of an input event, and are done in the order in which they have been declared in their textual syntax. Let us assume here that transitions in the figure are evaluated from left to right, and the current state is `State1`. Two ambiguities arise here:

1) Consider the case where `C1` and `C2` are both true when `E1` occurs. Consequently, the transition to `State2` will occur, but whether or not a subsequent transition to `State4` will take place is not clear. This is because the lifetime of an event is undefined in the standard.
2) Consider the case where `C2` is true, while `C1` and `C3` are both false when `E1` occurs. Then, the transition to `State3` will occur and control will remain there. In this scenario, two different interpretations are possible from the standard:
   - Transitions consisting of pure data conditions will only be evaluated *once*, upon the completion of the action in the state. Thus, the application must guarantee that "eventless" transitions have at least one guard condition that will be true upon the completion of the preceding state (e.g., `C1` and/or `C3` should be true at the end of `State3`). Otherwise, the ECC will freeze in the culpable state. This is the approach adopted by FBDK [4], a widely-accepted function block development/run-time environment.
   - "Eventless" transitions will be evaluated whenever an input event is fed to the function block. This implies a dependence on input events for the evaluation of pure boolean guard conditions. In this case, should `C1` and/or `C3` eventually become true, the transition out of `State3` will occur when the function block receives an input event.

   We have chosen to adopt the second interpretation, as the first will potentially lead to a deadlock.

With synchronous semantics, the lifetime of an event is always well-defined at every instant, while the evaluation of guard conditions in conjunction with input events can be explicitly dealt with.

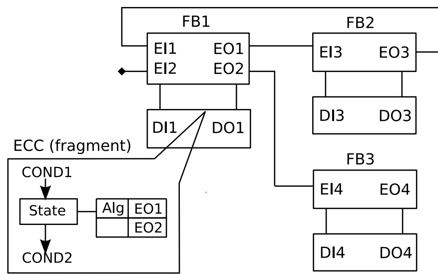

Fig. 6. Ambiguities in execution behaviour of a function block network.

```
 1:  module Simple:
 2:    input I; output Q, O : bool init false;
 3:    trap T in
 4:      loop
 5:        emit ?O <= not pre(?O);
 6:        if I then exit T end; pause
 7:      end loop
 8:    ||
 9:      loop
10:        await immediate pre(O);
11:        emit Q; pause
12:      end loop
13:    end trap
14:  end module
```

Fig. 7. Example of an Esterel program

B. Compositions of function blocks in a network

The standard does not define the composite behaviour for a network of function blocks. This results in different behaviours for a given application depending on the scheduling scheme chosen by a particular implementation.

Consider the network in Figure 6. Both `FB2` and `FB3` need to be invoked as a result of the execution of `FB1`. The decision of when to execute `FB2` and `FB3` is not **fully** clear. This ambiguity can be further compounded when there is an event feedback, like that between `FB2` and `FB1`. With ambiguous block scheduling times, such scenarios may not only result in different system behaviours, but may lead to:

- *race conditions*, as `FB1` may be triggered again by `FB2` before it can complete its current execution; and
- *starvation*, as `FB3` may be left unattended, while `FB1` and `FB2` monopolise the computational resources.

With synchronous semantics, the issues of composite behaviour and execution order are completely defined owing to synchronous parallel execution.

III. The Synchronous Approach

The synchronous programming paradigm [19] assumes an *idealized* reactive system that produces its output synchronously with its input by executing infinitely fast. This model treats time as a sequence of discrete *instants* with nothing happening between the completion of the current instant and the start of the next.

This idea is adopted by Esterel, where every program reaction occurs with respect to a logical instant of time, known as a *tick*. A new reaction is triggered at the start of each *tick* by taking a snapshot of the input signals, performing some computation, and generating the output signals before the start of the next. This can be thought of as an abstraction of a PLC's scan cycle. We highlight key features of the language through the example in Fig. 7, and depict the output trace for a particular input sequence in Table I.

TABLE I
Timing diagram for the execution of the program in Fig. 7.

Tick	Inputs	Outputs
0	-	O (true)
1	-	O (false), Q
2	I	O (true), Q *(program exits)*

The basic programming unit in Esterel is a *module*. Each module consists of an interface declaration (line 2), followed by a body of executable statements (lines 3–13). Signals in Esterel may consist of a *status* and/or a *value* component. Signals with only a status component are known as *pure* signals (e.g., signal `I`), while those with a value component are known as *valued* signals (e.g., signal `O`, which has status and a boolean value component).

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

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

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

A. Synchronous Model for Function Blocks

Our idea for synchronous function blocks has been inspired by Esterel's synchronous model. Intuitively, in our approach, each function block will be mapped to an Esterel module. Then, at the top-level, a main module will be created to connect all other modules in parallel to achieve the same event and data connections as in the function block network.

The key idea to achieve seamless composition is the following: whenever there is a producer-consumer relationship between two or more blocks, we ensure in our semantics that the producer will emit the event/data in the current instant, while the consumer will only read what is emitted in the next instant of the synchronous program. This simple "pipelining" of the producer-consumer blocks guarantees that their parallel composition will always be acyclic. This, in fact, weakens the synchrony requirements for communications between different blocks. Note that this behaviour can be obtained by reacting only to the `pre` of signals in Esterel.

While Esterel's synchronous model allows for very powerful expressions through instantaneous signal reactions, complete adherence to the synchrony hypothesis makes the compilation of Esterel complex. Every Esterel program would first need to be checked that it is deadlock-free before code can be generated for it. This checking is known as *causality analysis*, and involves ensuring that constraints arising from control and data dependencies will never lead to deadlocks [19]. This possibility of writing non-causal (deadlocking) programs makes composition difficult. For example, consider the causal programs `P1` consisting of,

```
emit U; if S then emit T end
```

and `P2` consisting of,

```
if S then emit T end; emit U
```

Both `P1` and `P2` have identical input/output behaviour. However, when each is placed in parallel with another program, `P3`, consisting of,

```
if U then emit S end
```

`P1` $\|$ `P3` remains causal, while `P2` $\|$ `P3` does not. In fact, Huizing *et al.* [20] have shown that no semantics can be *responsive* (obey the synchrony hypothesis), *causal* and *modular* (notions related to compositonality) at the same time.

The closest work related to this idea was proposed for the prototype language, SL [21]. However, there are key differences. SL allows instantaneous reactions to signal presence, but delays reactions to signal absence till the next instant. It forbids any assumptions on signal statuses: a signal can only be determined to be present if it is emitted, while its absence can only be decided at the end of the current instant. Thus, SL programs resolve signal dependencies during run-time. The execution of threads containing a test of an unresolved signal is suspended, until the signal is emitted, or until all other parallel threads pause, terminate, or become suspended as well. In this case, all unresolved signals will be recognised as absent.

In contrast, reactions in our approach always occur with respect to registered signals[2]. This avoids the need for slow run-time resolution of signal dependencies, and results in delayed reactions to both signal presence and absence. SL programs may avoid the run-time overhead by compiling to automata, but at the cost of an exponential increase in code size, and compilation complexity. Moreover, automata generation hinders modular compilation, while our semantic model makes possible the modular compilation of function blocks in true component-oriented fashion. Beside this, we have also extended our semantics to deal with data as well, which was not at all discussed in [21].

[2]The `pre` of signals in Esterel serve as registered signals.

TABLE II

FUNCTION BLOCK TO ESTEREL TRANSLATION MAP

Function block element	Esterel feature
Function blocks	Esterel modules
Events	Pure signals
Data	Value-only signals
Event-data binding	Explicitly done in Esterel code
Internal variables	Local variables in Esterel
EC states	Demarcated by `pause` statements
Algorithms	Procedure calls in host language
Transition conditions	`await` and `if` statements
Function block network	Top-level module, whereby each function block is a concurrent sub-module in it

Due to the weakening of the synchrony hypothesis, it is possible to derive a set of kernel statements for function blocks that will make it *impossible to write non-reactive or non-deterministic programs*. This avoids the need for causality analysis during compilation; thus, greatly simplifying the compiler construction. Moreover, this approach enables synchronous programming methodologies to be encapsulated within function blocks so that they can be easily understood and applied by industrial engineers who may not be familiar with synchronous languages.

B. Postulates for Function Blocks

In proposing a synchronous model for function blocks, it is desirable that the proposed semantics does not contradict what has been specified in the standard. The postulates used for the synchronous implementation of function blocks are listed here:

1) A function block *run* can only be activated with the occurrence of some input event.
2) An event has a lifetime of only a single transition, regardless of whether or not the event was actually used in the evaluation of the transition.
3) If more than one transition condition is true, they will be evaluated in the order in which they are declared.
4) The execution in EC states conceptually occur instantaneously, **with the EC actions executed in the order in which they have been declared**. This effectively treats each EC state as a synchronous state.

The standard is not clear on whether it allows function blocks to react to more than one event simultaneously (see [22] for a discussion on this). However, Esterel is able to handle multiple incoming events in the same instant. We have chosen to allow this in our implementation, and to leave the restriction to only a single event as an environment constraint.

The following section will show how function blocks can be mapped to Esterel.

IV. TRANSLATING FUNCTION BLOCKS TO ESTEREL

Table II shows the basic translation map from function blocks to Esterel. The mapping of function blocks and their event and data interfaces to Esterel modules and signals is straightforward. The encapsulation of algorithms within host

```
module CruiseControl:
    input INIT, FootBrake, AccelHold, AccelRelease, CCOff;
    input Resume, Clock, RUN, Distance : value integer;
    output ThrottleChange, ThrottleValue : value integer;
    output SpeedChange, CurrentSpeed : value integer;
    signal Lever_SetDesiredSpeed, Lever_INITO,
           Lever_DesiredSpeed : value integer, ... in
        run CruiseController [...]
    ||
        run CruiseControlLever [...]
    ||
        run SpeedMeasure [...]
    ||
        run Throttle [...]
    end signal
end module
```

Fig. 8. The top-level Esterel module that instantiates the sub-modules for the cruise control system.

language procedure calls follows the same principle used in function blocks for specifying algorithms.

The key point to note here is the use of the `pause` statement to demarcate states in an ECC. This provides the synchronous interpretation for function block execution. As a consequence, the computation of any procedure occurring in a state must fit within a *tick*.

We assume here that the *tick* is sufficiently fast to observe every event occurring in each state. This assumption is fundamental to every Esterel program, and *must* be valid for the synchrony hypothesis to hold [16]. Then, instead of evaluating transitions in an ECC with respect to an input event, transitions will now be evaluated at the occurrence of every *tick*. If the transition condition does not evaluate to true in a given *tick*, no state transition will take place. With this approach, the *lifetime of any event is explicitly bounded by the duration of the* tick *in which it occurred*.

The code generated by FBtoStrl conforms to Esterel's V7 textual syntax [14]. The Esterel Studio tool also has a graphical counterpart, called Safe State Machines (SSM). This allows state machines to be visualised in a manner similar to ECCs. However, we have chosen to stick to Esterel's textual syntax, as we found it to be more easily debug-able during the development stages of FBtoStrl, compared to the terse format required to represent SSMs. Moreover, the use of SSMs does not simplify the key algorithms of FBtoStrl in any way, as the same state and transition information would need to be extracted from a given ECC.

FBtoStrl begins the translation process by taking as input a function block network, created by a function block editor (e.g. FBDK [4]) in the XML format. 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. Each network will be instantiated within a top-level module in Esterel. If the input XML file contains networks for different resources, FBtoStrl will generate the network for each resource as a separate module.

Once every function block type in a network has been mapped into an equivalent module, the resulting modules will be composed in parallel in the top-level module created for that network. Fig. 8 shows a skeleton of how this is done for the cruise control system of Fig. 1. The event and data connections between function blocks are accomplished through appropriate signal binding at the module interfaces. Local signals in Esterel are declared to perform this binding.

The crux of the compilation actually lies in the mapping of basic function blocks to Esterel modules. This will be elaborated in the following subsection.

A. Translating the Basic Function Block

The translation steps in this stage proceed as follows:

1) Create a new module for each basic function block type.
2) Declare all input/output events of the function block as pure input/output signals at the module interface, and input/output data as value-only signals.
3) Declare all algorithms as host language procedures in Esterel.
4) Declare all internal variables in the function block as local variables in the Esterel module.
5) Extract state and transition information from the ECC to generate Esterel code.

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

- *Actions* is the list of algorithms and output events to be issued;
- *Transitions* is the list of transition conditions leading to 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, SetDesiredSpeed], Off, Disable, START}.

Once each EC state has been converted to a corresponding *SyncState*, each *SyncState* will be connected in a control-flow graph (CFG) that adheres to the transition dependencies of the original ECC. Using this CFG, the ECC can be converted to Esterel using the algorithm in Fig. 9–11.

The `TranslateECC` procedure initiates the conversion of an ECC to Esterel by invoking two recursive functions: the first function makes the ECC amenable to Esterel, while the second function generates actual code from it. The three main tasks performed by this algorithm involve:

- *Generating different types of transitions:* The event-triggered nature of ECCs can be neatly accomplished by awaiting on signals in Esterel. However, transitions involving pure data conditions need to be specially handled to create explicit dependence on incoming events.
- *Handling unstructured transitions in an ECC:* Transitions in an ECC can take place between any arbitrary EC state. However, Esterel code is highly structured, with no primitives like "goto" for handling unstructured branching. This complicates code generation as "goto" behaviour must be simulated.
- *Generating the RUN signal:* An Esterel program is triggered by a *tick* (clock-driven), and then pauses until the

```
1  procedure TranslateECC()
2    m := new module for given function block;
3    s := initial state for given ECC;
4    N := set for storing top-level states;
5    FormatECC(s,m,N);
6    if N.size() > 1 then
7      foreach n in N do
8        if n ≠ s then
9          generate if-case for n;      // simulates goto to n;
10         GenerateCode(n,N);
11       end
12     end
13   end
14   if N.size() > 1 then
15     generate default-case for s;
16   end
17   GenerateCode(s,N);
18 end procedure
```

Fig. 9. Initiates the process for translating an ECC into Esterel.

```
1  procedure FormatECC(s,m,N)
2    if s has been visited then
3      return;
4    end
5    mark s as visited;
6    foreach transition t of s do
7      if t is of type pureData then
8        if RUN input has not been created for m then
9          create RUN input signal in m;
10       end
11     end
12   end
13   if s has > 1 parents then
14     add s to N;
15   end
16   foreach child c of s do
17     FormatECC(c,m);
18   end
19 end procedure
```

Fig. 10. Prepares an ECC for structured code generation in Esterel.

```
1  procedure GenerateCode(s,N)
2    if state s has been visited then
3      return;
4    end
5    mark s as visited and generate new state boundary;
6    foreach Action a of s do
7      if a has alg algorithm then
8        generate call to procedure[alg];
9      end
10     if s has eo event output then
11       forall data output dout associated with eo do
12         generate emission of signal[dout];
13       end
14       generate emission of signal[eo];
15     end
16   end
17   if any transition from s is of type pureData then
18     foreach pureData transition, dt, in s do
19       generate test for dt;
20     end
21     generate emission of RUN signal;
22   end
23   foreach transition t of s do
24     if N.size() > 1 and t leads to child, n, in N then
25       synthesize "goto" statement to n;
26       return;
27     else
28       if t is of type pureEvent then
29         generate code to await t;
30       else
31         if t is of type pureData then
32           generate await on RUN signal or any input;
33         end
34         generate looping test for t;
35       end
36     end
37   end
38   foreach child c of s do
39     GenerateCode(c,m);
40   end
41 end procedure
```

Fig. 11. Generates Esterel code for a given ECC.

arrival of the next *tick*. In contrast, an ECC is triggered by an event (event-driven), and continues to evolve in a single *run* until no further transition is possible, as described in Section II. An additional *RUN* signal is thus required to emulate this behaviour.

These tasks are described next.

B. Generating different transition types

Fig. 11 provides the basic code generation algorithm. The Esterel code produced from this algorithm for the CruiseControlLever and the CruiseController function blocks are shown in Fig. 12 and 13 respectively. The `GenerateCode` procedure performs a depth-first traversal of the CFG to generate code for each state. Line 5 demarcates the start of each new state with the `pause` statement. Subsequently, code is generated for each action in that state in lines 6–16. If the output event issued by a given action has any associated data output, code to emit those data outputs will be generated together with the output event (lines 11–13).

In lines 28–35, each transition proceeding from the current state will be converted into an explicit test condition. We differentiate between transitions consisting only of events from those containing data conditions. Transitions consisting purely of event conditions are implemented using `await-cases` (lines 28–29). For instance, the transition conditions leading out from the START state in Fig. 3 have been translated to the following form in the equivalent Esterel code in Fig. 12:

```
await
  case immediate pre(INIT) do
    ...
  case immediate pre(AccelHold) do
    ...
  case immediate pre(AccelRelease) do
    ...
  case immediate pre(ResumeCC) do
    ...
end await
```

Meanwhile, transitions involving data conditions are implemented using `if-cases`, enclosed within a loop (lines 30–35). For example, the transition conditions leading out of the

```
module CruiseControlLever:
host procedure INITProc(in integer, inout integer, inout integer);
host procedure Accel(inout integer, inout integer);
host procedure Disable(in integer, inout integer, inout integer);
host procedure ResumeCC(inout integer, inout integer);
host procedure AccelHolding(in integer, inout integer, inout integer);
input INIT, AccelHold, AccelRelease, Off, Resume,
input CurrentSpeed : value integer;
output INITO, SetDesiredSpeed, DesiredSpeed : value integer;
var SpeedSet : integer, DesiredSpeed_var : integer in
  loop
    pause;
    await
    case immediate pre(INIT) do
      call INITProc(pre(?CurrentSpeed), DesiredSpeed_var, SpeedSet);
      emit ?DesiredSpeed <= DesiredSpeed_var;
      emit INITO; pause;
    case immediate pre(AccelHold) do
      call AccelHolding(pre(?CurrentSpeed),DesiredSpeed_var,SpeedSet);
      emit ?DesiredSpeed <= DesiredSpeed_var;
      emit SetDesiredSpeed; pause;
    case immediate pre(AccelRelease) do
      call Accel(DesiredSpeed_var, SpeedSet);
      emit ?DesiredSpeed <= DesiredSpeed_var;
      emit SetDesiredSpeed; pause;
      await immediate pre(Off);
      call Disable(pre(?CurrentSpeed), DesiredSpeed_var, SpeedSet);
      emit ?DesiredSpeed <= DesiredSpeed_var;
      emit SetDesiredSpeed; pause;
    case immediate pre(Resume) do
      call ResumeCC(DesiredSpeed_var, SpeedSet);
      emit ?DesiredSpeed <= DesiredSpeed_var;
      emit SetDesiredSpeed; pause;
    end await;
  end loop;
end var
end module
```

Fig. 12. Esterel code generated for the CruiseControlLever function block.

```
 1 module CruiseController:
 2 input INIT, SpeedChange, DesiredSpeedChange, RUN;
 3 input CurrentSpeed: value integer, DesiredSpeed: value integer;
 4 output INITO, ThrottleUP, ThrottleDown, ThrottleOff;
 5 var state : integer := 0, CurrentSpeedINT : integer in
 6 loop
 7   if state = 1 then
 8     trap T2 in loop
 9       await immediate (pre(RUN) or pre(INIT) or pre(SpeedChange)
                         or pre(DesiredSpeedChange));
10       if
11         case pre(?CurrentSpeed)<pre(?DesiredSpeed) do
12           emit ThrottleUP; pause; state = 0; exit T2;
13         case pre(?CurrentSpeed)>pre(?DesiredSpeed) do
14           emit ThrottleDown; pause; state = 0; exit T2;
15         case pre(?CurrentSpeed)=pre(?DesiredSpeed) do
16           state = 0; exit T2;
17       end if;
18       pause;
19     end loop end trap;
20   else
21     pause;
22     trap T0 in loop
23       if
24         case pre(INIT) do
25           emit INITO; pause; exit T0;
26         case pre(SpeedChange) do
27           if (pre(?CurrentSpeed)<pre(?DesiredSpeed) or
                 pre(?CurrentSpeed)=pre(?DesiredSpeed) or
                 pre(?CurrentSpeed)>pre(?DesiredSpeed)) then
28             emit RUN;
29           end if;
30           pause; state = 1; exit T0;
31         // Rest of the code is omitted
32       end if;
33       pause;
34     end loop end trap;
35   end if
36 end loop
37 end var
38 end module
```

Fig. 13. A fragment of the Esterel code generated for the CruiseController function block.

START state in Fig. 4 will result in the following form of code:

```
trap T0 in loop
  if
    case pre(INIT) do
      ...; exit T0;
    case pre(SpeedChange) do
      ...; exit T0;
    case pre(DesiredSpeedChange) and
         pre(?DesiredSpeed)>-1 do
      ...; exit T0;
    case pre(DesiredSpeedChange) and
         pre(?DesiredSpeed)=-1 do
      ...; exit T0;
  end if
end loop end trap
```

Esterel does not have constructs to wait for a data expression to become true; hence, this has to be explicitly done within a loop. The `trap` construct enclosing the loop is required to break out of the loop should any of the `if` cases succeed.

In both the `await` and `if` statements, the ordering of test cases are done following the sequence of their declaration in the XML file. Therefore, if more than one condition evaluates to true at the same time, Esterel's semantics for the `await` and `if` statements will ensure that the transition priority is correctly ordered, thus ensuring that all state transitions happen deterministically.

An important aspect in this translation is the use of the `pre` operator in all the test conditions. This idea to postpone all reactions to inputs to a function block is the key feature of our synchronous model that enables function blocks to be composed in a network without any causal problems. This effectively implements the idea of seamless composition of function blocks as described earlier in Section III-A.

C. *Handling unstructured transitions*

The control-flow among states in an ECC are, in general, unstructured. Transitions can take place between any arbitrary pair of states, making it impossible to generate structured code without excessive duplication of nodes. However, since Esterel is a highly structured language with no "goto" primitive, "goto" behaviour would need to be simulated using additional conditional statements and loops to avoid node duplication.

FBtoStrl will attempt to generate structured code whenever possible. For states with only a single predecessor, structured code will always be generated by nesting the successor state under a conditional statement in its predecessor, as exemplified in Fig. 12. However, for states with multiple predecessors (e.g., ControlThrottle in Fig. 4), a state variable will be used to encode the next state to go to (see line 30 in Fig. 13).

The algorithm accomplishes this by adding all states with multiple predecessors to some set N (see lines 13–15 in the `FormatECC` procedure of Fig. 10). Nodes in this set will be assigned with distinct indices to encode their "goto" position. Code will then be generated to test for these indices so that the appropriate state can be entered (lines 6–13 of Fig. 9). Then, whenever a transition is encountered that leads to a state in N, a "goto" to that state will be synthesize by setting the next state variable to the appropriate state index. This is accomplished during code generation in lines 24–26 of Fig. 11.

D. Generating the RUN signal

In our synchronous model, state transitions are now evaluated with the occurrence of a *tick*, rather than an input event. However, the execution model of ECCs, as described in Section II, requires that state transitions only happen when an input event occurs. Handling this requirement is not problematic for transitions that explicitly depend on some input event, as shown already in Section IV-B. However, when transitions are guarded only by pure data conditions, the dependence on input events for the transition evaluation must now be made explicit in the Esterel code. Moreover, to keep with the notion of a *run*, a new input event should not be presented to the Esterel module until that pure data transition is evaluated once. Otherwise, it is possible that fresh events will be fed into the module before the current *run* is complete.

Thus, to preserve this notion of a *run*, our algorithm automatically generates a *RUN* input at the module's interface whenever it detects that an ECC contains pure data transitions (refer to lines 6–12 of Fig. 10). The *RUN* signal will be emitted whenever a *run* needs to be sustained across a pure data transition. This is not required for states that are already dependent on some input event. Having the *RUN* as an input, which is emitted from within the module and tested again in the next instant, effectively makes it a delayed feedback signal. New inputs will not to be fed into the module in the immediate instant following the emission of the *RUN* signal.

To implement this, it is necessary to first understand how the reactive function generated by the Esterel compiler communicates with its physical environment. Typically, a *reactive interface* [23] is required to match the asynchronous environment with the synchronous model of Esterel. Concretely, the reactive interface accomplishes this by building a new input vector from the environment, feeding it to the reactive function, and emitting a new output vector, before generating a new *tick* to the module.

Therefore, to ensure that a *run* of the function block is atomic, the *RUN* signal is defined to be *mutually exclusive*[3] with all other inputs to the module. The reactive interface can enforce this requirement by ensuring that a new input tuple from the environment is never built in the immediate instant following the emission of the *RUN* signal.

Our idea of the *RUN* signal is similar to the *STEP* signal used in [24] to implement the behaviour of STATEMATE Statecharts [18] in Esterel. However, in [24], the *STEP* signal is directly mapped to Esterel's *tick*. Consequently, the reactive interface must provide the *STEP* signal not only for every triggering event that is internally generated, but also whenever there is an input from the external environment.

We illustrate the purpose of the *RUN* signal using the CruiseController ECC in Fig. 4, and its corresponding translation in Fig. 13. To emulate the behaviour of the *run* from Start to ControlThrottle, and back to Start, as explained earlier in Section II, our compiler generates code to test for the pure data conditions while still in the Start state (see lines 27–29 of Fig. 13). This is done in lines 17–22 in the `GenerateCode` procedure of Fig. 11. Then, in the ControlThrottle state, code is generated to await the *RUN* signal (line 9 of Fig. 13) before performing the tests on the various data conditions (lines 10–17). If the *RUN* signal was not previously emitted, the data conditions will only be tested with the occurrence of some input event. The explicit dependence on either the *RUN* signal or input events for the evaluation of pure data conditions is accomplished in lines 31–33 of the `GenerateCode` procedure.

[3]Esterel allows exclusion relations to be specified between input signals in its syntax.

V. Semantics of Synchronous Function Blocks

So far, we have demonstrated the feasibility of implementing a synchronous model for function blocks using Esterel. Here, we would like to present the semantics for synchronous function block execution without directly relying on Esterel. We will formally define the semantics for weakening the synchrony hypothesis in Esterel, that was already intuitively outlined earlier in Section III-A. This is done in order to prove that our proposed semantics makes it impossible to write non-reactive and non-deterministic function block programs. This guarantee simplifies the compiler implementation, and makes possible the modular compilation of function blocks, since causality analysis would no longer need to be performed over the whole program before generating code.

This section will thus introduce the set of *kernel statements* that forms the core of the proposed semantics (following the approach in [16]). Their syntax and intuitive semantics are given in Table III, while the formal semantics are presented in the following subsection.

A. Formal semantics

The formal semantics for the kernel statements are presented as program transitions using Structural Operational Semantic (SOS) rules [15] of the following form:

$$t, D \xrightarrow[I^p]{O,k} t', D'$$

where,

- t is any arbitrary composition of kernel statements;
- D is the set of values of data variables before the transition;
- O is the set of signals produced by the transition;
- k is the completion code of the transition;
- I^p is the set of signals registered in the previous instant;
- t' is the residual of t after the transition; and,
- D' is the set of values of data variables after the transition.

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

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

$$\frac{\cdots}{t, D \xrightarrow[I^p]{O,k} t', D'}$$

TABLE III

KERNEL STATEMENTS FOR SYNCHRONOUS FUNCTION BLOCKS

`nothing`	do nothing and terminate instantaneously
`pause`	pause execution till the next instant
t`;`u	run t, and then u in sequence
$t \| u$	run t and u concurrently
`loop` t `end`	repeat t forever
`emit S`	emit signal `S`
`present` $\mathtt{S}^p$ `then` t `else` u `end`	run t if the status of `S` in the previous instant (denoted by $\mathtt{S}^p$) is present; otherwise u
`trap T in` t `end`	declare and catch exception `T` in t
`exit T`	raise exception for `T`
$v := f(...)$	compute the value of f and assign it to v
`if` $c(...)$ `then` t `else` u `end`	run t if boolean function c is true; otherwise u

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

Completion codes are used to encode a given control thread's status after completing its execution, borrowing the idea from Esterel [16]. The `nothing`, `pause`, and `exit T` statements are the only ones that generate completion codes. The `nothing` statement is encoded with 0, `pause` with 1, and `exit T` with an integer ≥ 2. The completion code for statements that do not produce it will be represented by $\perp$.

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

$$syn(k,l) = \begin{cases} \perp & \text{if } k = \perp \text{ or } l = \perp \\ k & \text{if } k \geq l \\ l & \text{if } k < l \end{cases}$$

This effectively ensures that all branches of the parallel statement will terminate, pause, or exit some trap synchronously. The rewrite rules for all the kernel statements are presented next adopting the approach used in [16], [21].

1) Base statements: The `nothing` statement does nothing and terminates instantaneously.

$$\mathtt{nothing}, D \xrightarrow[I^p]{\emptyset,0} \mathtt{nothing}, D \tag{1}$$

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

$$\mathtt{pause}, D \xrightarrow[I^p]{\emptyset,1} \mathtt{nothing}, D \tag{2}$$

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

$$\mathtt{exit}\ \mathtt{T}_d, D \xrightarrow[I^p]{\emptyset,d+2} \mathtt{nothing}, D \tag{3}$$

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

$$t, D \xhookrightarrow[I^p]{E,k} t', D'$$

if there exists a transition sequence such that,

$$t, D \xrightarrow[I^p]{O_1,k_1} t_1, D_1 \xrightarrow[I^p]{O_2,k_2} \cdots \xrightarrow[I^p]{O_i,k_i} t_i, D_i \xrightarrow[I^p]{\emptyset,k} t', D'$$

where, $k \geq 0$, and $E = \bigcup_m O_m \quad \forall m \in [1,i]$.

2) Signal emission: The `emit S` statement emits the signal `S` for the current instant.

$$\mathtt{emit}\ \mathtt{S}, D \xrightarrow[I^p]{\{\mathtt{S}\},\perp} \mathtt{nothing}, D \tag{4}$$

3) Signal test: If the signal `S` was present in the previous instant, the `present` statement simply gets rewritten to its `then` branch (rule 5). Otherwise, it is rewritten to its `else` branch (rule 6).

$$\frac{\mathtt{S}^p \in I^p}{\mathtt{present}\ \mathtt{S}^p\ \mathtt{then}\ t\ \mathtt{else}\ u\ \mathtt{end}, D \xrightarrow[I^p]{\emptyset,\perp} t, D} \tag{5}$$

$$\frac{\mathtt{S}^p \notin I^p}{\mathtt{present}\ \mathtt{S}^p\ \mathtt{then}\ t\ \mathtt{else}\ u\ \mathtt{end}, D \xrightarrow[I^p]{\emptyset,\perp} u, D} \tag{6}$$

4) Data assignment: Variables can be assigned with values from an arbitrary data function. The values of variables are globally persistent. However, only the values registered in the previous instant can be read. Variable values for the current instant cannot be instantly accessed. In rule 7, D^p is the set of variable values that have been registered in the previous instant.

$$\frac{d_1^p, \ldots, d_i^p \in D^p}{v := f(d_1^p, \ldots, d_i^p), D \xrightarrow[I^p]{\emptyset,\perp} \mathtt{nothing}, D[v \leftarrow f(d_1^p, \ldots, d_i^p)]} \tag{7}$$

While no semantic restrictions are imposed on simultaneous assignments of a variable in an instant, write-write concurrency is prohibited at the syntactic level. For example, both the statements below will be rejected:

```
x:=1 || x:=2

  present S then x:=1 end
||
  present S else x:=2 end
```

The second statement illustrates the syntactic aspect of this restriction. While the assignment to x will never happen simultaneously, such programs are still, nevertheless, rejected.

5) Data test: Conditional branching can be performed based on boolean data expressions.

$$\frac{d_1^p, \ldots, d_i^p \in D^p \qquad c(d_1^p, \ldots, d_i^p) = true}{\texttt{if } c(d_1^p, \ldots, d_i^p) \texttt{ then } t \texttt{ else } u \texttt{ end}, D \xrightarrow[I^p]{\emptyset, \bot} t, D} \tag{8}$$

$$\frac{d_1^p, \ldots, d_i^p \in D^p \qquad c(d_1^p, \ldots, d_i^p) = false}{\texttt{if } c(d_1^p, \ldots, d_i^p) \texttt{ then } t \texttt{ else } u \texttt{ end}, D \xrightarrow[I^p]{\emptyset, \bot} u, D} \tag{9}$$

6) Sequential statement: Rule 10 expresses the fact that the sequence does not finish, if its left branch, t, does not.

$$\frac{t, D \xrightarrow[I^p]{O, \bot} t', D'}{t;u, D \xrightarrow[I^p]{O, \bot} t';u, D'} \tag{10}$$

If the left branch pauses, so does the sequence.

$$\frac{t, D \xrightarrow[I^p]{\emptyset, 1} t', D}{t;u, D \xrightarrow[I^p]{\emptyset, 1} t';u, D} \tag{11}$$

Moreover, if the left branch raises an exception, its right branch will never get executed.

$$\frac{t, D \xrightarrow[I^p]{\emptyset, k} t', D \qquad k \geq 2}{t;u, D \xrightarrow[I^p]{\emptyset, k} t', D} \tag{12}$$

Otherwise, control will be immediately transferred to the right branch, u, when t finishes.

$$\frac{t, D \xrightarrow[I^p]{\emptyset, 0} t', D}{t;u, D \xrightarrow[I^p]{\emptyset, \bot} u, D} \tag{13}$$

7) Parallel statement: Rule 14 uses the completion code synchronizer to specify the overall behaviour of the parallel statement. During unfinished transitions, the execution of t and u may possibly be interleaved. When they both perform finished transitions, the parallel statement synchronizes their execution using their completion codes.

$$\frac{t, D \xrightarrow[I^p]{O, k} t', D_{f(\ldots)} \qquad u, D \xrightarrow[I^p]{Q, l} u', D_{g(\ldots)}}{t \| u, D \xrightarrow[I^p]{O \cup Q, syn(k,l)} t' \| u', D' \text{ where } D' = D_{h(f(\ldots), g(\ldots))}} \tag{14}$$

As already mentioned in Section V-A.4, write-write concurrency on variables is disallowed, while read-write concurrency is semantically forbidden by rule 7. This means that the functions f and g in rule 14 cannot operate on any of the same variables in the same instant. Hence, D' can be treated as a single data store of h, consisting of the possibly interleaved store order of f and g.

8) Loop: The loop simply rewrites into a sequence of its body with the loop itself.

$$\texttt{loop } t \texttt{ end}, D \xrightarrow[I^p]{\emptyset, \bot} t;\texttt{loop } t \texttt{ end}, D \tag{15}$$

As a consequence, loop bodies that finish with the completion code of 0 will result in the undesired rewriting into an infinite sequence of unfinished transitions. Such instantaneous loops can be rejected by insisting that the body's reaction never terminates instantaneously, as required by rule 16.

$$\frac{t, D \xhookrightarrow[I^p]{E, k} t', D' \qquad k > 0}{\texttt{loop } t \texttt{ end}, D \xhookrightarrow[I^p]{E, k} t';\texttt{loop } t \texttt{ end}, D'} \tag{16}$$

9) Exception declaration: Rule 17 expresses the fact that the trap statement does not terminate if its body performs an unfinished transition.

$$\frac{t, D \xrightarrow[I^p]{O, \bot} t', D'}{\texttt{trap T in } t \texttt{ end}, D \xrightarrow[I^p]{O, \bot} \texttt{trap T in } t' \texttt{ end}, D'} \tag{17}$$

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

$$\frac{t, D \xrightarrow[I^p]{\emptyset, 1} t', D}{\texttt{trap T in } t \texttt{ end}, D \xrightarrow[I^p]{\emptyset, 1} \texttt{trap T in } t' \texttt{ end}, D} \tag{18}$$

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

$$\frac{t, D \xrightarrow[I^p]{\emptyset, k} t', D \qquad k = 0 \text{ or } k \geq 2}{\texttt{trap T in } t \texttt{ end}, D \xrightarrow[I^p]{\emptyset, \bot} \texttt{nothing}, D} \tag{19}$$

B. Definitions and proofs

Two properties that are of interest in synchronous function block programs are *reactivity* and *determinism*.

Definition 1: A program is reactive if, for any statement t and data set D, there exists at least one reaction for the set of signals registered in the previous instant, I^p.

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

$$\forall t, D, \quad \exists k \geq 0 \quad \text{such that} \quad t, D \xhookrightarrow[I^p]{E, k} t', D'$$

Proof: The proof can be shown by a structural induction on t. The base statements are easily verified, since rules 1, 2, and 3 imply that the following reactions are valid:

$$\texttt{nothing}, D \xhookrightarrow[I^p]{\emptyset, 0} \texttt{nothing}, D$$

$$\texttt{pause}, D \xhookrightarrow[I^p]{\emptyset, 1} \texttt{nothing}, D$$

$$\texttt{exit T}_d, D \xhookrightarrow[I^p]{\emptyset, d+2} \texttt{nothing}, D$$

Subsequently, only the sequential, parallel, and trap statements are of interest, since all other statements complete as unfinished transitions. For the trap statement, rule 18 is the

only one that performs a finished transition. However, since the trap body can only pause if it consists of the `pause` statement, or a sequential or parallel statement containing `pause`, only two cases remain of interest:

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

$$q, D \xhookrightarrow[I^p]{E_q, k_q} q', D' \qquad k_q \geq 0 \tag{20}$$

and,

$$r, D \xhookrightarrow[I^p]{E_r, k_r} r', D' \qquad k_r \geq 0 \tag{21}$$

Then,

- if $k_q \geq 2$, we simply get $q;r, D \xhookrightarrow[I^p]{E_q, k_q} q', D'$;
- if $k_q = 1$, we have, $q;r, D \xhookrightarrow[I^p]{E_q, 1} q';r, D'$;
- otherwise if $k_q = 0$, we have, $q;r, D \xhookrightarrow[I^p]{E_q \cup E_r, k_r} r', D'$.

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

2) Next, consider the case of $t = q\|r$. Due to hypotheses (20) and (21), rule 14 will then ensure that $q\|r$ will eventually finish, since both its branches will eventually perform finished transitions. Thus, any *parallel composition of kernel statements will also always be reactive*. ■

Definition 2: A program is deterministic if for any statement t and data set D, there exists at most one reaction for the set of signals registered in the previous instant, I^p.

Lemma 1: If $t, D \xrightarrow[I^p]{O_1, \perp} t_1, D_1$, then there exists no t_2, D_2 such that $t, D \xrightarrow[I^p]{O_2, k_2} t_2, D_2$, where $k_2 \geq 0$. Conversely, if $t, D \xrightarrow[I^p]{O_2, k_2} t_2, D_2$, where $k_2 \geq 0$, then there exists no t_1, D_1 such that $t, D \xrightarrow[I^p]{O_1, \perp} t_1, D_1$. Moreover, there is only one way to finish a reaction:

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

Proof: The proof can be shown by a structural induction on t. This is easily verified for the base statements, since they each only have one rewrite rule. Therefore, *finished transitions are always distinct, and mutually exclusive with unfinished transitions*. ■

Lemma 2: The rules are strongly confluent. Suppose $t, D \xrightarrow[I^p]{O_1, k_1} t_1, D_1$ and $t, D \xrightarrow[I^p]{O_2, k_2} t_2, D_2$. Then, there exists t', D', O, k such that $t_1, D_1 \xrightarrow[I^p]{O, k} t', D'$ and $t_2, D_2 \xrightarrow[I^p]{O, k} t', D'$.

Proof: Proof by structural induction on t. As before, only kernel statements for the sequence and parallel operator are of interest:

1) Consider first, $t = q;r$. Lemma 1 provides two inductive cases:
   a) If q terminates, $t_1 = t_2 = r$, $D_1 = D_2$, $O_1 = O_2$, and $k_1 = k_2$. Consequently, the rewriting of t_1, D_1 and t_2, D_2 will both yield the same resultant of t', D', O, k.
   b) Otherwise, $t_1 = q_1;r$, $D_1 = D_{q1}$, and $t_2 = q_2;r$, $D_2 = D_{q2}$. Then, using lemma 2 as the induction hypothesis, we get $q_1, D_{q1} \xrightarrow[I^p]{O, k} q', D_q$ and $q_2, D_{q2} \xrightarrow[I^p]{O, k} q', D_q$. Then, we obtain the rewrite rules for t_1, D_1 and t_2, D_2 as: $q_1;r, D_{q1} \xrightarrow[I^p]{O, k} q';r, D_q$ and $q_2;r, D_{q2} \xrightarrow[I^p]{O, k} q';r, D_q$ respectively.
2) Next, consider the case where $t = q\|r$. Suppose that,

$$q, D \xrightarrow[I^p]{O, k} q', D_{f(\dots)} \quad \text{and} \quad r, D \xrightarrow[I^p]{Q, l} r', D_{g(\dots)}$$

Then, from rule 14, we get

$$q\|r', D_{f(\dots)} \xrightarrow[I^p]{O \cup Q, syn(k,l)} q'\|r', D_{h(f(\dots), g(\dots))}$$

and

$$q'\|r, D_{g(\dots)} \xrightarrow[I^p]{O \cup Q, syn(k,l)} q'\|r', D_{h(f(\dots), g(\dots))}.$$

■

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

$$\forall t, D, \quad \forall E, k, t', D', \quad \forall F, l, t'', D'',$$
$$t, D \xhookrightarrow[I^p]{E, k} t', D' \quad \textit{and} \quad t, D \xhookrightarrow[I^p]{F, l} t'', D'' \quad \Rightarrow$$
$$E = F,\ k = l,\ t' = t'',\ \textit{and}\ D' = D''.$$

Proof: Suppose we have the situation where

$$t, D \xrightarrow[I^p]{O_1, \perp} \cdots \xrightarrow[I^p]{O_i, \perp} t_i, D_i, \text{ and } t_i, D_i \xrightarrow[I^p]{O, k} t', D',$$

where $k \geq 0$, as well as

$$t, D \xrightarrow[I^p]{Q_1, \perp} \cdots \xrightarrow[I^p]{Q_j, \perp} t_j, D_j, \text{ and } t_j, D_j \xrightarrow[I^p]{Q, l} t'', D'',$$

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

$$t_i, D_i \xrightarrow[I^p]{O_{i+1}, \perp} \cdots \xrightarrow[I^p]{O_{i+m}, \perp} t_j, D_j,$$

or

$$t_j, D_j \xrightarrow[I^p]{Q_{j+1}, \perp} \cdots \xrightarrow[I^p]{Q_{j+n}, \perp} t_i, D_i.$$

Lemma 1 then requires that $t_i = t_j$, $D_i = D_j$, and $O_i = Q_j$, and hence, $t' = t''$, $D' = D''$, $k = l$, and $O = Q$. ■

VI. RESULTS

We ran experiments to evaluate the performance of the function block programs obtained from the Esterel code generated by FBtoStrl. **Since there are no other synchronous implementations of function blocks available, we benchmarked the generated code against FBDK [4], a free and widely-accepted function block development kit.**

FBDK relies on a function block run-time (FBRT) environment in order to dispatch events among various function blocks. The code from FBtoStrl was compiled using the V7 Esterel compiler in Esterel Studio [13]. The suite of programs used for benchmarking range from small examples (about a hundred lines of code) to real life models (consisting of thousands of lines of code). The baggage conveyor program

TABLE IV
EXECUTION TIME FOR ONE MILLION CYCLES (IN MILLISECONDS)

Programs	FBRT	Esterel	Speedup factor
LED flasher	285	118	2.41
Speed regulator	255	129	1.98
Drill station	310	170	1.82
Cruise control	961	318	3.02
Distributed mutex	4786	1768	2.71
Baggage conveyor	25400874	7720630	3.29

in Table IV is a realistic model of part of an airport baggage handling system.

For this experiment, a set of pseudorandom input vectors were generated for each benchmark program. Separate testbenches were created to feed these input vectors in FBRT and to the reactive function generated by the Esterel compiler. The measured times do not include the time to run the testbench, so that the use of different testbenches in the two environments 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.

Table IV shows the average time taken to execute one million cycles of the benchmark programs. These times were measured on an AMD Turion 64 ML-32 processor with 1GB of RAM. The Esterel code from FBtoStrl consistenly ran faster compared to its counterpart in FBRT.

The significant speedup factor of 2 to 3 times clearly demonstrates the viability of translating function blocks into our synchronous model. **This model allows function block programs to be executed without the need of a run-time environment, as all scheduling decisions would have already been determined at compile-time. The removal of the run-time consequently results in substantial gains in the execution speed.** This approach, moreover, now opens up the possibility for performing observer-based verification [25] of function block programs within Esterel Studio.

VII. Related Work

Earlier attempts for a formal model of IEC 61499 function blocks have advocated the use of Net Condition/Event Systems (NCES) [17]. NCES is a formalism based on Petri nets that was originally intended for modelling discrete event systems. While there exists tools capable of verifying NCES [26], the NCES model itself is limited by its inability to handle data computation. Other models that use state-transition formalisms, like interacting automata [10] and timed automata [27], do not provide adequate support for handling data as well. The automata model in [10] is restricted to state transitions that do not involve data conditions, while that in [27] is even more limited by requiring that all data variables and their processing be abstracted out.

Others, like Dubinin *et al.* [28], have proposed a new semantics dedicated to function blocks, independent of other formalisms. The model adopted here allows function block networks to be subsequently verified as closed-loop systems using Prolog [29]. We have not followed the semantics here, however, as this model assumes a sequential execution of blocks in a network. This limitation is awkward for a standard intended for distributed systems. However, an even greater drawback of a sequential model is that it provides no means for compositional properties of function blocks to be studied. Without a notion of a product state for a combination of ECCs in a network, composition problems like that discussed in Section II-B may easily arise.

While all these methods mentioned so far are useful for verifying certain properties over a given model, it is significant to note that none of these approaches are able to automatically transfer the verified model to actual executable code. This is where a formal model that adopts the synchronous approach is advantageous, as they can readily exploit powerful compilers that already exists for synchronous languages.

A previous work [30] did attempt to map function blocks to a synchronous framework as well, but used SIGNAL [31] instead. We translate function blocks to a subset of Esterel, preferring its imperative syntax over SIGNAL's declarative style for describing event-oriented reactive systems, like function blocks. More significantly, our choice to delay all signal reactions in Esterel guarantees the reactivity and determinism of all function block programs, without requiring costly causality analysis. This is comparable to the semantics of STATEMATE Statecharts [18], where outputs in the current step are sensed only in the next step. Delayed reactions also make it potentially easier for distributed implementations, as suggested in [32], since it effectively makes communication non-instantaneous.

One consequence of delayed reactions is that it makes possible the separate compilation of Esterel programs. Separate compilation of synchronous programs has been known, in general, to be difficult [19]. However, by delaying all signal reactions to the next instant, the control-flow of a given thread can no longer be affected by other threads running in parallel. Therefore, complete knowledge of the system would no longer be required during compilation, as threads can be arbitrarily scheduled in each instant. Such modularity in compilation is of practical importance to industrial engineers, who would be inclined to treat function blocks simply as opaque reusable components in their designs.

In fact, the SOS rules presented in Section V may be viewed as a slightly more fundamental work that contributes to earlier proposals for weakening the synchrony hypothesis, along the lines of what has been done in [21]. By trading off the compositional expressivity afforded by instantaneous reactions, we are able to achieve separate compilation and guaranteed acyclicity. We believe that the inability to react to signals instantaneously is a minor limitation in the function block domain, which has hitherto, not even had a formal notion of composition. Moreover, while our work here has been presented in the context of function blocks, the semantics itself may find application in other domains requiring similar features.

There have also been previous attempts to introduce the synchronous approach in the industrial control systems domain for the programming languages of the earlier IEC 61131 standard [33]. The work in [34] proposed a mapping for SFCs to Esterel. While some simple translation rules were sketched out in [34], that work did not seem to have any

automated means for converting an arbitrary SFC to Esterel. The faithfulness of the Esterel translation to the original SFC description was also not thoroughly treated. We have proposed a synchronous model for the IEC 61499 function blocks, and have developed a prototype compiler that can automatically generate Esterel code from function blocks, while guaranteeing their causal correctness in an arbitrary network.

VIII. Conclusion and Future Directions

This paper has presented a synchronous approach for implementing IEC 61499 function blocks. This approach gives precise execution semantics to function blocks. Consequently, various ambiguities that has plagued function block implementations, and the need for a run-time environment to execute them, have been avoided.

For the first time, Esterel code can be automatically generated from a function block description using our prototype compiler. The execution times using this approach has achieved a **significant speedup** over the current approach for executing function blocks. Moreover, we have also introduced a synchronous semantics that will guarantee the reactivity and determinism of function block programs, irrespective of their composition. This has been achieved by restricting the ability for instantaneous signal reactions in exchange for easier compositionality.

This proposed approach greatly simplifies the compilation of synchronous programs, as costly causality analysis would no longer be required. In fact, we intend to apply this semantics to create a variant of FBtoStrl that can generate C code directly from function block descriptions, by-passing Esterel altogether. Consequently, this will enable us to produce code that is reactive and deterministic by construction.

The proposed synchronous model also paves the way for observer-based verification of function block programs, as mentioned in Section VI. We are at the moment investigating this possibility using a combination of assertions and observers in Esterel Studio for a large industrial model that we are developing as a case study.

Our compiler is currently only capable of generating fully synchronous code with Esterel. However, another potential approach for implementing function block systems would be to adopt the Globally Asynchronous Locally Synchronous (GALS) model [35]. The work in [35] provides a method to automatically derive distributed code for a GALS implementation directly from a synchronous program, like Esterel.

Alternatively, a GALS model can be directly generated by our compiler using a GALS language, like SystemJ [36]. This can be done as a simple modification to the back-end code generated by our compiler. SystemJ adopts the GALS model of computation and allows synchronous programs to be described in an imperative manner similar to Esterel. At the same time, it also provides constructs to abstractly model asynchronous communication, which Esterel does not. Communication between distributed function blocks may potentially be done through asynchronous channels, while execution within a single resource can be kept synchronous.

Future work combining a mixture of synchronous and asynchronous implementations of function blocks seems highly likely, as it will provide the flexibility for distribution over a variety of networks. Challenges in the verification of such systems will be a key research area for future investigation.

References

[1] *International Standard IEC 61499-1: Function blocks – Part 1: Architecture*, 1st ed., International Electrotechnical Commission, Geneva, January 2005.

[2] *International Standard IEC 61131-3: Programmable Controllers – Part 3: Programming Languages*, 2nd ed., International Electrotechnical Commission, Geneva, 2003.

[3] V. Vyatkin, *IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design*. USA: ISA, 2007, pp. 28–29.

[4] *Function Block Development Kit*, Holobloc Inc., http://www.holobloc.com (Last Accessed: 26/9/2006), July 2005.

[5] K. Thramboulidis and A. Zoupas, "Real-Time Java in Control and Automation: A Model Driven Development Approach," in *10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Catania, September 2005, pp. 39–46.

[6] *4DIAC-RTE (FORTE): IEC 61499 Compliant Runtime Environment*, PROFACTOR Produktionsforschungs GmbH, http://www.fordiac.org (Last Accessed: 30/10/2007), 2007.

[7] *Function Block Execution Runtime (Fuber)*, http://sourceforge.net/projects/fuber (Last Accessed: 26/9/2006).

[8] J. Chouinard and R. Brennan, "Software for Next Generation Automation and Control," in *4th IEEE International Conference on Industrial Informatics (INDIN)*, Singapore, August 2006, pp. 886–891.

[9] C. Sünder, A. Zoitl, J. H. Christensen, V. Vyatkin, R. W. Brennan, A. Valentini, L. Ferrarini, T. Strasser, J. L. Martinez-Lastra, and F. Auinger, "Usability and Interoperability of IEC 61499 based distributed automation systems," in *4th IEEE International Conference on Industrial Informatics (INDIN)*, Singapore, August 2006, pp. 31–37.

[10] G. Čengić, O. Ljungkrantz, and K. Åkesson, "Formal Modeling of Function Block Applications Running in IEC 61499 Execution Runtime," in *11th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Prague, September 2006.

[11] L. Ferrarini and C. Veber, "Implementation approaches for the execution model of IEC 61499 applications," in *2nd IEEE International Conference on Industrial Informatics (INDIN)*, Berlin, Germany, June 2004, pp. 612–617.

[12] L. H. Yoong, P. Roop, V. Vyatkin, and Z. Salcic, "Synchronous Execution of IEC 61499 Function Blocks Using Esterel," in *5th IEEE International Conference on Industrial Informatics (INDIN)*, Vienna, July 2007, pp. 1189–1194.

[13] *Esterel Studio User Manual, Version 5.2*, Esterel Technologies SA, November 2004.

[14] *The Esterel v7 Reference Manual: Version v7_30 for Esterel Studio 5.3*, Esterel Technologies SA, Villeneuve-Loubet, France, December 2005.

[15] G. D. Plotkin, "A structural approach to operational semantics," *Journal of Logic and Algebraic Programming*, vol. 60-61, no. 1, pp. 17–140, 2004.

[16] G. Berry, *The Constructive Semantics of Pure Esterel (Draft Book)*. http://www-sop.inria.fr/esterel.org (Last Accessed: 25/9/2006): Available online, 1999.

[17] V. Vyatkin, "Execution Semantic of Function Blocks based on the Model of Net Condition/Event Systems," in *4th IEEE International Conference on Industrial Informatics (INDIN)*, Singapore, August 2006.

[18] D. Harel and A. Naamad, "The STATEMATE semantics of statecharts," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 4, pp. 293–333, October 1996.

[19] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone, "The Synchronous Languages 12 Years Later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, January 2003.

[20] C. Huizing and R. Gerth, "Semantics of Reactive Systems in Abstract Time," in *Proceedings of the Real-Time: Theory in Practice, REX Workshop*. London, UK: Springer-Verlag, 1992, pp. 291–314.

[21] F. Boussinot and R. de Simone, "The SL Synchronous Language," *IEEE Transactions on Software Engineering*, vol. 22, no. 4, pp. 256–266, April 1996.

[22] V. Vyatkin and V. Dubinin, "Sequential Axiomatic Model for Execution of Basic Function Blocks in IEC 61499," in *5th IEEE International Conference on Industrial Informatics (INDIN)*, Vienna, July 2007, pp. 1183–1188.

[23] C. André, F. Boulanger, and A. Girault, "Software Implementation of Synchronous Programs," in *Proceedings of the Second International Conference on Application of Concurrency to System Design (ACSD)*. Newcastle upon Tyne: IEEE Computer Society, June 2001, pp. 133–142.

[24] S. A. Seshia, R. K. Shyamasundar, A. K. Bhattacharjee, and S. D. Dhodapkar, "A Translation of Statecharts to Esterel," in *World Congress on Formal Methods (2)*, ser. LNCS, A. Sangiovanni-Vincentelli and J. Sifakis, Eds., vol. 1709. Toulouse: Springer-Verlag, September 1999, pp. 983–1007.

[25] N. Halbwachs, F. Lagnier, and P. Raymond, "Synchronous Observers and the Verification of Reactive Systems," in *AMAST '93: Proceedings of the Third International Conference on Methodology and Software Technology*. London: Springer-Verlag, 1994, pp. 83–96.

[26] V. Vyatkin, H.-M. Hanisch, and T. Pfeiffer, "Object-oriented modular place/transition formalism for systematic modeling and validation of industrial automation systems," in *IEEE International Conference on Industrial Informatics (INDIN)*, Banff, August 2003, pp. 224–232.

[27] M.-P. Stanica and H. Guéguen, "A Timed Automata Model of IEC 61499 Basic Function Blocks Semantic," in *Euromicro European Conference on Real-Time Systems (ECRTS)*, Porto, July 2003.

[28] V. Dubinin and V. Vyatkin, "Towards a Formal Semantic Model of IEC 61499 Function Blocks," in *4th IEEE International Conference on Industrial Informatics (INDIN)*, Singapore, August 2006.

[29] V. Dubinin, V. Vyatkin, and H.-M. Hanisch, "Modelling and Verification of IEC 61499 Applications using Prolog," in *11th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Prague, September 2006.

[30] C. Schnakenbourg, J.-M. Faure, and J.-J. Lesage, "Towards IEC 61499 Function Block Diagrams Verification," in *IEEE International Conference on Systems, Man and Cybernetics*, Hammamet, October 2002.

[31] P. L. Guernic, T. Gautier, M. L. Borgne, and C. L. Maire, "Programming Real Time Applications with SIGNAL," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1321–1336, September 1991.

[32] F. Boniol, "Synchronous Communicating Reactive Processes," in *2nd AMAST Workshop on Real-Time Systems*, Bordeaux, June 1995.

[33] F. Jiménez-Fraustro and E. Rutten, "A synchronous model of IEC 61131 PLC languages in SIGNAL," in *13th Euromicro Conference on Real-Time Systems (ECRTS)*, Delft, June 2001, pp. 135–142.

[34] C. André and M.-A. Péraldi, "GRAFCET and Synchronous Languages," *APII*, vol. 27, no. 1, pp. 95–205, 1993.

[35] A. Girault and C. Ménier, "Automatic Production of Globally Asynchronous Locally Synchronous Systems," in *2nd International Workshop on Embedded Software (EMSOFT)*, ser. LNCS, A. Sangiovanni-Vincentelli and J. Sifakis, Eds., vol. 2491. Grenoble: Springer-Verlag, October 2002, pp. 266–281.

[36] F. Gruian, P. Roop, Z. Salcic, and I. Radojevic, "The SystemJ Approach to System Level Design," in *4th International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, Napa, California, July 2006.

Li Hsien Yoong is a PhD candidate in the Department of Electrical and Computer Engineering at the University of Auckland in New Zealand. His primary research interests include design languages and their compilation, system level modelling and verification, distributed software systems, and related areas in embedded systems design. Yoong has an ME in computer systems engineering from the University of Auckland, and a BE in electronic engineering from the Multimedia University in Malaysia.

Partha S Roop is a Senior Lecturer in the Department of Electrical and Computer Engineering at the University of Auckland in New Zealand. He has a PhD in Computer Science from the University of New south Wales, Sydney, Australia (2000), an MTech from the Indian Institute of Technology, Kharagpur, India (1992) and a BE from the College of Engineering, Anna University, Madras, India (1989). His primary research interests are in the design and verification of embedded systems. Of particular interest to him are formal verification techniques such as model checking and module checking and their applications in embedded systems. Roop is on the editorial boards of Elsevier journal of Microprocessors and Microsystems and EURASIP Journal on Embedded Systems.

Valeriy Vyatkin is a Senior Lecturer in the Department of Electrical and Computer Engineering at the University of Auckland, New Zealand. He holds Dr. Sci. degree (1998) and Ph.D. (1992) from Taganrog State University of Radio Engineering (TSURE), Taganrog, Russia, and Dr. Eng. (1999) from Nagoya Institute of Technology, Nagoya, Japan, in 1999. His previous faculty positions were with Martin Luther University of Halle-Wittenberg in Germany (1999-2004), and with TSURE (research assistant, assistant professor in 1989-1994 and professor in 1999-2002). The main research interests of Dr. Vyatkin are in the area of industrial informatics, including software engineering for industrial automation systems, distributed software architectures, methods of formal validation of industrial automation systems and theoretical algorithms for improving their performance. The specific expertise area of Dr. Vyatkin is in distributed automation and the IEC 61499 standard. He is the author of over 160 international publications and of several software tools.

Zoran Salcic is a professor of computer systems engineering at the University of Auckland. He has a BE (1972), an ME (1974), and a PhD (1976) degree in electrical engineering from the University of Sarajevo. He did most of his PhD research at the City University of New York (CUNY). His main research interests include complex digital systems design, custom-computing machines, reconfigurable systems, FPGAs, processor and computer systems architectures, embedded systems and their implementation, design automation tools for embedded systems, hardware-software co-design, new computing architectures and models of computation for heterogeneous embedded systems, and related areas in computer systems engineering. Salcic is editor-in-chief of EURASIP Journal on Embedded systems. He is a fellow of the Royal Society New Zealand (Academy of Science) and senior member of the IEEE.